



User Manual

SCADE Suite User Manual

CONTACTS

Legal Contact

Ansys France
15 place Georges Pompidou
78180 Montigny-le-Bretonneux FRANCE
Phone: +33 1 30 60 15 00
Fax: +33 1 30 64 19 42

Technical Support

Ansys France
Parc Avenue, 9 rue Michel Labrousse
31100 Toulouse FRANCE
Phone: +33 5 34 60 90 50
Fax: +33 5 34 60 90 41

Submit questions to Ansys SCADE Products Technical Support at scade-support@ansys.com.

Contact one of our Sales representatives at scade-sales@ansys.com.

Direct general questions about SCADE products to scade-info@ansys.com.

Discover latest news on our products at www.ansys.com/products/embedded-software.

LEGAL INFORMATION

Copyrights © 2024 ANSYS, Inc. All rights reserved. Ansys, SCADE, SCADE Suite, SCADE Display, SCADE Architect, SCADE LifeCycle, SCADE Test, and Twin Builder are trademarks or registered trademarks of ANSYS, Inc. or its subsidiaries in the U.S. or other countries. SCADE Suite User Manual – Published October 2024.

All other trademarks and tradenames contained herein are the property of their respective owners.

TERMS OF USE

Important! Read carefully before starting this software and referring to its user documentation. This publication, as well as the software it describes, is distributed as part of the SCADE user documentation under license and may be used or copied only in accordance with the terms and conditions of the Software License Agreement (SLA) accepted during SCADE product installation. Except as permitted by the SLA, the content of this publication cannot be reproduced, stored, or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior permission of Ansys. Any existing artwork or images that you may want to reuse in your projects may be protected under copyright law. The unauthorized use of such material into your own work may constitute a violation of Ansys copyrights. If needed, make sure you obtain the written permission from Ansys. By starting the software you expressly agree to the following conditions:

- **Property:** This software is and remains the exclusive property of Ansys. It cannot be copied or distributed without written authorization from Ansys. Ansys retains title and ownership of the software.
- **Warranty:** The software is provided, as is, without warranty of any kind. Ansys does not guarantee the use, or the results from the use, of this software. All risk associated with usage results and performance is assumed by you the user.

DOCUMENTATION DISCLAIMER

The content of SCADE products user documentation is distributed for informational use only, is subject to change without notice, and should not be construed as a commitment by Ansys. Although every precaution has been taken to prepare this manual, Ansys assumes no responsibility or liability for any errors that may be contained in this book or any damages resulting from the use of the information contained herein.

THIRD-PARTY LEGAL INFO

The Legal Notice (PDF) about third-party software can be found under the help folder of the SCADE installation.

Shipping date: 17/10/24

Revision: SCS-UM-25 - DOC/rev/41852-10

User Manual Overview

This manual provides user-oriented instructions that help using and mastering the tools integrated into SCADE Suite application.

[“Introduction”](#)

Part 1: [“Design and Model Editing”](#)

- Chapter 1: [“Fundamentals about SCADE Suite Editor”](#)
- Chapter 2: [“Setting Project and Design Environment”](#)
- Chapter 3: [“Designing Models”](#)
- Chapter 4: [“Designing with Predefined Operators and Constructs”](#)
- Chapter 5: [“Navigating in Models”](#)
- Chapter 6: [“Editing Properties of Design Elements and Objects”](#)
- Chapter 7: [“Computing Metrics and Checking Rules on Model”](#)
- Chapter 8: [“Change Control of Models”](#)
- Chapter 9: [“Managing Model Refactoring”](#)
- Chapter 10: [“Running Coverability Analysis on Design”](#)

Part 2: [“Simulation and Debugging”](#)

- Chapter 11: [“Fundamentals about SCADE Suite Simulation”](#)
- Chapter 12: [“Running Simulation in SCADE Suite”](#)
- Chapter 13: [“Running Simulation from External Tools”](#)
- Chapter 14: [“Running Advanced Simulation”](#)
- Chapter 15: [“Generating Desktop and Mobile Applications”](#)

Part 3: [“Verification”](#)

- Chapter 16: [“Running Formal Verification Sessions”](#)
- Chapter 17: [“Verifying Timing and Stack Usage”](#)

Part 4: [“Code Generation”](#)

- Chapter 18: [“Fundamentals about SCADE Suite Code Generation”](#)
- Chapter 19: [“Setting and Launching Code Generation”](#)
- Chapter 20: [“Implementing Models with Imported Code”](#)
- Chapter 21: [“Integrating Monolithic Code”](#)
- Chapter 22: [“Generating and Integrating Task-Based Code”](#)

Appendices and Index:

- Appendix A: [“SCADE Suite in Command Line”](#)
- Appendix B: [“SCADE Suite Files in Configuration Management”](#)
- Appendix C: [“SCADE Suite and Source Code Control”](#)
- Appendix D: [“Compiler Configuration for SCADE Suite Code Generation”](#)

[“Index”](#)

RELATED DOCUMENTS

- *SCADE Suite Technical Manual*
- *SCADE Architect User Manual*
- *SCADE LifeCycle User Manual*
- *SCADE Test User Manual*
- *SCADE Display User Manual*
- *SCADE UA Page Creator User Manual*
- *Scade Language Reference Manual*

TYPOGRAPHICAL CONVENTIONS

Bold	User interface buttons, options, tabs, radio buttons, drop-down lists, window titles, panels.
<i>Italics</i>	Path for menu selection from main menu bar or in contextual menus, new technical concepts or words, references to product manuals.
Courier New	Code extracts, constant or variable names, file names, directory path, or typed text.
Title Case	Names of application components, menus.
UPPERCASE	File formats

Table of Contents

User Manual Overview

Introduction	1
SCADE Suite Environment Toolset	2
SCADE Suite Editor	2
SCADE Suite Simulation	3
SCADE Suite Design Verifier	3
SCADE Suite Timing and Stack Optimizer	3
SCADE Suite Timing and Stack Verifiers	4
SCADE Suite KCG Code Generator	4
SCADE Suite Multicore Code Generator	4
SCADE Display® Integration	5
SCADE Suite Simulink® Importer	5
SCADE Suite Gateway for NI VeriStand	5
Real-Time Operating System Wrappers	6
SCADE Configuration Management	6
SCADE Test Integration	6
SCADE LifeCycle® Integration	7
Launching/Closing SCADE Suite Environment	8
Documentation and Online Help	9
Displaying SCADE Suite Startup Page	9
Accessing SCADE Suite Help Resources	10
Navigating in Online Help Interface	11
Accessing Contextual Help Resources	12
Getting Information about Product Version	12
Getting Online Support	13

Table of Contents

Part 1: Design and Model Editing

1. Fundamentals about SCADE Suite Editor	1 - 19
Model-Based Design Principles	1 - 19
Design Architecture Modeling: Operators	1 - 20
Data Structuring: Types and Constants	1 - 21
Control Formalism: State Machines	1 - 21
Requirements Management	1 - 27
Model Coverability Analysis	1 - 27
Exploring SCADE Suite Editor User Interface	1 - 28
Perspectives	1 - 29
Docking Windows	1 - 29
Model Editing Views	1 - 40
Toolbars	1 - 45
Symbol Editor	1 - 50
Status Bar Display	1 - 50
Model Development Workflow	1 - 51
Standard File and Project Management	1 - 53
Opening/Closing Files	1 - 53
Opening/Closing Workspace Files	1 - 54
Saving Files and Workspace Files	1 - 56
Opening Recently Opened Projects and Files	1 - 59
Canceling/Restoring Previous Actions	1 - 59
SCADE Suite Files	1 - 60
Workspace and Project Files	1 - 60
Libraries Files	1 - 61
SCADE Files	1 - 62
Scenario and Application State Files	1 - 62
Simulation Output and Observability Files	1 - 63
Semantic Error File	1 - 63

Table of Contents

Annotation Type Files	1 - 63
Annotation Files	1 - 64
Symbol Files	1 - 64
Font Files	1 - 65
Option Files	1 - 65
Reporting Files	1 - 65
Log Files	1 - 65
Multifile Management	1 - 66
Default File Management	1 - 66
File Management Customization	1 - 68
Locating Objects from SCADE Files	1 - 71
2. Setting Project and Design Environment	2 - 73
Setting SCADE Suite Tools Options	2 - 74
Selecting Environment Options	2 - 74
Selecting General Options	2 - 77
Selecting Views Options	2 - 79
Selecting Commands Options	2 - 82
Selecting Language Construct Options	2 - 83
Selecting Advanced Options	2 - 84
Customizing Tools Menu	2 - 89
Managing Perspectives	2 - 92
Default Perspectives	2 - 92
Switching Perspectives	2 - 93
Creating New Perspectives	2 - 93
Managing Projects	2 - 95
Creating Blank Workspace	2 - 95
Creating New Project	2 - 96
Creating New Python Files	2 - 100
Creating New Tcl Files	2 - 101

Table of Contents

Switching Active Project in Same Workspace	2 - 102
Renaming Projects	2 - 102
Managing Project Archives	2 - 103
Inserting Existing Project or Files	2 - 112
Creating New Folders in Projects	2 - 114
Saving Files in UNIX Text Format	2 - 114
Managing Libraries	2 - 115
Inserting Library in Projects	2 - 115
Replacing Libraries in Projects	2 - 116
Setting Library Path for Sharing Projects	2 - 116
Managing Default or Custom Configurations	2 - 117
Default Configurations	2 - 117
Enabling Configurations	2 - 119
Creating and Setting Configurations	2 - 121
Editing Default and Custom Configurations	2 - 123
Organizing Model Design with Packages	2 - 124
Creating New Packages	2 - 125
Opening Packages	2 - 128
Managing Visibility of Package Declarations	2 - 130
Managing Favorites and Using Shortcuts	2 - 131
Creating and Modifying Styles	2 - 134
Creating User-Defined Styles	2 - 135
Applying Styles in Diagrams	2 - 138
Modifying Style Definitions	2 - 141
Deleting Styles	2 - 142
Enabling Display Options in Diagram Views	2 - 143
Organizing and Working with Model Editing Views	2 - 144
Managing Views in Multi-Tab Display	2 - 144
Setting Model Editing Views for Multi-Screen Display	2 - 147

Table of Contents

Duplicating Tabbed Views	2 - 147
Closing Tabbed Views	2 - 148
Grabbing and Moving Design Representation	2 - 148
Zooming Design Content in Views	2 - 148
Underlining Truncated Text in Views	2 - 149
Displaying Active View on Full Screen	2 - 150
Setting Grid Properties to Assist Design Positioning	2 - 152
Printing Active Views	2 - 153
Organizing and Working with Docking Windows	2 - 154
3. Designing Models	3 - 159
Declaring Design Elements in Model	3 - 160
Declaring Model Operators	3 - 160
Declaring Operator Interface	3 - 162
Declaring Operator Local Data	3 - 164
Creating and Designing State Machines	3 - 168
Creating and Designing If Blocks	3 - 180
Creating and Designing When Blocks	3 - 188
Declaring and Editing Imported Operators	3 - 194
Declaring and Editing Specialized Operators	3 - 196
Setting States/Actions in Embedded/Hidden Mode	3 - 199
Splitting Operator Design in Multiple Diagrams	3 - 204
Inserting Existing Item	3 - 206
Defining Data Types	3 - 207
Declaring New Types	3 - 207
Setting Array Types	3 - 209
Setting Enumeration Types	3 - 211
Setting Structure Types	3 - 212
Completing Definition of Structured Types	3 - 213
Setting Imported Types	3 - 214

Table of Contents

Declaring Operators Using Generic Types	3 - 216
Defining Constants	3 - 218
Declaring New Constants	3 - 218
Setting Array/Structure Type Constants	3 - 220
Setting Imported Constants with External Code	3 - 225
Setting Imported Constants with Imported Types	3 - 227
Defining Sensors	3 - 230
Declaring New Sensors	3 - 230
Setting Design Instances in Diagram Views	3 - 232
Creating Operator Calls	3 - 233
Creating Calls to Predefined Operators	3 - 234
Creating Predefined Operator Symmetric Calls	3 - 234
Adding/Removing Predefined Operator Inputs	3 - 235
Composing/Decomposing Structured Types	3 - 236
Substituting Operator and Variable Calls	3 - 238
Creating References to Inputs/Outputs	3 - 239
Creating References to Local Data	3 - 240
Creating References to Constants or Sensors	3 - 242
Creating Contracts: Assume/Guarantee	3 - 242
Creating Terminators	3 - 243
Creating Lasts of Variables and Signals	3 - 243
Setting Default Values on Variables	3 - 245
Connecting Data Flow Elements	3 - 246
Connecting Multiple Data Flow Elements	3 - 248
Defining Value of Hidden Inputs on Operators	3 - 250
Encapsulating Design in Operators or States	3 - 252
Setting and Defining Equation Sets	3 - 254
Setting Operator Parameterized by Size	3 - 257
Designing Model with Textual Definitions	3 - 259

Table of Contents

Defining Operators with Textual Diagrams	3 - 259
Defining Textual States or Actions	3 - 266
Creating Instances from Textual Files	3 - 270
Creating Textual Expressions	3 - 272
Managing Instance Names Textually	3 - 273
Checking Model Semantics	3 - 274
Fundamentals about Checker	3 - 274
Launching Semantic Checks	3 - 275
Modifying Design Elements	3 - 279
Moving Design Elements in Models	3 - 279
Modifying Definitions from File View	3 - 281
Deleting Declarations from Scade View	3 - 282
Optimizing Design Layout	3 - 283
Selecting Design Elements	3 - 283
Aligning Elements	3 - 284
Aligning Design Components	3 - 284
Distributing Elements and Connection Wires	3 - 288
Resizing Elements	3 - 288
Resizing Elements Manually	3 - 290
Rotating Elements	3 - 290
Nudging Elements	3 - 291
Optimizing Connection Wires	3 - 291
Editing Connection Wires and State Transitions	3 - 292
Straightening Connection Wires	3 - 293
Snapping Design Elements	3 - 293
Editing Design Elements with Symbol Editor	3 - 294
Editing Layout of Graphical Symbols	3 - 295
Shaping Graphical Symbols	3 - 297
Drawing Graphical Symbols	3 - 298

Table of Contents

Editing and Inserting Text in Symbols	3 - 299
Inserting Images in Symbols	3 - 300
Positioning Name on Variable Symbols	3 - 300
Shifting Symbol Foreground or Background	3 - 301
Developing Control and Display Software	3 - 302
Joint Development Workflow	3 - 304
Inserting Graphical Panels in Project	3 - 305
Connecting Operator and Graphical Panel Interfaces	3 - 306
4. Designing with Predefined Operators and Constructs	4 - 315
Designing with Arithmetic Operators	4 - 316
Setting and Designing Additions	4 - 317
Setting and Designing Subtractions	4 - 318
Setting and Designing Multiplications	4 - 319
Setting and Designing Polymorphic Divisions	4 - 320
Setting and Designing Modulos	4 - 321
Setting and Designing Unary Minus	4 - 322
Setting and Designing Numeric Cast	4 - 323
Designing with Bitwise Arithmetic Operators	4 - 324
Setting and Designing Bitwise AND	4 - 325
Setting and Designing Bitwise OR	4 - 326
Setting and Designing Bitwise XOR	4 - 327
Setting and Designing Bitwise NOT	4 - 328
Setting and Designing Logical Left Shift	4 - 329
Setting and Designing Logical Right Shift	4 - 330
Designing with Choice Operators	4 - 331
Setting and Designing If Then Else	4 - 331
Setting and Designing Case	4 - 332
Designing with Comparison Operators	4 - 336
Setting and Designing Strictly Less Than	4 - 337

Table of Contents

Setting and Designing Less Than or Equal	4 - 338
Setting and Designing Strictly Greater Than	4 - 339
Setting and Designing Greater Than or Equal	4 - 340
Setting and Designing Different	4 - 341
Setting and Designing Equal	4 - 342
Designing with Logical Operators	4 - 343
Setting and Designing AND	4 - 344
Setting and Designing OR	4 - 345
Setting and Designing XOR	4 - 346
Setting and Designing Sharp	4 - 347
Setting and Designing NOT	4 - 348
Designing with Temporal Operators	4 - 349
Setting and Designing Init	4 - 350
Setting and Designing Previous	4 - 352
Setting and Designing Followed By	4 - 353
Setting and Designing Times	4 - 355
Setting and Designing When	4 - 356
Setting and Designing Merge of Flows	4 - 359
Designing with Structured Values	4 - 362
Setting and Designing Data Structure	4 - 362
Setting and Designing Make or Flatten	4 - 363
Designing with Array Values	4 - 367
Setting and Designing Arrays by Enumeration	4 - 368
Setting and Designing Arrays by Repetition	4 - 369
Setting and Designing Array Indexation Access	4 - 370
Setting and Designing Array Static Slice Access	4 - 373
Setting and Designing Permutation of Array Values	4 - 374
Setting and Designing Array Concatenation	4 - 374
Setting and Designing Arrays by Copy with Modification	4 - 375

Table of Contents

Setting and Designing Array Transposition	4 - 376
Designing with Array Iterators	4 - 378
Definition for Map Iteration Pattern	4 - 379
Definition for Mapi Iteration Pattern	4 - 381
Definition for Mapw Iteration Pattern	4 - 383
Definition for Mapwi Iteration Pattern	4 - 385
Definition for Fold Iteration Pattern	4 - 387
Definition for Foldi Iteration Pattern	4 - 389
Definition for Foldw Iteration Pattern	4 - 391
Definition for Foldwi Iteration Pattern	4 - 393
Definition for MapFold Iteration Pattern	4 - 395
Definition for MapFoldi Iteration Pattern	4 - 397
Definition for MapFoldw Iteration Pattern	4 - 398
Definition for MapFoldwi Iteration Pattern	4 - 399
Creating and Setting Iteration Patterns	4 - 400
Designing with Operator Activation	4 - 402
Setting and Designing Boolean Activation	4 - 402
Setting and Designing Clocked Activation	4 - 405
Designing with Restart Instantiation	4 - 408
5. Navigating in Models	5 - 409
Finding and Locating Design Elements	5 - 409
Searching for Design Dependencies	5 - 412
Searching for Design Element Definitions	5 - 412
Searching for Design Element References	5 - 413
Searching for Producers or Deep Producers	5 - 413
Searching for Consumers or Deep Consumers	5 - 414
Searching for Operator Calls	5 - 416
Searching for Operator Callers	5 - 416
Navigating in Model Content	5 - 417

Table of Contents

Navigating in Model and Package Hierarchy	5 - 417
Navigating in Output Window	5 - 418
Navigating in Views	5 - 419
Navigating to Declaration from Diagrams	5 - 419
Navigating to Type Definition	5 - 419
Navigating between Opened Views and Files	5 - 420
Navigating in HTML Documents	5 - 420
Filtering Model Content	5 - 421
6. Editing Properties of Design Elements and Objects	6 - 423
Editing Project Properties	6 - 424
Editing File Folder Properties	6 - 430
Editing Project Files Properties	6 - 431
Editing Package Properties	6 - 434
Editing Diagram Properties	6 - 436
Editing Type Properties	6 - 438
Editing Constant Properties	6 - 440
Editing Sensor Properties	6 - 442
Editing Operator Properties	6 - 444
Editing Operator Call Properties	6 - 448
Editing Use Properties of Operator Calls	6 - 448
Editing Parameters of Operator Calls	6 - 451
Editing Size Parameters of Operator Calls	6 - 452
Editing Input/Output Properties	6 - 453
Editing Probe Properties	6 - 456
Editing Local Variable Properties	6 - 459
Editing Signal Properties	6 - 462
Editing If Block and If Node Properties	6 - 464
Editing When Block and When Branch Properties	6 - 467

Table of Contents

Editing State Machine Properties	6 - 470
Editing State Properties	6 - 471
Editing Transition Properties	6 - 474
Editing Wire Properties	6 - 477
Editing Assertion Properties	6 - 480
Editing Textual Expression Properties	6 - 481
Changing Design Element Layout	6 - 482
Adding and Editing Comments	6 - 483
Adding and Editing Notes	6 - 484
Editing Instance Names	6 - 487
Accessing Traceability Information	6 - 488
7. Computing Metrics and Checking Rules on Model	7 - 489
Metrics and Rules Checker Overview	7 - 489
Purpose of Metrics and Rules Checker	7 - 489
Metrics Computation and Rules Checker Fundamentals	7 - 490
Setting Metrics and Rules Checker	7 - 498
Setting Active Metrics and Rules	7 - 498
Managing Project Metrics and Rules	7 - 501
Computing Metrics and Checking Rules	7 - 503
Computing Metrics	7 - 503
Checking Rules	7 - 505
Creating User-Defined Metrics and Rules	7 - 510
Defining Python Metrics	7 - 510
Defining Python Rules	7 - 516
8. Change Control of Models	8 - 525
Fundamentals about Model Diff	8 - 525
Semantic Model/Object Differences	8 - 526
Analysis and Merge of Differences	8 - 526

Table of Contents

Reports about Model and Object Differences	8 - 528
Analyzing Model and Object Differences	8 - 529
Launching Model Comparison	8 - 529
Launching Object Comparison	8 - 531
Examining Semantic Differences	8 - 532
Locating Differences in Diagrams and Views	8 - 534
Generating Diff Reports	8 - 536
Merging Differences between Models	8 - 538
9. Managing Model Refactoring	9 - 541
Model Refactoring	9 - 541
List and Visualization of Refactoring Issues	9 - 542
Applying Quick Fixes to Refactoring Issues	9 - 544
Applying User Fixes to Inconsistencies	9 - 545
Launching Resolve Inconsistencies Wizard	9 - 545
Resolving Missing Declarations	9 - 546
Resolving Operator Interface Inconsistencies	9 - 548
Resolving Type Inconsistencies	9 - 550
Resolving Constant Value Inconsistencies	9 - 552
Generating Inconsistencies Resolution Report	9 - 553
10. Running Coverability Analysis on Design	10 - 555
Coverability Analysis Diagnostics and Report	10 - 555
Setting Coverability Analysis	10 - 557
Launching Coverability Analysis	10 - 559
Launching Coverability Analysis Batch	10 - 564

Table of Contents

Part 2: Simulation and Debugging

11. Fundamentals about SCADE Suite Simulation	11 - 569
Model-Based Simulation Principles	11 - 570
Simulation Architecture	11 - 570
Observable Variables and Traceability	11 - 571
Simulation Cycles	11 - 572
SCADE Simulation Executables	11 - 572
Step-by-Step or Continuous Simulation	11 - 572
Simulation Scenarios/States: Storing and Replaying	11 - 573
Breakpoints	11 - 573
Exploring Simulator User Interface	11 - 574
Instances Model Browser	11 - 575
Simulation Views	11 - 576
Simulation Window	11 - 577
Watch Window	11 - 577
Graph Window	11 - 578
Simulation Toolbars	11 - 579
Simulation Workflow	11 - 580
12. Running Simulation in SCADE Suite	12 - 583
Setting Simulation Environment	12 - 583
Setting Project for Simulation Sessions	12 - 585
Setting Simulation Code for Simulation Sessions	12 - 586
Setting Basic Options for Simulation Code	12 - 587
Enabling Simulator Code Generation	12 - 587
Setting Simulation Binary Generation	12 - 589
Enabling Simulation with Graphical Extensions	12 - 592
Enabling Partial Model Simulation	12 - 593
Setting Variable Visibility	12 - 594

Table of Contents

Running Model Simulation	12 - 596
Launching Simulation Session	12 - 596
Executing Simulation Cycles	12 - 597
Recording Simulation Scenarios/Application States	12 - 614
Setting and Managing Breakpoints	12 - 616
13. Running Simulation from External Tools	13 - 623
Co-Simulation Principles	13 - 624
Simulating Models with Graphical Panels	13 - 625
Workflow for Simulation from Graphical Panels	13 - 625
Enabling Simulation with Graphical Panels	13 - 626
Managing Model Simulation from Graphical Panels	13 - 628
Simulating Models in FMI-Compliant Tools	13 - 629
Code Generation Workflow for FMU Simulation	13 - 629
Setting Sensors as Calibration Parameters	13 - 631
Setting Code Generator for FMU Simulation	13 - 633
Exporting SCADE Suite Model as FMU	13 - 636
Importing Generated Files in FMI-Compliant Tools	13 - 640
Running Simulation with Distributed FMUs	13 - 641
Simulating Models in Twin Builder Environment	13 - 647
Code Generation Workflow for Twin Builder Simulation	13 - 647
Editing Operator Properties for Twin Builder	13 - 650
Setting Code Generator for Twin Builder Simulation	13 - 651
Exporting SCADE Suite for Twin Builder Simulation	13 - 654
Importing Generated Files in Twin Builder	13 - 654
Importing FMU for Parametric Analysis	13 - 655
Simulating Models in Simulink Environment	13 - 656
Code Generation Workflow for Simulink Co-Simulation	13 - 657
Setting Simulink Code Wrapping	13 - 659
Creating S-Function Dynamic Library/SFW Library Model	13 - 663

Table of Contents

Launching Co-Simulation with Simulink	13 - 665
Executing MATLAB Files	13 - 666
Advanced Tuning	13 - 667
Simulating Models in VeriStand Environment	13 - 671
14. Running Advanced Simulation	14 - 673
Simulating Models with External C Code	14 - 673
Simulating Models with Imported C Types	14 - 675
Implementing C Macro User Definitions	14 - 676
Implementing Supported Type Definition	14 - 677
Implementing Unsupported Type Definition	14 - 680
Implementing Unsupported Macro Definition	14 - 680
Implementing Imported Type Definition	14 - 681
Declaring Type/Macro Definition in Projects	14 - 681
Simulating Models with User C-Macros	14 - 684
Simulating Models with External Ada Code	14 - 685
Using Simulation API for Co-Simulation	14 - 687
Setting Co-Simulation Compilation Resources	14 - 688
Instrumenting Application for Co-Simulation	14 - 688
Running Double Simulation Sessions	14 - 694
Enabling/Launching Double Simulation Mode	14 - 694
Using Watch in Double Simulation Mode	14 - 696
Using Graph in Double Simulation Mode	14 - 696
Handling Breakpoints, Assertions and Exceptions	14 - 697
15. Generating Desktop and Mobile Applications	15 - 699
Application Generation Principles	15 - 700
Generating Desktop Applications	15 - 701
Setting Windows Executable Generation	15 - 701
Generating Windows Executable	15 - 704

Table of Contents

Running Generated Windows Executable	15 - 705
Customizing Windows Executable	15 - 706
Generating Mobile Applications	15 - 707
External Tools Requirements for Mobile Generation	15 - 707
Setting Mobile Application Generation	15 - 708
Launching Mobile Application Generation	15 - 712
Building Mobile Application	15 - 713

Part 3: Verification

16. Running Formal Verification Sessions	16 - 717
Overview of Design Verifier	16 - 717
Purpose of Design Verifier	16 - 718
Design Verifier Workflow	16 - 721
Exploring Design Verifier User Interface	16 - 722
Generated Output Files	16 - 728
Fundamentals in Design Verification	16 - 729
SCADE Arithmetic	16 - 729
Property Expression	16 - 729
Verifiable Properties	16 - 731
User Proof Objectives	16 - 732
Predefined Proof Objectives	16 - 732
Strategies	16 - 734
Setting Design Verifier Environment	16 - 736
Creating Verification Project	16 - 736
Enabling Design Verifier from SCADE Suite Projects	16 - 737
Configuring User Property Verification	16 - 738
Expressing User Property and Context	16 - 738
Specifying User Proof Objectives	16 - 740
Applying Predefined Properties	16 - 742

Table of Contents

Specifying Predefined Proof Objectives	16 - 742
Setting Predefined Proof Objectives Properties	16 - 744
Specifying User-Defined Strategies	16 - 746
Enabling Analysis Using Abstraction	16 - 749
Organizing Design Verifier Analysis Objects	16 - 752
Organizing Proof Objectives in Folders	16 - 752
Editing Properties of Task Folders	16 - 753
Organizing Strategies in Folders	16 - 754
Associating Strategies with Proof Objectives	16 - 754
Documenting Verification Projects	16 - 755
Launching Property Analyses	16 - 756
Interpreting Property Analyses Results	16 - 758
Reading Basic Result Messages	16 - 758
Displaying Analysis Information in Reports	760
Launching Verification Scenarios from Reports	16 - 761
Displaying Counter-Examples in Scenario Viewer	16 - 763
Defining Custom Strategies	16 - 765
Configuring Prover PSL SAT-Solvers	16 - 766
Defining Analysis Strategies	16 - 770
Composing Strategies and Supported Use Cases	16 - 775
Description of Strategies Algorithm	16 - 779
Illustrated Examples	16 - 782
17. Verifying Timing and Stack Usage	17 - 785
Overview of Timing and Stack Analysis Tools	17 - 785
Purpose of Timing and Stack Analysis Tools	17 - 786
Timing and Stack Analysis Workflow	17 - 787
Exploring Timing and Stack Analysis User Interface	17 - 788
Fundamentals about Timing and Stack Tools	17 - 795
Timing Analysis	17 - 795

Table of Contents

Stack Analysis	17 - 795
Analysis Session and Results	17 - 795
Timing and Stack Analysis Tools and KCG	17 - 795
Setting Timing and Stack Verification	17 - 797
Setting Timing and Stack Result Options	17 - 797
Setting Code Generation for Timing and Stack	17 - 798
Building Binary for Timing and Stack Analysis	17 - 802
Setting Timing and Stack Analysis Tools	17 - 803
Setting Advanced Timing and Stack Analysis	17 - 805
Launching Timing and Stack Analysis Tools	17 - 809
Launching Timing or Stack Analysis	17 - 809
Launching Analysis in Interactive Mode	17 - 810
Examining Timing and Stack Analysis Results	17 - 811
Displaying Analysis Session Results	17 - 811
Displaying Timing Analysis Detailed Results	17 - 814
Locating Analyzed Nodes in Model	17 - 816
Sorting Analysis Results	17 - 816
Importing External Results	17 - 818
Updating Analysis Sessions	17 - 819
Comparing Analysis Sessions	17 - 819

Part 4: Code Generation

18. Fundamentals about SCADE Suite Code Generation	18 - 823
Code Generation Principles	18 - 823
Generation of Monolithic or Task-Based Code	18 - 823
SCADE Suite Input Models	18 - 825
Code Generation Files	18 - 825
Optimization of Generated Code	18 - 826
Assertions in Generated Code	18 - 826

Table of Contents

Probes in Generated Code	18 - 827
Code Generation Checks	18 - 827
Exploring Code Generator User Interface	18 - 829
Code Generation Toolbar	18 - 829
Code Generator Settings	18 - 830
Code Generation Capabilities	18 - 830
Display of Build and Log Information	18 - 832
Code Generation Workflow	18 - 833
19. Setting and Launching Code Generation	19 - 835
Specifying Code Generation	19 - 836
Setting Optimization Options	19 - 839
Setting Debug Options	19 - 840
Setting Code Configuration Options	19 - 842
Setting Code Expansion Mode	19 - 847
Setting Code Integration Capabilities	19 - 849
Setting Periodicity of Code Execution	19 - 852
Setting Compiler and Build Options	19 - 853
Adjusting Variable Name Length for Compiler	19 - 856
Setting I/O Separation Options	19 - 857
Setting Pragmas on Model Elements	19 - 859
Launching KCG Code Generation from UI	19 - 864
Launching KCG Executable in Command Line	19 - 866
20. Implementing Models with Imported Code	20 - 869
Generating Templates for Imported Operators	20 - 870
Implementing Imported C Code	20 - 871
Implementing C Code for Imported Functions	20 - 872
Implementing C Code for Imported Nodes	20 - 874
Implementing Imported Ada Code	20 - 877

Table of Contents

21. Integrating Monolithic Code	21 - 879
Integrating C/Ada Code in Software Application	21 - 879
List of KCG-Generated Files for Module Integration	21 - 880
Writing Main Function/Procedure	21 - 881
Module Integration with C Code	21 - 882
Module Integration with Ada Code	21 - 884
Generating and Integrating SCADE Display Code	21 - 885
Launching Co-Generation of Display Applications	21 - 885
Integrating Wrapping C Code with Display Application	21 - 888
22. Generating and Integrating Task-Based Code	22 - 889
Task-Based Code Generation Principles	22 - 890
Model Annotation for Task Identification	22 - 890
Task-Based Generated Code	22 - 891
Task Allocation to Computing Resources	22 - 896
Task-Based Code Generation Workflow	22 - 897
Annotating Models for MCG Code Generation	22 - 898
Specifying Task Pragmas	22 - 898
Declaring Iterator Instances as Unrolled	22 - 904
Setting Task-Based Code Generation	22 - 905
Launching Task-Based Code Generation	22 - 905
Launching MCG from User Interface	22 - 905
Launching MCG Executable in Command Line	22 - 905
Computing Task WCET on MCG-Generated Code	22 - 907
Implementing Code for Multicore Targets	22 - 910
Allocating and Scheduling Generated Tasks	22 - 910
Implementing Integration Code	22 - 911

Table of Contents

Appendixes	913
A - SCADE Suite in Command Line	A - 915
General Command Line Syntax	A - 915
Command Line Options	A - 917
SCADE Suite Launch Command Line	A - 917
Viewer Mode Command Line	A - 917
Locate Command Line	A - 918
Semantic Check Command Line	A - 918
Metrics and Rules Checker Command Line	A - 918
Timing and Stack Verifiers Command Line	A - 919
Design Verifier Command Line	A - 919
Code Generator Command Line	A - 920
Scripts Command Line	A - 920
Model Diff Command Line	A - 921
Archive Export/Import Command Line	A - 921
B - SCADE Suite Files in Configuration Management	B - 923
Files Unnecessary for Configuration Management	B - 923
Files Necessary for Configuration Management	B - 924
C - SCADE Suite and Source Code Control	C - 927
Introduction	C - 927
SCC Integration Principles	C - 928
Setting SCADE Suite Configuration Management	C - 929
Enabling Source Code Control Integration	C - 930
Setting Options for Source Code Control	C - 931
Using Source Code Control from Projects	C - 932
Source Control Toolbar	C - 933
Adding/Removing from Source Control	C - 935

Table of Contents

Checking-In Project Items under SCC	C - 936
Checking-Out Project Items under SCC	C - 936
Showing Configuration Management History	C - 937
Showing Differences between Project Versions	C - 937
Showing Source Control Properties	C - 938
Importing from Source Code Control	C - 938
D - Compiler Configuration for SCADE Suite Code Generation	D - 939
Declaring Compilers	D - 940
Configuration File Example	D - 956
INDEX	959

Introduction

This manual guides you along the use of SCADE Suite® tools with basic feature descriptions, user task-oriented instructions, critical information, and tips.

Before starting with SCADE Suite, you can get a basic introduction to the toolset of SCADE Suite Model-Based Design Environment (MBDE) or get a glimpse of the complete user documentation and accesses to Online Help resources.

- [“SCADE Suite Environment Toolset”](#)
- [“Launching/Closing SCADE Suite Environment”](#)
- [“Documentation and Online Help”](#)

The *SCADE Suite User Manual* is divided into parts covering:

- [“Design and Model Editing”](#)
- [“Simulation and Debugging”](#)
- [“Verification”](#)
- [“Code Generation”](#)

For reference about SCADE Suite customization, SCADE file formats, and other back-end data, consult *SCADE Suite Technical Manual* from [“Technical Manual Overview”](#).

SCADE Suite Environment Toolset

Depending on acquired licenses, SCADE Suite environment gives access to:

- [“SCADE Suite Editor”](#)
- [“SCADE Suite Simulation”](#)
- [“SCADE Suite Design Verifier”](#)
- [“SCADE Suite Timing and Stack Optimizer”](#)
- [“SCADE Suite Timing and Stack Verifiers”](#)
- [“SCADE Suite KCG Code Generator”](#)
- [“SCADE Suite Multicore Code Generator”](#)
- [“SCADE Display® Integration”](#)
- [“SCADE Test Integration”](#)
- [“SCADE Suite Simulink® Importer”](#)
- [“SCADE Suite Gateway for NI VeriStand”](#)
- [“Real-Time Operating System Wrappers”](#)
- [“SCADE Configuration Management”](#)
- [“SCADE LifeCycle® Integration”](#)

SCADE Suite Editor

SCADE Suite Editor provides various means to get a clear and comprehensive representation of the project framework based on efficient graphic designing and multi-view management in compliance with model-based methodology:

- Model-based design to capture data and control flows on large projects
- Model check with default or user-defined design/methodological rules
- Computation of default or user-defined metrics about models
- Analysis of semantic differences between models or similar objects
- Dynamic consistency checking and error browsing to find design mistakes
- Predefined operators and library operators for common design functions
- Ability to associate libraries for quick access to frequently-used operators
- Ability to perform a consistent refactoring of models
- Integration of Configuration Management tools
- Use of Requirements Management tools via SCADE LifeCycle ALM Gateway
- Automatic generation of customizable reports via SCADE LifeCycle Reporter

For details and usage instructions, see part about [“Design and Model Editing”](#).

SCADE Suite Simulation

SCADE Suite Simulation supports block-by-block debugging using breakpoints and stop conditions, graphical monitoring of clean data, and the debugging of state machine sub-systems in SCADE Suite models. For details and usage instructions, see part about [“Simulation and Debugging”](#).

SCADE Suite Design Verifier

SCADE Suite Design Verifier¹ is able to formally verify that a property of a SCADE Suite design holds. SCADE Suite Design Verifier comes with predefined properties like absence of division by zero or overflow that can be verified on demand. For details and usage instructions, see Chapter 16 about [“Running Formal Verification Sessions”](#).

SCADE Suite Timing and Stack Optimizer

SCADE Suite Timing and Stack Optimizer² enables to improve the efficiency of SCADE Suite models with respect to WCET and stack usage. This analysis tool compares the worst-case execution time (WCET) and stack size estimation for different versions of a SCADE Suite model to determine the most efficient design. For details and usage instructions, see Chapter 17 about [“Verifying Timing and Stack Usage”](#).

-
1. SCADE Suite Design Verifier is powered by Prover PSL from Prover Technology. Prover, Prover Technology, Prover Plug-in, and the Prover logo are trademarks or registered trademarks of Prover Technology AB in European Union, the United States, China, and in other countries.
 2. Timing and Stack Optimizer is powered by aiT, a product of AbsInt.

SCADE Suite Timing and Stack Verifiers

SCADE Suite Timing Verifier and SCADE Suite Stack Verifier¹ enable to verify the final efficiency of application developed in SCADE Suite for integration on a target platform. These analysis tools compute precise (WCET) and stack size for a model on a specific hardware target. Each aiT target must be installed under AbsInt licenses. They allow comparing results and visualizing them graphically. For details and usage instructions, see Chapter 17 about [“Verifying Timing and Stack Usage”](#).

SCADE Suite KCG Code Generator

SCADE Suite KCG Code Generator generates code for constraining development processes and industries. Certification kits are available for faster and easier code certification. SCADE Suite KCG generates portable and modular C code compliant with ANSI standards. It is OS-independent, as well as customizable to target and project constraints. For details and usage instructions, see part about [“Code Generation”](#).

SCADE Suite Multicore Code Generator

SCADE Suite Multicore Code Generator (MCG) generates task-based code for integration on multicore target platforms. Generated code can be targeted to any multi-, many-core platforms while preserving determinism of the application. SCADE Suite Multicore Code Generator also generates portable and modular C code compliant with ANSI standards. For details and usage instructions, see part about [“Code Generation”](#).

1. Timing and Stack Verifiers are powered by aiT, a product of AbsInt.

SCADE Display® Integration

SCADE Display and SCADE Suite enable the joint model-based design of control logic and graphics display in embedded applications. Their integration provides access to the simulation, life cycle management, and code generation joint capabilities. SCADE Display allows to design complex and interactive display systems with advanced graphical prototyping and specification, animation features, design optimizations, automatic documentation production, configuration management, requirements management and traceability, and optimized C code generation for DO-178C Level A and B, for IEC 61508 at SIL 3, and for EN 50128 at SIL 3/4. For details, see [“SCADE Display Environment Toolset”](#) on page 2 in *SCADE Display User Manual*.

SCADE Suite Simulink® Importer

SCADE Suite Simulink Importer extends SCADE Suite with the ability to capture algorithm design from Mathworks® Simulink® models, Stateflow® charts, and MATLAB® variables. Discrete controllers including Stateflow diagrams and constructs are translated into SCADE Suite models. The tool supports modular import from Simulink models. For details, see *Simulink Importer Guidelines* from [“Guidelines Overview”](#).

SCADE Suite Gateway for NI VeriStand

SCADE Suite Gateway for NI VeriStand extends SCADE Suite with the ability to perform co-simulation of SCADE Suite models in National Instrument VeriStand environment. For details, see *Gateway Guidelines for VeriStand* from [“Gateway Guidelines Overview”](#).

Real-Time Operating System Wrappers

RTOS Adaptors extend SCADE Suite with the ability to integrate SCADE Suite-generated code with Real-Time Environment platforms. Each RTOS Adaptor licensed with SCADE Suite is based on a generic RTOS integration mechanism that can be customized through SCADE Expertise Services for any real-time environment. For details, see *RTOS Adaptor Guidelines* from [“RTOS Adaptor Guidelines Overview”](#).

SCADE Configuration Management

SCADE Configuration Management for SCADE Suite projects supports Configuration Management tools that are compliant with Microsoft SCCI (Source Code Control API) such as ClearCase, CM Synergy, CVS, eChangeMan, Perforce, PVCS, Visual Source Safe, etc. For details and usage instructions, see Appendix C about [“SCADE Suite and Source Code Control”](#) and Appendix B about [“SCADE Suite Files in Configuration Management”](#).

SCADE Test Integration

SCADE Suite model-based design environment is integrated with SCADE Test toolset which provides access to prototyping, testing, and verification capabilities:

- Creation of easy-to-design interactive graphical panels using predefined widgets (buttons, sliders, etc.) with **SCADE Test Rapid Prototyper** to interact with the application under test and facilitate early requirements validation.
- Creation, automation, and execution of tests for coverage analysis with **SCADE Test Environment for Host** including coverage results from SCADE Test Model Coverage.

- Verification of verification activities through coverage measures and analysis with **SCADE Test Model Coverage**. Measuring coverage based on test cases allows to assess the verification activities performed on SCADE Suite models and examine how extensively the behavior of software design was simulated. Model coverage fully integrates into one's own software verification flow.
- Generation of test harnesses from SCADE test procedures for target execution with **SCADE Test Target Execution**.

For details, see [“SCADE Test Toolset”](#) on page 2 in *SCADE Test User Manual*.

SCADE LifeCycle® Integration

SCADE Suite model-based design environment is integrated with SCADE LifeCycle and provides access to application life cycle management capabilities:

- Automatic documentation generation with **SCADE LifeCycle Reporter**
- Requirements management and traceability with **SCADE LifeCycle ALM Gateway**

For details, see [“SCADE LifeCycle Toolset”](#) on page 2 in *SCADE LifeCycle User Manual*.

Launching/Closing SCADE Suite Environment

You can launch and close SCADE Suite model-based design environment as described below. If no valid license is found, SCADE Suite starts in Viewer mode. This mode allows opening any project and model in the Editor, and accessing all features except those implying a Save action (and any other action provoking model files update on the disk).

To launch session on Windows

- 1 Open the **Start** menu and select **SCADE** application.
- 2 In Windows Explorer, double-click a VSW or ETP file.

To close session on Windows

- 1 Select *File > Exit*. A dialog box may pop up in the following cases:
 - Modifications done in Symbol View or Textual View left unapplied.
 - SCADE Suite model or workspace modified.
- 2 In the pop-up window, you choose between the following:
 - Click **Yes** to apply modifications, save the project/workspace, and continue exiting.
 - Click **No** to continue exiting without saving modifications.
 - Click **Cancel** if you want to stop exiting the application.

Documentation and Online Help

The documentation and online help resources corresponding to SCADE Suite are available from various points of access and in various formats. You access them from the Windows button before launching SCADE products or from Startup pages and Help menus after launching SCADE products.

- [“Displaying SCADE Suite Startup Page”](#)
- [“Accessing SCADE Suite Help Resources”](#)
- [“Navigating in Online Help Interface”](#)
- [“Accessing Contextual Help Resources”](#)
- [“Getting Information about Product Version”](#)
- [“Getting Online Support”](#)

Displaying SCADE Suite Startup Page

You can display the view of the startup page at any time. This page provides quick access to basic project start, distributed examples, product documentation, support contacts, or web resources.

To display the Startup page

- 1 Select *View > Startup Page* menu.

Tip

To disable this page at application launch, select *Tools > Options*, go to Environment options, and unselect *Display startup page* in the list of Workspace options.

- 2 Click link to learn more about **SCADE Suite**.

Accessing SCADE Suite Help Resources

You consult help resources and navigate in the content via online help topics or the PDF documentation. To access help contextually from SCADE Suite user interface, see [“Accessing Contextual Help Resources”](#) on page 12.

To access SCADE Suite online help resources

- 1 Select *Help > Help Topics* or press F1 to open SCADE Help Resources.
- 2 Display **Contents** panel and double click the **SCADE Suite Resources** topic to expand its content, or click the **SCADE Suite Help** link on the **Welcome** page to display related resources.
- 3 Browse for topics (see [“Navigating in Online Help Interface”](#)).

To consult SCADE Suite documentation in PDF

- 1 Select *Help > PDF Documentation > Ansys SCADE Suite Manuals*. Click on links to access SCADE Suite documentation.

Tip

Select *Help > PDF Documentation > Ansys SCADE Release Notes* to open Release Notes about SCADE products and access to SCADE Suite Release Notes.

- 2 Search for information in the opened document. You can consult all distributed manuals and documents in PDF format: [Click here](#).

To access documentation from Windows desktop

- 1 From Windows Start button, select *Ansys SCADE* installation folder:
 - Click **Ansys SCADE Documentation** to navigate to the different documentation packages and look for documents.
 - Click **Ansys SCADE Release Notes** to open a top-level Release Notes and navigate to the different Release Notes.

Navigating in Online Help Interface

In online help interface, you can look for topics in several ways: **Contents** bookmarks, **Index** list, **Search** features, embedded navigation features like related topic links, drop-down sections, or hyperlinks in the content.

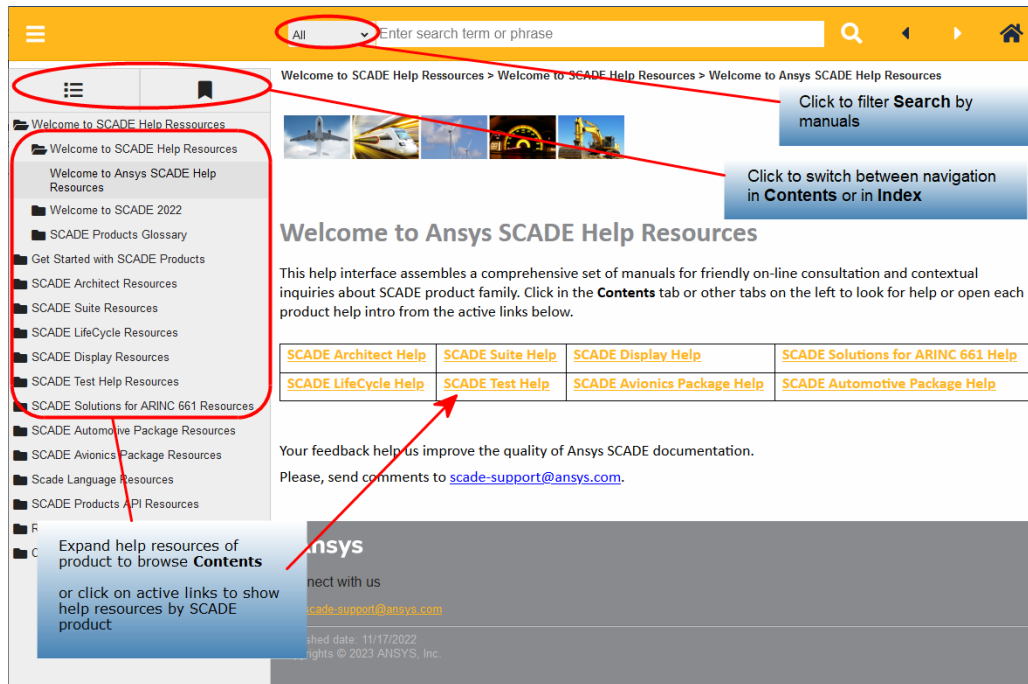


Figure 1: Online help interface general display and SCADE online help resources

The online help interface displays with the following features:

- *Navigation panel* on the left for bookmark navigation in content or index search
- *Search bar* on top with capability to limit search by manuals
- *Controls* on top for hiding left panel ☰, for navigating between **Previous** ◀ and **Next** ▶ pages, or navigating back to **Home** 🏠
- Navigate back to parent pages via breadcrumbs (top of pages)

Accessing Contextual Help Resources

You can consult help contextually about menus and interface items.

To display contextual help topics about menus

- 1 Select the menu from the main menu bar (or any contextual menu).
- 2 Press F1 to open the corresponding help topic.

To display contextual help topics about Scade operators

- Click **Help**  in **Standard** toolbar and click again on Operator buttons in toolbars or symbols of predefined operators in diagrams to get help.

To display contextual help topics about interface items

- Click **Help**  in **Standard** toolbar and click again on toolbar buttons, Docking windows, or Browser areas to get help.

To display tooltip help on Settings dialogs

- Click **?** in upper-right corner and click again on option item in the current dialog. A tooltip opens showing a short help text (if any). Clicking a tab opens the help topic like the **Help** button in the lower-right corner.

Getting Information about Product Version

To get information about installed software version

- Select **Help**  in **Standard** toolbar (or *Help > About*) to display the version.

Getting Online Support

To access online support web site and contact support team

- 1 Select *Help > Ansys SCADE Support > Support web site* to access Ansys Customer Support and Help Desk.

Part 1

Design and Model Editing

- Chapter 1: [“Fundamentals about SCADE Suite Editor”](#)
- Chapter 2: [“Setting Project and Design Environment”](#)
- Chapter 3: [“Designing Models”](#)
- Chapter 4: [“Designing with Predefined Operators and Constructs”](#)
- Chapter 5: [“Navigating in Models”](#)
- Chapter 6: [“Editing Properties of Design Elements and Objects”](#)
- Chapter 7: [“Computing Metrics and Checking Rules on Model”](#)
- Chapter 8: [“Change Control of Models”](#)
- Chapter 9: [“Managing Model Refactoring”](#)
- Chapter 10: [“Running Coverability Analysis on Design”](#)



1 / Fundamentals about SCADE Suite Editor

This chapter contains generic information about **SCADE Suite Editor** usage in the context of model-based design development of software applications.

- [“Model-Based Design Principles”](#)
- [“Exploring SCADE Suite Editor User Interface”](#)
- [“Model Development Workflow”](#)
- [“Standard File and Project Management”](#)
- [“SCADE Suite Files”](#)
- [“Multifile Management”](#)

Model-Based Design Principles

SCADE Suite Editor empowers application designers with the ability to model graphically, as well as textually, their system architectures and data based on the Scade language formalism that integrates both control and data flows. In combination with dedicated tools, SCADE Suite design environment offers a large degree of openness and flexibility for project management, as well as for system requirements capture and management: to manage traceability between design and textual requirements; to edit models imported from other development environments; to set models for configuration management; to compare model designs; or to generate and configure model reports.

- [“Design Architecture Modeling: Operators”](#)
- [“Control Formalism: State Machines”](#)
- [“Data Structuring: Types and Constants”](#)
- [“Requirements Management”](#)

Design Architecture Modeling: Operators

SCADE Suite makes it possible to design the architecture of models graphically with the help of operators. Operators allow to compose the functional parts identified in the high-level requirements (HLRs). SCADE supports the following types of operators:

- **Nodes:** operators with memory and defined in graphical or textual diagrams.
- **Functions:** operators without memory and defined in graphical or textual diagrams.
- **Imported nodes/functions:** operators defined with external C code to extend Scade language expressiveness.
- **Predefined operators:** built-in operators that provide access to basic primitives.
- **Library operators:** operators that provide access to all sorts of computation operations that are usable in the design architecture, and that support model translation in SCADE from external development tools. Library operators are contained in SCADE standard libraries proposed at project creation.

Note

Any operator you create in SCADE Suite models is referred to as a user-defined operator to distinguish them from the predefined operators distributed in SCADE. Notice also that user-defined operators can be used in SCADE Suite models as library operators contained in user-defined libraries.

All types of SCADE operators except built-in predefined operators can be defined with the support of graphical and textual diagrams. An operator is created with a default graphical diagram. Any number of graphical diagrams can be associated with an operator. Textual diagrams allow to complete manually the SCADE Suite design architecture with textual Scade definitions.

Data Structuring: Types and Constants

SCADE Suite provides various ways to facilitate the structuring of data flows in models. The Scade language is strongly typed. All variables are explicitly typed and users can define as many types as necessary. SCADE supports the following language constructions:

- Predefined data types (bool, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, float64, and char)
- Structured types (data structure and array)
- Imported types: types defined in C or Ada code.
- User-defined types
- Constants: constants named and typed by users.
- Imported constants: constants defined in external C code.
- Constant literals or expressions
- Structured type operators with static indexing
- Array operators with dynamic indexing

Control Formalism: State Machines

State machines are used by the community of system architects and software designers. The graphical formalism of State Machines in SCADE Suite allows to model the behavior of reactive systems with their control logic environment. Applications usually contain control-driven and data-driven parts. Generally, the control flow drives the data flow, but some functions of the control-driven parts can be data-driven subsystems.

In any case, State Machines can be easily and consistently mixed with data-flow design. When modeling systems with control logic, designers may want to express complex discrete control. State Machines introduce an enriched formalism that handles complex control structures and extends the notation

with strong and unique synchronous semantics able to enforce model determinism and achieve efficient formal proofs. The work of application designers is empowered by the following capabilities:

- *hierarchy*: hierarchical decomposition of State Machines for designing the system architecture
- *parallelism*: parallel composition of several State Machines for modeling concurrent behaviors of the system
- *preemption*: ability to interrupt or allow state transitions for handling exceptional situations in the system

State Machines in a given state react to various signals through transitions. These transitions are triggered depending on some conditions and activate other states. Priorities on transitions enforce model determinism.

- [“State Machine and States”](#)
- [“State Machine Transitions”](#)
- [“State Machines with Signals on Transitions”](#)
- [“Parallel Composition of State Machines”](#)
- [“Hierarchical Decomposition of State Machines”](#)
- [“Broadcast Communication”](#)
- [“Mix of Data Flow and control Flow”](#)

STATE MACHINE AND STATES

States are the basic memory element of State Machines. Each State Machine must have a unique initial state and can contain one or more states. States do not model any behavior in the system; they are idle unless you associate them with an explicit behavior. At each cycle, a state is either active or inactive. No output is produced as a result of the system being in a given state. To trigger transitions and produce outputs, the behavior must be defined explicitly inside a state using State Machines modeling constructs introduced below.

STATE MACHINE TRANSITIONS

Transitions are State Machine components that connect a source state to a target state. Whenever the source state is active, the transition preempts the activity of the source state and starts the activity of the target state. The following types of transitions are available for modeling application behaviors:

- *Strong preemption*: this transition deactivates the source state and activates the target state to start its internal activity.
- *Weak preemption*: this transition allows the source state to finish its activity and delays the start of the target state activity till next cycle.
- *Synchro preemption*: this transition is triggered when a state terminates (i.e., all its sub-State Machines are in terminal state at the end of the cycle) and starts the activity of the target state on next cycle.

Transitions can be defined with trigger conditions and actions. A trigger condition is any Scade Boolean expression that contains, if needed, a signal presence test. The transition can be triggered once the condition is true. An action is a list of signal emissions and/or variable definitions produced when the transition is triggered; it is the effect of an activated transition.

A transition can be triggered if all the conditions on the path to the target state are true. When a transition is triggered, the action is performed (if any), the source state is preempted, and the target state of the transition becomes the active state. In the simplest case, the transition is triggered when the condition of the transition between two states is verified. For instance, A or B is a valid condition expression between two states. If A, B, or both are present in the input of the current transition, the transition is triggered (depending on other possible preempting transitions). In more complex cases like forked transitions (see definition below), the transition can be triggered only once all conditions required to reach a given state are verified. As illustrated in [Figure 1.1](#), an action is not performed as long as the condition

evaluation remains partial. Considering C and C1 as conditions to **StateB**, C and C2 as conditions to **StateC**: the action that emits signal A is defined on condition C and is performed only when C and C1 are true, or C and C2 are true.

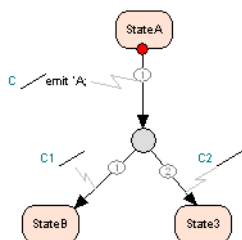


Figure 1.1: Action execution and condition evaluation on fork transitions

Fork transitions allow to represent decisions branches based on conditions and priorities. *Resuming transitions* (i.e., transitions with History connector) allow to resume the execution of a state as it was when last exited and executed. Without resuming transition (*restart transition*), a state is restarted (i.e., reset when activated). You can attach any number of forked or resuming transitions to the incoming transitions of a state. Multiple fork transitions can be managed by setting higher and lower priorities. For more details, consult [“Forks”](#) on page 65 in *Scade Language Primer*.

STATE MACHINES WITH SIGNALS ON TRANSITIONS

Signals are essentially used to describe the interactions of a control flow model with its environment. In State Machines, the communication model is based on the presence of signals: if a signal is emitted its presence in the synchronous cycle can be tested. Testing the presence of a signal means checking whether the signal was emitted or not. If not emitted, the presence is false; if emitted at least once, the presence is true.

Emitted signals are visible within the scope of the operator or state they are defined in. It is always possible to emit the same signal several times in a given cycle. Once emitted in a cycle, a signal is present in that cycle. Unless explicitly set as present again, the signal is considered to be absent in the next cycle. For more details, consult [“Pure Signals”](#) on page 55 in *SCADE Language Primer*.

PARALLEL COMPOSITION OF STATE MACHINES

You can use State Machines in parallel to describe application behaviors that run concurrently as illustrated in [Figure 1.2](#). A parallel system is the conjunction of several State Machines. At a given cycle, all parallel State Machines are activated in their respective initial states. The termination of parallel State Machines is effective only when all parallel State Machines reach their terminal state. For more details, consult [“Parallel Composition of State Machines”](#) on page 59 in *SCADE Language Primer*.

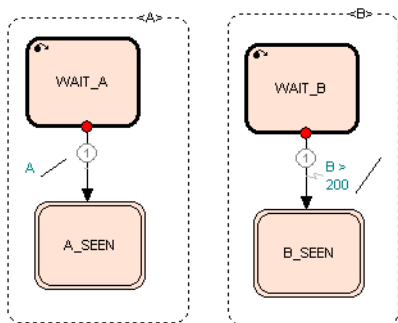


Figure 1.2: Complex State Machine with parallel composition

HIERARCHICAL DECOMPOSITION OF STATE MACHINES

You can design State Machines that contain other State Machines and states to describe the application behavior within hierarchical subsystems as illustrated in [Figure 1.3](#). You can define State Machines with unlimited hierarchical levels to refine complex behaviors.

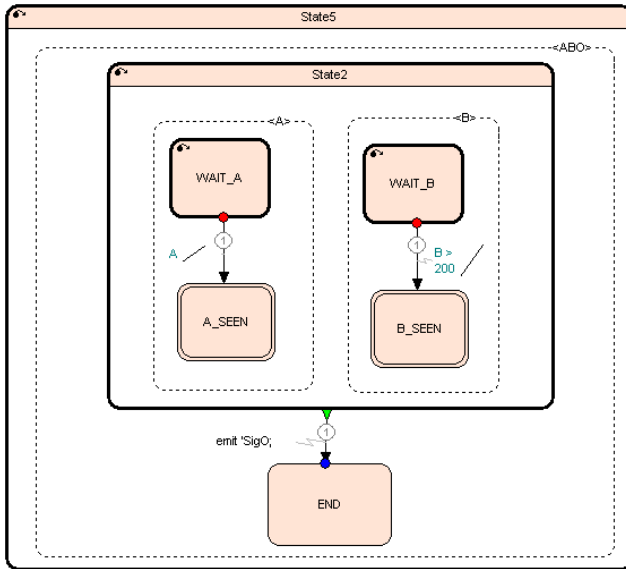


Figure 1.3: Complex State Machine with hierarchical decomposition

BROADCAST COMMUNICATION

Signals emitted by one parallel component can be tested by another component, thus triggering changes in other components, and so on. Signal emission becomes necessary to synchronize and trigger dependent states running in parallel systems. Because a read operation does not reset the presence of the emitted signal, any number of listening State Machine can read an emitted signal simultaneously. This is called broadcast communication: like radio communication there can be one emitter and any number of listeners.

MIX OF DATA FLOW AND CONTROL FLOW

You can design State Machines with several data-flow definitions embedded into states. Such data-flow algorithms get computed only when the state are activated. State transitions can also carry data-flow definitions which are computed directly when transitions are triggered. For clear and efficient

designs, you must pay attention to any undefined flows. Each local or output flow declared must have exactly one definition for each cycle where its scope is active. Whenever a data flow definition is missing in a given state, you must ensure that you either maintain the latest value and set its initial value, or define a default value. In case a data flow component is defined with both Last and Default values, the Default value has the highest priority. For more details, consult [“Flow Definitions on Transition”](#) on page 66 in *SCADE Language Primer*.

Requirements Management

SCADE Suite is able to link model design to High Level Requirements (HLRs) of the system application you develop. SCADE LifeCycle ALM Gateway enables the capture of HLRs automatically from a database or from any textual documents. Captured HLRs can be associated with SCADE Suite objects to facilitate the interactive analysis and the tracking of requirements coverage. For more details about requirements capture, consult *SCADE LifeCycle User Manual*, Part 2 about [“Requirements and Traceability”](#).

Model Coverability Analysis

Because coverage analysis (or coverage resolution) is a complex and time-consuming activity, it is possible to run coverability analysis sessions during design phase. Designers can analyze the coverability of all coverage points in SCADE Suite models and report a diagnostic about coverage points (coverable, uncoverable, or undecided) before starting any testing phase with SCADE Test toolset. For details about coverability analysis at design phase, see Chapter 10 about [“Running Coverability Analysis on Design”](#).

Exploring SCADE Suite Editor User Interface

SCADE Suite is a Model-Based Design Environment for designing critical softwares based on the native integration of the Scade language and its formal notation. The user interface allows to manage project content, to edit definitions, and to represent software design in dedicated views.

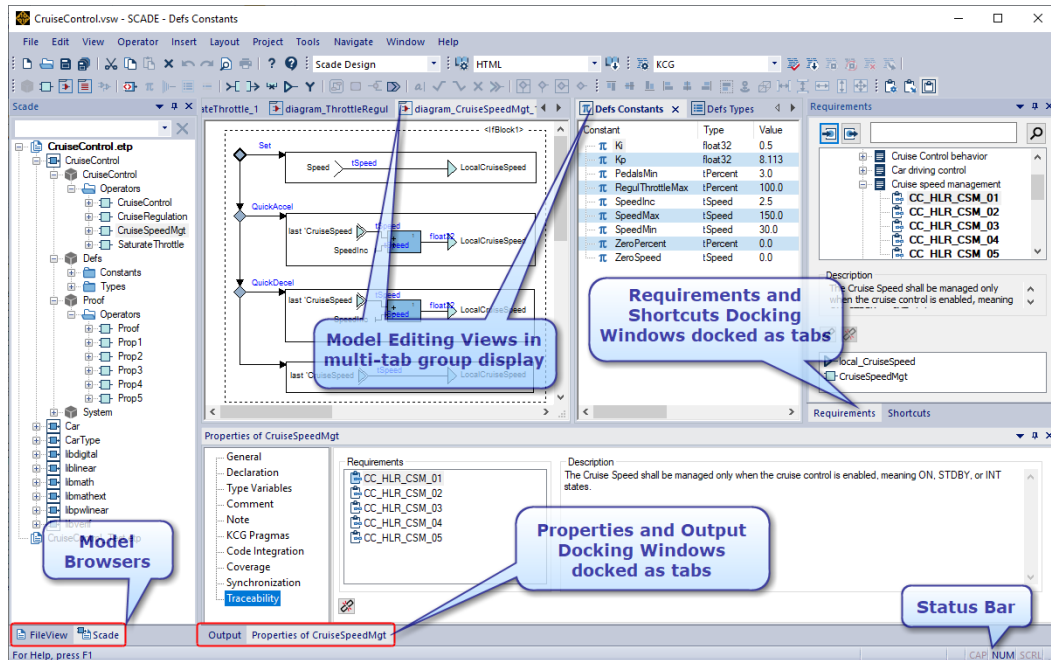


Figure 1.4: Multi-windowing display of Windows and Views in SCADE Suite Editor

If no valid license is found, SCADE Suite starts in Viewer mode. This mode allows to open any project and model, and access to all features that do not imply a Save action or any action requiring model files update on the disk.

The main interface components are introduced below:

- [“Perspectives”](#)
- [“Docking Windows”](#)
- [“Model Editing Views”](#)
- [“Toolbars”](#)
- [“Symbol Editor”](#)
- [“Status Bar Display”](#)

Perspectives

Perspectives store the configuration of SCADE Suite display and organization whether in full screen mode or not. They allow to control the display and position of the menu bar and toolbars, as well as the display, position, and size of docking windows. Switching perspectives is a quick way to adapt the application display to your context of use. You may use predefined or custom perspectives as detailed in [“Managing Perspectives”](#) on page 92.

Docking Windows

SCADE Suite Editor provides access to various levels of model information in docking windows which contain views displayable for different user purposes:

- [“Model Browsers”](#)
- [“Requirements Window”](#)
- [“Shortcuts Window”](#)
- [“Tool-Dedicated Browsers”](#)
- [“Output Window”](#)
- [“Diff View Window”](#)
- [“Properties Window”](#)

To show/hide Docking Windows and Model Browsers

- 1 In SCADE Suite, select *View > Docking Windows* or *View > Model Browsers*.
- 2 Select a window or browser from the following list:

Output	Properties	Scade
Shortcuts	Diff View	Design Verifier
Requirements	File View	Test View

SCADE Suite windows are all dockable by double click. To undock windows, double click on the title bar. The usage and layout of docking windows in SCADE Suite is described in [“Organizing and Working with Docking Windows”](#) on page 154. The docking window layout is saved by [Perspectives](#). Other views obtained via API-related Tcl commands can be used (see details in Chapter 4 about [“Scripting Capabilities in SCADE Environments”](#) in *SCADE Suite Technical Manual*).

Model Browsers

The basic browser views allow to manage project files, declare and define design elements used in models, and examine dependencies between model operators. These views can be displayed as docked or tabbed windows.

Note

It is possible to extend SCADE Suite interface with custom browsers. For details, see [“GUI Display-Related Tcl Commands”](#) on page 76 in *SCADE Java/Tcl API Guide* or [“GUI Display-Related Python Commands”](#) on page 99 in *SCADE Python API Guide*.

The default standard installation includes the following browsers:

Table 1.1: Basic model browsers

Browsers	Description
“File View”	Tree view displaying all SCADE Suite model files and library files.
“Scade View”	Tree view displaying SCADE Suite design elements defined in the model.
“Call Graph”	Tree view displaying the dependence tree of model operators.

The content of any browser view can be expanded or collapsed clicking the standard + or - symbols in front of any item. You can also open contents using the keyboard arrow keys to expand/collapse and move up/down.

To expand/collapse selected items in tree views

- Select any tree item and press the + key on the numeric pad of the keyboard. Expansion applies to selection.
- Select any tree item and press the - key on the numeric pad of the keyboard. Collapse applies to selection.

To expand/collapse all sub-items in tree views

- Select any tree item and press the * key on the numeric pad of the keyboard. Expansion applies to all sub-items recursively.
- Select any tree item and press the / key on the numeric pad of the keyboard. Collapse applies to all sub-items recursively.

FILE VIEW

The **File View** includes a workspace file (.vsw), one or several projects (.etp) grouped in the same workspace, as well as all files necessary to the project and its design. The SCADE Suite model files include XSCADE files that may be completed by external definition files (C-file, H-files, or Ada files). For details about files, see [“SCADE Suite Files”](#) on page 60.

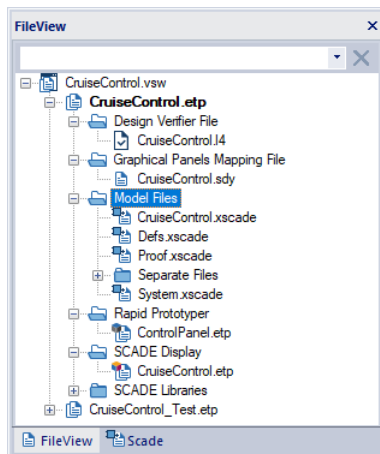


Figure 1.5: File tree and folder organization in File View

Whenever a model uses a library, the library appears as a user library file included in the model, but its content (*i.e.*, references to any SCADE Suite model files) cannot be accessed. You can display all XSCADE model files of the

project, but not the ones referenced by any project inserted as a library. To do so, you need to open the ETP file of the library in another SCADE Suite application session.

The **File View** allows to manage SCADE Suite project files as follows:

- Displaying the list of all project files, searching for files, checking file information
- Editing the project(s) by inserting, moving, or removing files
- Adding new virtual folders to organize project files
- Specifying project properties at project file level

The **File View** supports the creation of virtual folders to organize all project files. You can add/remove folders if needed.

From any project file (etp) or file referenced in model browsers, it is possible to open the directory that contains them in an Explorer window.

To display file directory containing project resources

- Right-click any project or file in browsers and select *Open containing folder*.

The folders provided at project creation are as follows:

Table 1.2: List of virtual folders created in File View

Folder	Type of Files	Description
Default folders at project creation:		
Design Verifier File	.l4	References the file that enables Design Verifier functions
Model Files	.xscade	References the files corresponding to the packages you create in the current project
SCADE Libraries	.etp	References the library files inserted into the current project
Folders created automatically during some model editing operations:		
External Code	.c .ada .adb	References the files that define design elements like imported operators declared with external code
External Type Definitions	.h	References the files that define design elements like imported types

RELATED TOPICS

- [“Inserting Existing Project or Files”](#) on page 112
- [“Modifying Definitions from File View”](#) on page 281
- [“Creating New Folders in Projects”](#) on page 114
- [“Editing Project Properties”](#) on page 424
- [“Editing Project Files Properties”](#) on page 431
- [“Filtering Model Content”](#) on page 421
- [“Organizing and Working with Docking Windows”](#) on page 154

SCADE VIEW

The **Scade View** contains the definition of all design elements used in models. For each project included in the same workspace, the design elements can be sorted and grouped by packages. SCADE Suite organizes new design elements in virtual folders inside or outside packages. The virtual folders containing design elements are sub-folders of package folders.

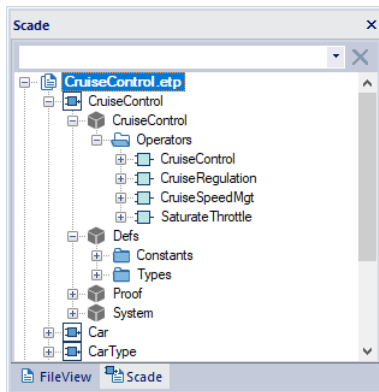


Figure 1.6: Design elements tree and folder organization in Scade View

The project above shows a CruiseControl package containing some definitions of the model in the Constants, Types, and Operators folders. In each folder, elements appear in alphabetical order except in operator interfaces where user-defined order is kept. Private declarations display a small key on top of package, operator, constant, or type icons.

The **Scade View** allows to visualize and define design elements in the model as follows:

- Displaying and searching model elements
- Displaying model libraries: since model libraries are read-only, it is necessary to open them as a main project to be able to edit them
- Creating new design elements, renaming elements, or inserting existing items

- Editing properties and notes for all design elements and objects
- Moving/removing any design element with cut/paste or drag and drop inside **Scade View**
- Opening design elements with a double click in the corresponding view
- Dragging and dropping declared design elements in diagrams

To collapse project packages and libraries at once

- Right-click in **Scade View** and select *Collapse all tree*.

RELATED TOPICS

- [“Declaring Design Elements in Model”](#) on page 160
- [“Inserting Existing Item”](#) on page 206
- [“Moving Design Elements in Models”](#) on page 279
- [“Finding and Locating Design Elements”](#) on page 409
- [“Setting Design Instances in Diagram Views”](#) on page 232
- [“Filtering Model Content”](#) on page 421
- [“Organizing and Working with Docking Windows”](#) on page 154

CALL GRAPH

The **Call Graph** displays the dependency tree of all model operators. Diagrams are displayed in this view when several diagrams belong to the same operator.

You can customize this view since it is based on a Tcl script. To create other custom views, consult Chapter 4 about [“Scripting Capabilities in SCADE Environments”](#) in *SCADE Suite Technical Manual*.

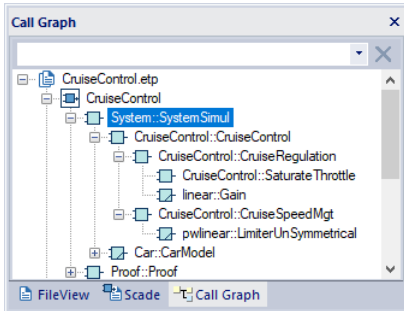


Figure 1.7: Model operator tree in Call Graph

Like any customizable view, the **Call Graph** does not refresh automatically. If you change the model, select *View > Call Graph* to refresh its content.

You can expand/collapse the content of the view by clicking the standard + or - symbols in front of any item. You can also navigate in the tree using the arrow keys of the keyboard.

Tip

Double-click an operator in the Call Graph to open it in diagram.

RELATED TOPICS

- [“Filtering Model Content”](#) on page 421
- [“Organizing and Working with Docking Windows”](#) on page 154

Tool-Dedicated Browsers

Such browser views allow to display information related exclusively to some SCADE Suite tools. The default standard installation includes the following browsers:

Table 1.3: Tool-dedicated browsers

Browsers	Description
Instances	Displays observable model instances when running simulation session (see “Instances Model Browser” on page 575 in “Simulation and Debugging” part)
Design Verifier	Displays the verification tools defined for verifying model properties (see “Design Verifier View” on page 723 in “Verification” part)
Model Coverage View	Displays coverage results produced during coverage sessions (see “Model Coverage View” on page 36 in <i>SCADE Test User Manual</i>)

Properties Window

This docking window displays the properties associated with any design element selected in models and allows to edit them as required. Editing properties can be performed on one element, on several elements in a row,

or on multiple elements at the same time. The properties also enable to associate comments and/or annotations with any design elements, and even propagate such information in the generated code.

Tips

While specifying properties, you can switch between elements from the Properties window without losing window focus using the `Shift+F8` or `F8` keys. You switch only between elements listed on the same tree level.

- To validate option changes in Properties window, press `Enter`, or select another category of properties, or select another element.
 - To close the Properties window and ignore property changes, press `Esc`.
 - To specify and validate the properties of several objects, it is not necessary to close the Properties window. Changing selection automatically validates property changes.
 - To specify the same property on several objects at a time, use multi-selection and enter the value of the selected property.
-

Requirements Window

This docking window is available for managing requirements and traceability at SCADE Suite model level using SCADE LifeCycle ALM Gateway. The purpose and content of this window is described in [“Model-Level Requirements and Traceability”](#) on page 84 in *SCADE LifeCycle User Manual*.

Output Window

This docking window is the message area for all SCADE Suite modules and tools. Whenever using external tools, the window may contain tabs for messages related to such tools. It may also contain custom tabs. For details, consult Chapter 4 about [“Scripting Capabilities in SCADE Environments”](#) in *SCADE Suite Technical Manual*.

The default standard installation includes the following tabs:

Table 1.4: Tabs available from Output Window

Tabs	Description
Messages	General command execution messages
Dump	Textual representation of any element selected in SCADE Suite views
Build	Messages from build generation (“Display of Build and Log Information” on page 832)
Simulator	Messages returned during simulation sessions
Script	Messages related to results from Tcl requests
Info Log	Messages from model checking, simulation code building, or code generation
Browse	Results from dependency search in models (“Searching for Design Dependencies” on page 412)
Find	Results from search in the active project (“Finding and Locating Design Elements” on page 409)
Reporter	Messages from SCADE LifeCycle Reporter
DiffAnalyzer	Messages from DiffAnalyzer (“Analyzing Model and Object Differences” on page 529)

To clear output tab contents on Output Window

- Right-click in the output tab from which to remove content and select *Clear*. Not applicable from Info Log, Browse, Find, or Diff output tabs.

Shortcuts Window

This docking window allows to customize the environment with quick access to toolbar commands or operators (not displayed by default). It may contain:

- Icons representing favorite toolbar buttons grouped in folders named as available [toolbars](#). Grayed icons for predefined operators are unavailable functions or deactivated constructs (see [“Selecting Language Construct Options”](#) on page 83).

- Icons representing favorite operators from either a model or a library.
- It is possible to drag and drop predefined operators from the **Shortcuts Window** into diagrams. For details, see [“Managing Favorites and Using Shortcuts”](#) on page 131.

Diff View Window

This docking window is available when comparing differences between two SCADE Suite models or design objects of similar nature using the Diff Analyzer. The purpose and content of this window is described in [“Diff View”](#) on page 527. For details about model comparison, see Chapter 8 about [“Change Control of Models”](#).

Model Editing Views

The model editing views help you display and edit the models for different purposes with different design elements. SCADE Suite maintains model consistency and ensures that user changes made in the model propagate to all editing views. The default standard installation includes the following model editing views:

- [“Constants Views”](#)
- [“Sensors Views”](#)
- [“Types Views”](#)
- [“Textual Views”](#)
- [“Diagram Views”](#)

Tips

- Edit Constants Views, Types Views, and Sensors Views using F2 to modify definitions, the Tab key or the keypad arrow-keys to switch editing focus between columns, and Enter to validate any text entered in definition columns.
 - Focus over the tab of these views to display context information in tooltip.
-

All editing views are accessible from the **Scade View**. Their usage and layout is described in [“Organizing and Working with Model Editing Views”](#) on page 144.

To open model editing views by double click

- 1 In current project, display **Scade View** and expand the Package folders.
- 2 Double click on the following icons to open the corresponding editing view:

Constants		to open the corresponding Constants View
Sensors		to open the corresponding Sensors View
Types		to open the corresponding Types View
Diagrams		to open the corresponding graphical design
Operators		to open the design of the operator (if single diagram defines it)
State Machines		to open the design of the State Machine
States		to open the design of the State
Textual definition		to open the corresponding textual design

You can also open the diagram of an operator by double clicking its graphical instance in diagrams, or by selecting the operator and pressing ENTER on the keyboard. When double clicking an operator or state designed with several graphical or textual diagrams, select the diagram to open from the displayed list of diagrams. Graphical and textual diagrams are distinguished by their icon and name. For details about multiple diagram declaration, see [“Inserting Graphical Diagrams”](#) on page 204 and [“Inserting Textual Diagrams”](#) on page 260.

Constants Views

A **Constants View** displays the declaration of constants. In columns, you declare, for each constant, its name, type, and value(s). You may also add any descriptive comment. Constants are listed in alphabetic order.

Fighter Constants			
Constant	Type	Value	Comments
! IfScope	float32	15.	If scope, in nautic miles
! RdrNarrowAngle	float32	80.0	Radar angle in Narrow mode, in degrees
! RdrNarrowScope	float32	25.	Radar scope in Narrow mode, in nautic miles
! RdrWideScope	float32	10.	Radar scope in Wide mode, in nautic miles

Figure 1.8: Display of a Constants View

When constant values are not consistent with the type, they are displayed with <unknown> type in the View and highlighted by a red checkmark.

The **Constants View** allows for:

- Displaying definition of constants
- [“Setting Imported Constants with External Code”](#)
- [“Declaring New Constants”](#)
- [“Setting Imported Constants with Imported Types”](#)
- [“Setting Array/Structure Type Constants”](#)

Sensors Views

A **Sensors View** displays the definition of sensors. In columns, you declare, for each sensor, its name and type. You may also add any descriptive comment. Sensors are listed in alphabetic order.

Fighter Sensors		
Sensor	Type	Comments
BtnRandomEnv	bool	
BtnResetEnv	bool	
Fighter_!IfBtnMouseIn	bool	
Fighter_LockBtnMouseIn	bool	
Fighter_MousePressed	bool	

Figure 1.9: Display of a Sensors View

The **Sensors View** allows for:

- Displaying the definition of sensors
- [“Declaring New Sensors”](#)

Types Views

A **Types View** displays the declaration of named data types. In columns, you declare, for each type, its name, and set its definition using predefined types or user-defined types. You may also add any descriptive comment. Types are listed in alphabetical order.

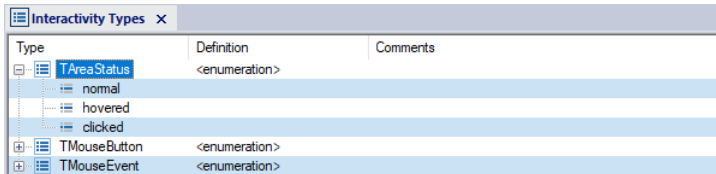


Figure 1.10: Display of a Types View

The **Types View** allows for:

- Displaying definition of types
- [“Declaring New Types”](#)
- [“Setting Array Types”](#)
- [“Setting Enumeration Types”](#)
- [“Setting Structure Types”](#)
- [“Setting Imported Types”](#)

Textual Views

A **Textual View** enables to specify and edit the design of an operator in Scade textual syntax. When double clicking **Text Diagrams** in the Scade **View**, you open Textual Views.

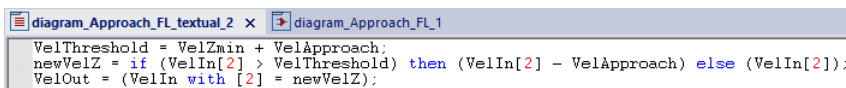


Figure 1.11: Textual View display of an operator specified in Scade syntax

The **Textual View** allows for:

- Displaying content of textual definitions
- [“Applying Changes in Textual Diagrams”](#)
- [“Inserting Textual Diagrams”](#)
- [“Navigating from Text Diagrams to Other Views”](#)
- [“Editing Textual Diagrams”](#)

Diagram Views

A **Diagram View** displays the graphical representation of an operator. Basically, it contains all calls and references to the design element definitions necessary to process control and data flows with respect to the Scade language semantics. Several diagrams can be associated with the same operator. When passing over variables in diagrams, a tooltip displays the variable type and name.

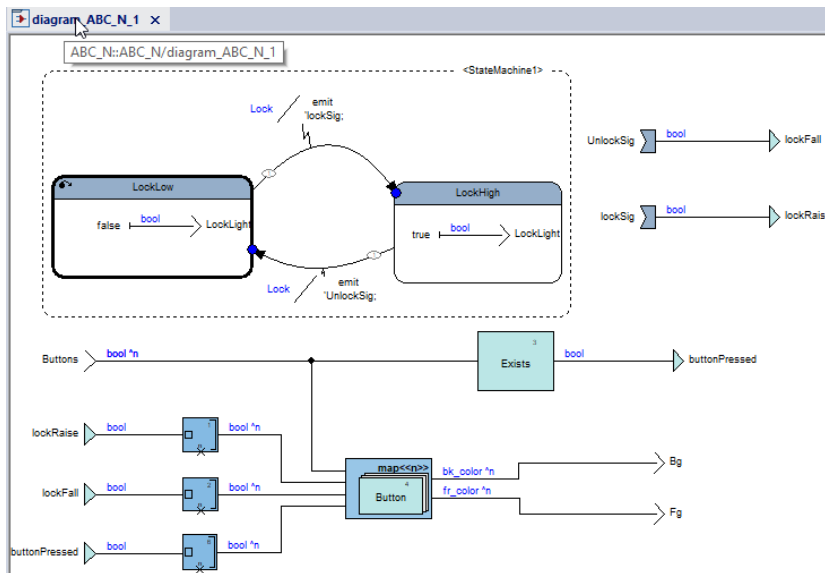


Figure 1.12: Display of a Diagram View

The **Diagram View** allows for:

- [“Creating Operator Calls”](#)
- [“Composing/Decomposing Structured Types”](#)
- [“Substituting Operator and Variable Calls”](#)
- [“Creating References to Inputs/Outputs”](#)
- [“Creating Textual Expressions”](#)
- [“Creating Contracts: Assume/Guarantee”](#)
- [“Creating Terminators”](#)
- [“Creating References to Local Data”](#)
- [“Connecting Data Flow Elements”](#)
- [“Adding/Removing Predefined Operator Inputs”](#)
- [“Defining Value of Hidden Inputs on Operators”](#)
- [“Moving Design Elements in Models”](#)

Toolbars

The toolbars provide shortcuts to some of the most common application commands used for project administration, model design and editing, and navigation in model-based data. Some toolbars are specific to the tools and modules integrated to SCADE Suite Editor. All toolbars are dockable and undockable with a double click on their handle bar. They can be docked to the top, the bottom, the left, or the right of the user interface. They can be moved in the interface by dragging their title bar to the desired location. The toolbar layout is saved by [Perspectives](#). Some toolbars can even be displayed in the [Shortcuts Docking Window](#). The following toolbars are available:

- [“Generic Toolbars”](#)
- [“Design Toolbars”](#)
- [“Model Navigation Toolbars”](#)
- [“Tool-Specific Toolbars”](#)

To show/hide toolbars

- Select *View > Toolbars* and select the name of the toolbar; or right-click in the toolbar area and select the name of the toolbar to show/hide from the interface. Alternatively, hide any toolbar by right-clicking on toolbar and selecting *Hide*.

Generic Toolbars

These toolbars provide shortcuts to the most common commands used for project and file administration, workspace display, and quick tool launch.

Standard	Standard application commands: • “Creating Blank Workspace” on page 95 • “Creating New Project” on page 96 • “Creating New Tcl Files” on page 101 • “Opening/Closing Files” on page 53 • “Opening/Closing Workspace Files” on page 54 • “Saving Files and Workspace Files” on page 56 • “Canceling/Restoring Previous Actions” on page 59 • “Printing Active Views” on page 153 • Displaying Properties window • “Accessing Contextual Help Resources” on page 12 • “Getting Information about Product Version” on page 12
Project	Project management and tool commands: • “Switching Active Project in Same Workspace” on page 102 • “Launching Semantic Checks” on page 275 • Launching simulation sessions, generating and compiling code • Interrupting code generation or simulation • “Launching Property Analyses” on page 756 (Design Verifier) • “Generating Project Documentation” on page 40 (Reporter) • “Launching Testing Sessions on Host” on page 70 in <i>SCADE Test User Manual</i> • “Generating Test Harnesses” on page 203 in <i>SCADE Test User Manual</i> • “Launching Timing or Stack Analysis” on page 809 (TSV/TSO) • “Launching Model Comparison” on page 529 (Model Diff) • “Generating Diff Reports” on page 536 (Model Diff)
Perspective	Adapting content of SCADE Suite workspace: • “Switching Perspectives” on page 93

Design Toolbars

These toolbars provide shortcuts to some of the most common commands used for model design and editing.

Create	Organizing model design and defining data and control flow: • “Creating New Packages” on page 125 • “Inserting Graphical Diagrams” on page 204 • “Inserting Textual Diagrams” on page 260 • “Defining Constants” on page 218 • “Defining Sensors” on page 230 • “Defining Data Types” on page 207 • “Setting Operator Parameterized by Size” on page 257 • “Declaring Model Operators” on page 160 • “Declaring Operator Interface” on page 162 • “Declaring Operator Local Data” on page 164 • “Creating Contracts: Assume/Guarantee” on page 242 • “Creating Terminators” on page 243 • “Creating and Designing State Machines” on page 168 • “Creating and Designing If Blocks” on page 180 • “Creating and Designing When Blocks” on page 188
State Machine	Editing state machine transitions and states: • “Creating and Designing State Machines” on page 168
Predefined operator toolbars:	
Mathematical	Adding mathematical operations: • “Designing with Arithmetic Operators” on page 316
Bitwise	Adding bitwise mathematical operations: • “Designing with Bitwise Arithmetic Operators” on page 324
Comparison	Adding comparison operations: • “Designing with Comparison Operators” on page 336
Logical	Adding logical operations: • “Designing with Logical Operators” on page 343
Structure/Array	Adding structured or array operators: • “Designing with Structured Values” on page 362 • “Designing with Array Values” on page 367

Higher Order	Adding iterators and activate/restart operators: • “Designing with Array Iterators” on page 378 • “Designing with Operator Activation” on page 402 • “Designing with Restart Instantiation” on page 408
Time	Adding time-related operations: • “Designing with Temporal Operators” on page 349
Choice	Adding choice operations: • “Designing with Choice Operators” on page 331
Graphical design toolbars:	
Nudge	Moving selected design elements one pixel up, down, left or right: • “Nudging Elements” on page 291.
Rotate	Rotating elements to left or right: • “Rotating Elements” on page 290.
Layout	Managing alignment, distribution, and size of elements: • “Aligning Elements” on page 284 • “Aligning Design Components” on page 284 • “Distributing Elements and Connection Wires” on page 288 • “Resizing Elements” on page 288
Symbol Drawing	Creating and editing object symbols: • “Shaping Graphical Symbols” on page 297 • “Drawing Graphical Symbols” on page 298 • “Editing and Inserting Text in Symbols” on page 299 • “Inserting Images in Symbols” on page 300
Style	Applying predefined or user-defined styles to design element symbols: • “Applying Styles in Diagrams” on page 138
Zoom/Pan	Switching active view between selection, zoom and pan modes: • “Zooming Design Content in Views” on page 148 • “Grabbing and Moving Design Representation” on page 148
Full Screen	Switching active view in full screen: • “Displaying Active View on Full Screen” on page 150

Model Navigation Toolbars

These toolbars provide shortcuts to some of the most common commands used for navigating in model content.

Find	Standard search commands: • “Finding and Locating Design Elements” on page 409
Navigate	Navigating to object declarations, between views, files, or HTML content: • “Navigating to Declaration from Diagrams” on page 419 • “Navigating between Opened Views and Files” on page 420 • “Navigating in HTML Documents” on page 420
Go To	Navigating between design objects with respect to package hierarchy and model organization, or navigating in window content: • “Navigating in Model and Package Hierarchy” on page 417
Browse	Finding dependencies between design elements: • “Searching for Design Dependencies” on page 412
Script	Launching Script Wizard and executing scripts: • “Using Script Wizard for Tcl Scripting” on page 169 in <i>SCADE Java/Tcl API Guide</i> • “Using Script Wizard for Python Scripting” on page 28 in <i>SCADE Python API Guide</i>

Tool-Specific Toolbars

These toolbars provide shortcuts to some of the most common commands used for managing and running tools integrated to SCADE Suite environment.

Reporter	Quick access to documentation generation, see “Reporter Toolbars” on page 25 of <i>SCADE LifeCycle User Manual</i>
Requirements Management	Quick access to requirements traceability, see “ALM Gateway Toolbar” on page 88 of <i>SCADE LifeCycle User Manual</i>
Synchronization	Quick access to design synchronization between SCADE Suite and SCADE Architect models, see “Synchronizing System and Software Design” on page 424 of <i>SCADE Architect User Manual</i>
Simulator and Simulator Extension	Quick access to simulation activities, see “Simulation Toolbars” on page 579

Test Creation and Test Tool	Quick access to testing activities as described in “Test Environment Toolbars” on page 39 in <i>SCADE Test User Manual</i>
Timing and Stack Analysis Tools	Quick access to Worst-Case Execution Time and stack usage evaluation as described in “Timing and Stack Toolbar” on page 788
Metrics and Rules Checker	Quick access to metrics computation and rules check on model as described in “Metrics and Rules Checker Toolbar” on page 497
Design Verifier	Quick access to formal model verification activities as described in “Design Verifier Toolbar” on page 722
Code Generator	Quick access to code generation as described in “Code Generation Toolbar” on page 829

Symbol Editor

Symbol Editor displays the graphical symbol of any operator selected in the Diagram View and enables editing the symbol of design elements. For user instructions, see [“Editing Design Elements with Symbol Editor”](#) on page 294.

Status Bar Display

The status bar may display the following type of information:

- Help about selected toolbar buttons.
- Error messages related to graphical operations.
- Status of tool execution: when running, the indicator is , otherwise .

Model Development Workflow

The following figure describes the SCADE Suite general workflow:

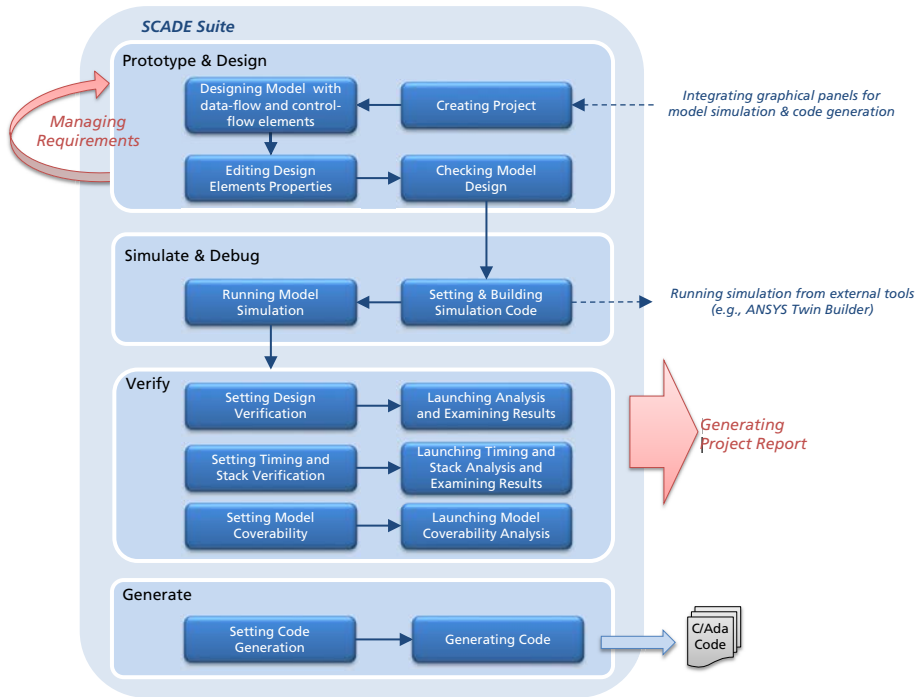


Figure 1.13: Typical workflow with SCADE Suite

This workflow consists of successive tasks that you can iterate as you need. It can follow several phases as described below:

- 1 Creating new project: see [“Creating New Project”](#) on page 96
- 2 Designing models with data-flow and control-flow: Chapter 3 about [“Designing Models”](#), Chapter 4 about [“Designing with Predefined Operators and Constructs”](#), Chapter 5 about [“Navigating in Models”](#), Chapter 6 about [“Editing Properties of Design Elements and Objects”](#), and Chapter 7 about [“Computing Metrics and Checking Rules on Model”](#).

- 3 Controlling revisions and refactoring model: Chapter 8 about [“Change Control of Models”](#) and Chapter 9 about [“Managing Model Refactoring”](#).
- 4 Checking model design: [“Checking Model Semantics”](#).
- 5 Simulating models using SCADE Suite Simulation or external tools: see Chapter 12 about [“Running Simulation in SCADE Suite”](#) or Chapter 13 about [“Running Simulation from External Tools”](#)
- 6 Launching formal verification sessions or timing and stack usage analysis: see Chapter 16 about [“Running Formal Verification Sessions”](#) and Chapter 17 about [“Verifying Timing and Stack Usage”](#)
- 7 Launching coverability analysis sessions on model design before testing: see Chapter 10 about [“Running Coverability Analysis on Design”](#)
- 8 Generating code: see Chapter 19 about [“Setting and Launching Code Generation”](#)
- 9 Generating project report: see Chapter 2 about [“Reporting about Suite Design”](#) from *SCADE LifeCycle User Manual*
- 10 Managing project traceability: see Chapter 7 about [“Managing Requirements and Traceability”](#) from *SCADE LifeCycle User Manual*

Standard File and Project Management

Read about the following operations:

- [“Opening/Closing Files”](#)
- [“Opening Recently Opened Projects and Files”](#)
- [“Opening/Closing Workspace Files”](#)
- [“Canceling/Restoring Previous Actions”](#)
- [“Saving Files and Workspace Files”](#)

Opening/Closing Files

You can open existing projects (.etp), HTML source files, Tcl files, or textual files provided you have read-access to the corresponding files. Opening a project loads the project, its objects, and its workspace. You may also load a project from its workspace, see [“Opening/Closing Workspace Files”](#) on page 54.

A project can only be edited inside a workspace: if the opened project has no associated workspace, a new workspace is automatically created.

To open files

- 1 Select *File > Open* (Ctrl+O) or click **Open**  in **Standard** toolbar.
- 2 Select a file format in drop-down list, browse for files in the **Open** dialog.
- 3 Select the file and click **Open** to validate. If an unsaved workspace is open at file opening, answer prompt: click **Yes** to save the workspace and load the selected file; click **No** to continue opening the file without saving the open workspace; click **Cancel** to stop opening.

Check in **Messages** output tab whether loading is successful. Various information may display. About errors at file opening, see [“Inserting Existing Project or Files”](#) on page 112. In general, closing views requests confirmation to update the model if the content of the view was edited and left unsaved. In case a warning message about duplicate OIDs in the model displays when loading a project, duplicate OIDs in the model are fixed at next saving (except

in libraries). In case of duplicate OIDs inside a library, it is necessary to load and save the library to fix them. In case of duplicate OIDs between two different models, it is necessary to fix the issue out of the tool. Having duplicate OIDs does not impact the models but may cause malfunction of traceability links, annotations, and quick fixes.

To close the active view

- 1 Select the view you want to close if it is not the active one.
- 2 Select *File > Close* or close the corresponding window.

Graphical views always close with all changes saved without requesting confirmation. Closing Symbol Editor, text diagrams, Tcl files or any other file always requests confirmation before closing. Click **Yes** to apply changes (or save the file) and close; click **No** to close without saving; click **Cancel** to stop closing.

Opening/Closing Workspace Files

You can load the content of SCADE Suite projects from an existing workspace file (.vsw) or from the project files (.etp). Opening a workspace closes all open views of the active project and displays new project content in the **File View** and **Scade View**. When opening a project, you get prompted for saving all modified files relative to the active workspace.

When loading projects from ETP files in directories, a confirmation message asks if a workspace corresponds to the project file. Click **Yes** to resume project opening. If the opened project has no associated workspace, a new workspace is automatically created.

To open project workspace

- 1 Select *File > Open Workspace* or click **Open**  in **Standard** toolbar.

- 2 Browse for the workspace file to open in the **Open** window and select it.
- 3 Click **Open** to load the selected project workspace. If an unsaved workspace is already open, answer prompt: **Yes** to save the workspace and load the selected file; **No** to open the file without saving the open workspace; **Cancel** to stop opening.

Check in the **Message** output tab whether loading is successful. Error messages may display in this tab. In some cases, the message is only about missing information which does not prevent the opening of the workspace. About errors detected when opening project workspace, see [“Inserting Existing Project or Files”](#) on page 112.

In general, closing the active workspace requests confirmation to update the model in case you edited the content of the view and left it unsaved. Graphical views always close with all changes and without requesting confirmation. For Symbol Editor, text diagrams, Tcl files, or any other file, a dialog asks whether to save modifications before closing the workspace.

To close the active workspace

- Select *File > Close Workspace*.

If the model was edited but unsaved, you are prompted for confirmation. Click **Yes** to apply changes to the open model and close; click **No** to close without saving; click **Cancel** to stop closing.

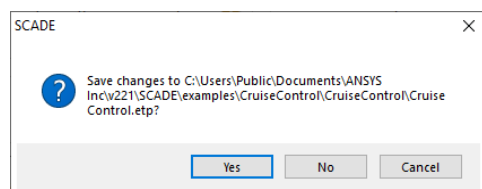


Figure 1.14: Example of confirmation message when closing workspace

If the workspace you close after editing contains a project and/or project files in read-only access, a confirmation message pops up. It is recommended to click **No** to close the message without saving the workspace. Answering **Yes** may introduce incoherences in the model: a warning message lists read-only files that cannot be saved and may suffer data loss. Files are identified as follows:

- Files with red symbol are in read-only access and cannot be saved.
- Files with a green checkmark are saved once you click **OK** on the dialog.

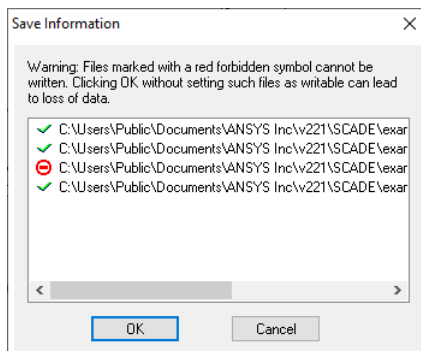


Figure 1.15: Display of Save Information dialog

Warning

Clicking OK on the Save Information dialog saves changes in files which are not reported as read-only with the risk of incoherences.

Saving Files and Workspace Files

File saving can be done in different contexts. You perform file saving:

- [“To save files and associated files”](#)
- [“To save SCADE Suite descriptions as new file”](#)
- [“To save all”](#)
- [“To save project as template”](#)
- [“To save active workspace”](#)

When saving files in read-only access, a warning message pops up. Click **OK** to close the message. Since you may introduce incoherences in the model, the **Save Information** dialog warns about the list of files that can be saved due to modifications and those that cannot be saved as in [Figure 1.15](#).

To save files and associated files

- Click **Save**  in **Standard** toolbar (Ct r L+S). If selecting a project file, the project is saved with all modified files. If selecting an XSCADE file, this file only is saved.

To save all

- Click **Save All**  in **Standard** toolbar (Ct r L+L). SCADE Suite saves the workspace, its projects, all modified model files, and any open files.

To save active workspace

- Select *File > Save Workspace*. SCADE Suite saves the workspace, its projects as well as all modified model files.

To save SCADE Suite descriptions as new file

- 1 Select *File > Save As* if you want to save in a new file. You can save as from **TCL Views**, **Symbol Views**, or **HTML Views**. Save as does not apply to projects: to rename projects, see [“Renaming Projects”](#) on page 102.
- 2 Name the new file in the **Save As** dialog or browse to select an existing file and overwrite it.
- 3 Click **Save** to validate and confirm overwriting when requested. The new file containing symbol or textual description is not referenced in the current project. If needed, the file can be inserted in the project afterwards.

A template file contains the same information as the current project: files and libraries, project configurations and tool settings, as well as project properties. Such files can be reused to create new projects. If you have a generic project with specific option files (. t o t), libraries, and settings for code generation and simulation, you can base new projects on this template and retrieve all properties automatically at creation (see [“Creating New Project”](#) on page 96).

To save project as template

- 1 Select a project file from **Scade View** or **File View**.
- 2 Select *File > Save as Template* to open the **Save As** dialog in the `templates` folder of SCADE Suite installation.
- 3 Name the template in the **File name** field, or select an existing template file (. e t t extension) to overwrite it.
- 4 Browse to locate the template file to be saved elsewhere.
- 5 Click **Save** to validate and confirm overwriting when requested.

Opening Recently Opened Projects and Files

You can open models/files from the list of most recently opened projects and files.

To open recently opened projects

- 1 Select *File > Recent Workspaces*.
- 2 Select the project you want to open from the available list. SCADE Suite opens and displays the selected project.

To open recently opened files

- 1 Select *File > Recent Files*.
- 2 Select the file you want to open from the displayed list. SCADE Suite opens and displays the selected file.

Canceling/Restoring Previous Actions

It is always possible to cancel the previous editing action(s) you performed or to restore the last editing action(s) you canceled in SCADE. You cannot undo actions involving file saving. If you close a view after performing various actions, you can undo these actions but the view does not reopen automatically.

To undo previous actions

- Select *Edit > Undo* (Ctrl+Z) or click **Undo**  from the [Standard toolbar](#) to cancel the last editing action. Repeat to undo several actions.

To redo canceled actions

- Select *Edit > Redo* (Ctrl+Y) or click **Redo**  from the [Standard toolbar](#) to restore the last canceled action. Repeat to restore several actions.

SCADE Suite Files

The set of files used and produced by SCADE Suite while you are editing models are the following ones:

[“Workspace and Project Files”](#)

[“Libraries Files”](#)

[“SCADE Files”](#)

[“Scenario and Application State Files”](#)

[“Simulation Output and Observability Files”](#)

[“Semantic Error File”](#)

[“Annotation Type Files”](#)

[“Annotation Files”](#)

[“Symbol Files”](#)

[“Font Files”](#)

[“Option Files”](#)

[“Reporting Files”](#)

[“Log Files”](#)

File extensions

.vsw, .etp, and .ewo

.etp

.xscade and .scade

.in, .sss, .sts, and .sns

.out, .csv, .obs

.err

.aty

.ann

.ssl

.fonts and .ffonts

.tot

.htm

.log

Workspace and Project Files

SCADE Suite models are described in a project file (.etp), which lists the files where model information is stored. A project is included into a workspace stored as VSW file (.vsw). Several projects can be grouped together in the same workspace. An EWO file (.ewo) containing only user preferences is associated with the project workspace.

Libraries Files

Libraries can be used to structure user definitions in SCADE Suite projects (constants, nodes, types, etc...). Like any SCADE Suite model, a library can reference other libraries. In general, a model is a hierarchy of models: cycles in such hierarchy are not allowed. For instance, if model M uses library L1, which uses library L2, then, L2 cannot use library L1.

This notion of library is not directly supported by Scade language, but rather by SCADE Suite environment. The principle is described below:

- Any library used by a model is specified in the model project. When opening this project, SCADE Suite Editor can access the library in read-only mode.
- Any library used at any level in the hierarchy is displayed in **Scade View**, and their elements can be used in the model.

Note

Operators, constants, types, and sensors can be contained at SCADE Suite project root or defined in packages. The packages prevent naming conflicts between operators/types/constants defined in a package, and those defined in other packages or directly at project root.

Basically, this mechanism behaves as the `#include` directive of the C language, with the management of multi-inclusion. For details, see [“Managing Libraries”](#) on page 115.

SCADE Files

The SCADE files are read and loaded at project launch; they contain all descriptions declared in the model. These files are referenced in the project file (.etp). The following SCADE files are used in SCADE Suite projects:

- .xscade: These files contain design functions in XML format. All elements created in Editor are stored in this format. These files are updated whenever you save a model that contains SCADE elements modified since last save. For reference details, see Chapter 2 about [“Editor Save Format”](#) in *SCADE Suite Technical Manual*.
- .scade: These files contain design definitions in textual format that can be edited directly from SCADE Suite Textual Editor or outside SCADE Suite as any text file. They can be inserted in SCADE Suite projects via the *Insert files* command. Once loaded without syntactic error, SCADE Suite elements described in such files are created in read-only mode and can be used in models as any other element.

Scenario and Application State Files

Scenarios and application states can be used for storing and replaying simulation steps. Scenario files (.sss or .in) allow to describe simulation steps by setting input values and expected output values. Application state files allow to describe application states at any given cycle either in binary format (.sts) or in text format (.sns). For more usage details, see [“Recording Simulation Scenarios/Application States”](#) on page 614.

Simulation Output and Observability Files

Output values and observable variables produced during simulation sessions at each cycle can be saved in text files. Output values are saved in output files (.out or .csv). Observable variables are saved in observability files (.obs). For more usage details, see [“Saving Simulation Output”](#) on page 606.

Semantic Error File

The semantic error file (.err) contains diagnoses produced by semantic checks performed on SCADE Suite models. It is referenced in the project file (ETP files). You can specify its name and location when creating the project or by editing project properties. By default, SCADE Suite Editor creates this file in the project directory and names it according to the project name. The ERR file is updated when semantic checks are launched.

Annotation Type Files

The annotation type files (.aty) define the types of notes that can be created when editing a model. They are referenced in the project file (ETP files). By default, the ATY file of the standard product installation is used. Annotation type files are read and loaded when a model is edited. Editing notes is impossible if no annotation type file exists or if they are in read-only mode, empty, or syntactically incorrect. These files must be edited outside SCADE Suite using any text editor. For reference details, consult Chapter 1 about [“Annotation Mechanism”](#) in *SCADE Suite Technical Manual*.

Annotation Files

All model annotations are saved as ANN files (.ann). Each ANN file is associated with an XSCADE file and contains annotations on elements defined in XSCADE files. The name of annotation files derives from the name of the associated XSCADE file. Annotation files are related to the files associated with each element:

- If element saved in a separate file, annotations are saved in a separate file.
- If element saved in a parent file, annotations are saved in the parent annotation file.

SCADE Suite Editor prevents users from inserting annotations on elements defined textually. Annotation files are read and loaded when an existing model is loaded. They are updated when saving a model containing elements modified since last save. For reference details, consult Chapter 1 about [“Annotation Mechanism”](#) in *SCADE Suite Technical Manual*.

Symbol Files

The graphical representation associated with any SCADE operator is saved in SSL files (.ssl) and considered as a graphical property. Such files are created when using Symbol Editor to create new symbols and loaded when the graphical representation it describes is set as operator property. These files are updated when applying changes in Symbol Editor. For lexical and syntax details, consult Chapter 3 about [“Specific SCADE Suite Files”](#) in *SCADE Suite Technical Manual*.

Font Files

External fonts used in Symbols or Annotation files can be specified in a FONTS file (.font s). This file must be located in the same directory as the project and have the same name suffixed by font s. A default font file (.ffont s) is in the /tv g of SCADE installation folders.

Option Files

SCADE Suite Editor options are saved in TOT files (.tot). Such TOT files are referenced in the project file (.et p). Several option files can coexist in SCADE Suite projects. For user instructions about option files, see [“Selecting Advanced Options”](#) on page 84. For reference details, see Chapter 3 about [“Specific SCADE Suite Files”](#) in *SCADE Suite Technical Manual*.

Reporting Files

SCADE Suite integrates several modules which are able to generate reports in HTM format. An HTM report is produced after launching model check on operators, after running model or object comparison and launching Model Diff Report, after importing models with SCADE Suite Gateways, after comparing analysis sessions with Timing and Stack Verifiers, or after launching model analysis with Design Verifier.

Log Files

SCADE Suite log files (.Log) may be generated by SCADE Suite Code Generators or SCADE Suite Simulation.

Multifile Management

To facilitate the management of the project files in a teamwork context, SCAD Suite Editor saves any operator or package of the model in its own individual file and allows locating any object. You can customize multifile management according to one's own needs and locate objects in the model from the project files.

- [“Default File Management”](#)
- [“Locating Objects from SCAD Suite Files”](#)
- [“File Management Customization”](#)

Default File Management

By default, a SCAD Suite project is associated with a `<ProjectName>.etp` file which contains all the `.xcsade` and `.scade` files of the model.

Tip

Edit the default model file assigned at project creation in the project creation wizard (see [“To create a new project”](#) on page 96) or from the properties of the project (see [“To edit the SCAD Suite properties of a project”](#) on page 426).

[Figure 1.16](#) shows the correspondence between objects and `.xscade` files in default multifile management.

By default, file management in SCADE Suite projects behaves as follows:

Any root operator (declared outside any package) or child operator (declared within a package) and its content, are saved in <OperatorName>.xscade file.

Change file name from the general properties of the operator (see [“To edit the general properties of an operator”](#) on page 444).

Any root or child package declared in the SCADE Suite model are saved respectively in <PackageName>.xscade and <ChildPackageName>.xscade files.

Change each file name from the general properties of the package (see [“To edit the general properties of a package”](#) on page 434).

Any root constants, sensors, or types (declared outside any package) are saved in <ProjectName>.xscade file.

Change file name from the general properties of each constant, sensor or type (see [“Editing Type Properties”](#) on page 438, [“Editing Constant Properties”](#) on page 440, or [“Editing Sensor Properties”](#) on page 442).

Any child constant, sensor, or type (declared in a package) is saved in the same file as the package in which it is declared.

Change file name from the general properties of the containing package (see [“To edit the general properties of a package”](#) on page 434).

Note

If the default file is missing or invalid, SCADE Suite prompts you to specify a new default file and location.

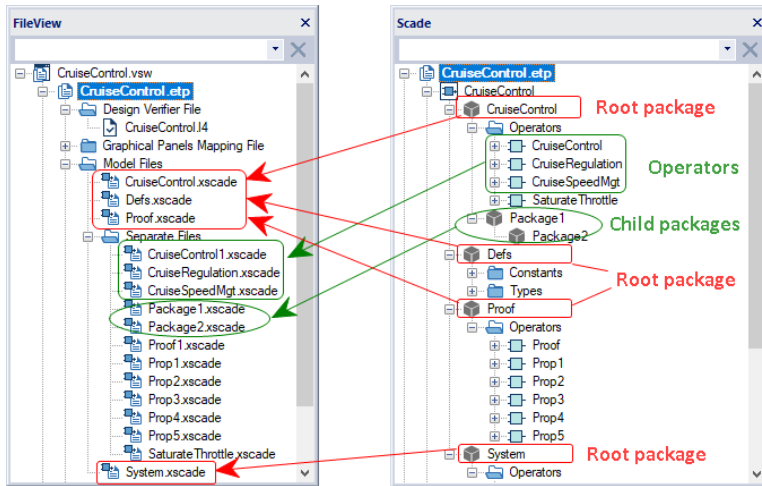


Figure 1.16: Illustration of default multifile management

File Management Customization

SCADE Suite allows customizing the default file management in the project. You can control how any *child operator* and *sub-package* are saved, at project level, and at any child operator or sub-package level:

- [“Controlling File Management at Project Level”](#)
- [“Controlling File Management at Object Level”](#)

Controlling File Management at Project Level

You can set file management at project level from the general properties of SCADE Suite environment to disable the automatic association of any new child operator or sub-package (declared in a package) with a separate file. Setting file management from these general properties does not change already existing files. You can change any file in which child operators or sub-packages were previously saved from the general properties of each object, as described in [“Controlling File Management at Object Level”](#) on page 69.

To control file management at project level

- 1 Select *Tools > Options* and go to **General** options.
- 2 Set the **Associate a separate file at creation** option as required:
 - Check to save new child operator or sub-package into a separate file.
 - Uncheck to save new child operator or child package in the same file named <PackageName> .xscade corresponding to the top-level package (see [Figure 1.17](#)).

Tips

- Change this package file name on the first parent package saved in a separate file as described in [“To edit the general properties of a package”](#) on page 434.
- Unchecking Separate File Name from the general properties of child operator and sub-packages has same effect at object level.

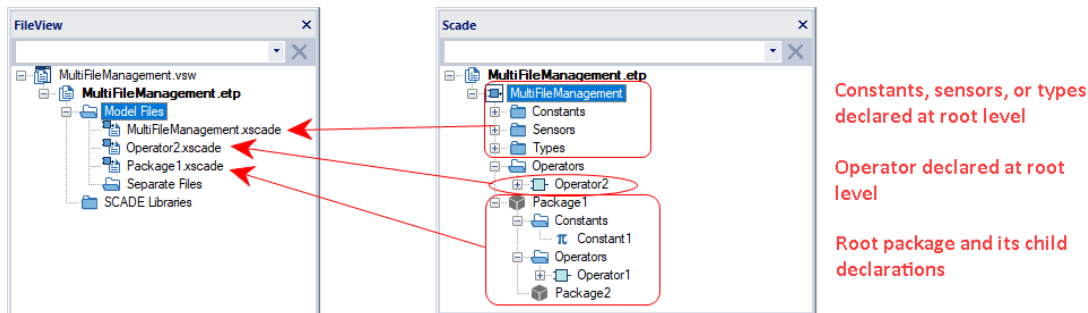


Figure 1.17: Default separation of files disabled at project level

Controlling File Management at Object Level

You can also set file management from the general properties of each child operator or sub-package to disable the automatic association of the selected object (child operator or sub-package) with a separate file.

For a package, the separate file is used to save the whole hierarchy, as long as included packages and operators are not themselves saved in their own separate file. The reverse is true: if you uncheck the **Separate File Name** option from the general properties of a sub-package, its whole hierarchy is saved in the same file as this package, as long as included packages and operators are not themselves saved in their own separate file.

To control file management at object level

- 1 Select a child operator or sub-package in **Scade View** and go to **General** in the **Properties** window.
- 2 Set the **Separate File Name** option as required:
 - Check to save the child operator or sub-package into a separate file. Filename above is automatically enabled and renamed.
 - Uncheck to save the selected child operator or sub-package in the same file as the package containing the object declaration (see [Figure 1.18](#) and [Figure 1.19](#)).

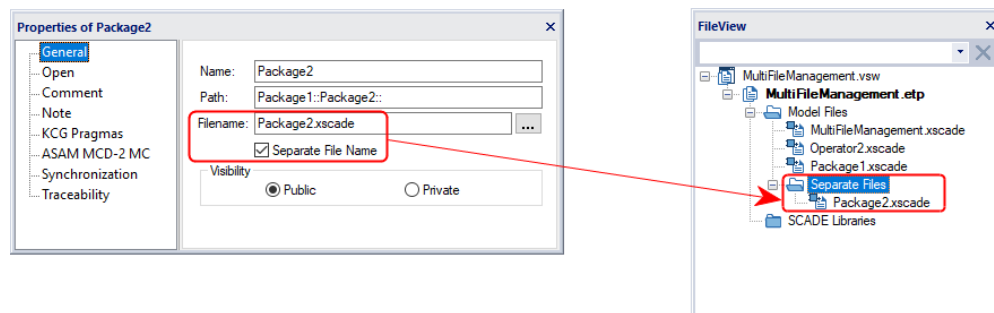


Figure 1.18: File Management settings at object level: default file separation enabled

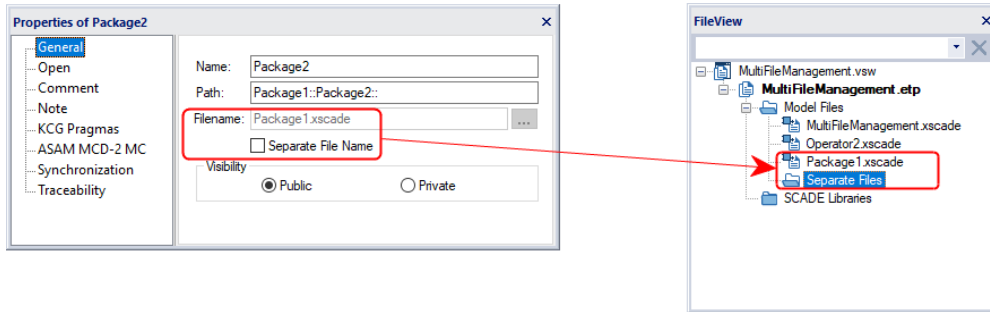


Figure 1.19: File Management settings at object level: default file separation disabled

Tips

- Change the package file name on the first parent package saved in a separate file as described in [“To edit the general properties of a package”](#) on page 434.
- Unchecking Associate a separate file at creation from Tools > Options has the same effect at project level (see [“Selecting General Options”](#) on page 77).

Locating Objects from SCADE Files

The model files available from **File View** allow the following actions:

- Checking which root object(s) is/are described in the selected file.
- Checking which child object is described in the selected file and its hierarchy in the model.
- Locating any object in the model.

You can access this information from **SCADE** in the properties of any .scade or .xscade file, as described in [“Editing Project Files Properties”](#) on page 431.

2 / Setting Project and Design Environment

This chapter covers all features and options which allow to configure SCAD Suite environment and facilitate the display of SCAD Suite model parts.

- [“Setting SCAD Suite Tools Options”](#)
- [“Customizing Tools Menu”](#)
- [“Managing Perspectives”](#)
- [“Managing Projects”](#)
- [“Managing Libraries”](#)
- [“Managing Default or Custom Configurations”](#)
- [“Organizing Model Design with Packages”](#)
- [“Managing Favorites and Using Shortcuts”](#)
- [“Creating and Modifying Styles”](#)
- [“Enabling Display Options in Diagram Views”](#)
- [“Organizing and Working with Model Editing Views”](#)
- [“Organizing and Working with Docking Windows”](#)

Setting SCADE Suite Tools Options

You can set up the working environment by setting parameters related to SCADE Suite tools. By default, all option changes are saved in a user option file.

- [“Selecting Environment Options”](#)
- [“Selecting General Options”](#)
- [“Selecting Views Options”](#)
- [“Setting Simulation Environment”](#) on page 583 for setting **Simulator** options
- [“Setting Timing and Stack Result Options”](#) on page 797 for setting **Timing and Stack** options
- [“Selecting Commands Options”](#)
- [“Selecting Language Construct Options”](#)
- [“Selecting Advanced Options”](#)

Selecting Environment Options

You can set the SCADE Suite working environment with options related to file saving policy, automatic file backups, application workspace basic behavior, connection to Source Code Control Systems, and access to advanced options.

To select Environment options

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Environment** or use **Ct r l+Tab** key to switch option display.

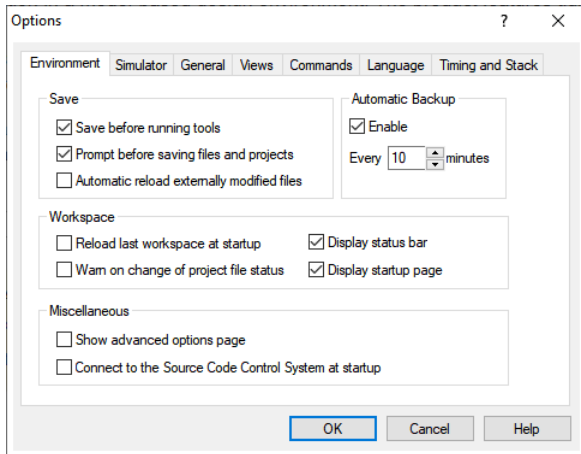


Figure 2.1: Options related to SCADE Suite working environment

3 Check/uncheck the **Save** options as needed:

- **Save before running tools** to save automatically the modifications each time any SCADE Suite tool is launched.
- **Prompt before saving files and projects** to confirm before saving files. SCADE Suite saves the corresponding files upon exiting, when loading another workspace, and before running any external tool (if previous option checked).
- **Automatic reload externally modified files** to reload automatically the project files modified outside SCADE Suite. If unchecked, SCADE Suite asks whether to reload the files. *Does not apply to SCADE files (.scade and .xscade).*

4 Set the **Automatic Backup** options as needed. Check **Enable** to activate automatic backup and allow restoring project changes (model description and annotations) after application downtime. Set the frequency of automatic backup using the up/down arrows: from 1 minute to 120 minutes. Project backup is done regularly as preset and is kept till next project save. Only changes done in .etp, .xscade, and .ann files are

backed up. Any changes stored in other project files like traceability links are not backed up. At project loading, if a backup version exists, SCADE Suite asks whether to load the backup state:

- Answer **No** to restore the project in the last saved state losing definitively possible updates contained in the backup.
- Answer **Yes** to restore the project from its backup state. Note that the backup state is not kept until the Save button is pressed. If the backup state is not correct for any reasons, it is still possible to restore/reload the project in the last saved state. Note that the backup content will no more be available then.

5 Check/uncheck the **Workspace** options as needed:

- **Reload last workspace at startup** to open and load the last workspace automatically at startup.
- **Warn on change of project file status** to get a warning when the access right of a project file is Read Only to avoid losing modifications.
- **Display status bar** to show/hide a **Status** bar at bottom of application.
- **Display startup page** to show/hide a startup page providing useful information and links to projects, documentation, distributed models, or support contacts.

6 Check/uncheck the **Miscellaneous** options as needed:

- **Show advanced options page** to show/hide the **Advanced** options to manage project option files and control options (see [“Selecting Advanced Options”](#) on page 84).
- **Connect to the Source Code Control System at startup** to enable SCC commands from SCADE Suite project. A Source Code Control System (SCC System) must be installed (see details in Appendix C about [“SCADE Suite and Source Code Control”](#)).

Selecting General Options

Once a project is loaded, you can set the general behavior of SCADE Suite with options related to file management, default types when declaring new types, access to dump information, and graphical symbol management.

To select General options

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **General** or use `Ctrl+Tab` key to switch option display.

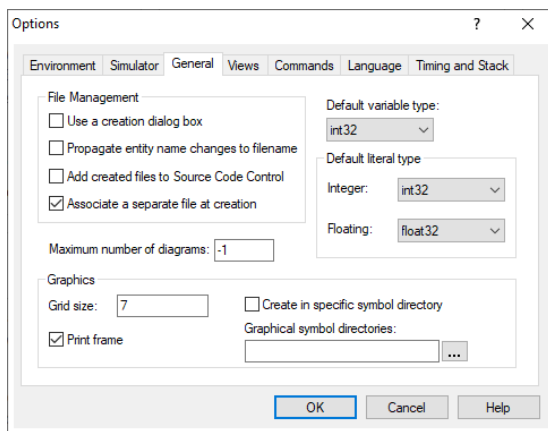


Figure 2.2: Options related to general behavior of SCADE

- 3 Check/uncheck the **File Management** options as needed:
 - **Use a creation dialog box** to allow editing the default **Name** and **File name** at each package or operator creation. If unchecked, a default naming pattern is used.
 - **Propagate entity name changes to filename** to modify filenames automatically when renaming the corresponding package or operator saved in separate files. When renaming an operator from `OldName` to

NewName: if the old file name was OldName.xscade, the new file name becomes NewName.xscade.; if the file name was OtherFileName.xscade, it remains OtherFileName.xscade.

- **Add created files to Source Code Control** to include new files in Source Code Control. For instructions about SCC in SCADE Suite, see Appendix C about [“SCADE Suite and Source Code Control”](#).
 - **Associate a separate file at creation** to associate a separate file to new operator or package (see [“Multifile Management”](#) on page 66).
- 4 Specify the maximum number of diagrams per operator or state in the **Maximum number of diagrams** field. Keep the default value -1 if you need no limitation of diagrams.
 - 5 Set **Default variable type** assigned to new inputs, outputs, hidden inputs, locals, probes and constants from drop-down list. Select <none> if no need to assign a default type at variable creation.
 - 6 Set **Default literal type** assigned to new literals (see [“Creating Textual Expressions”](#) on page 272) from **Integer** and **Floating** drop-down lists.
 - 7 Set the **Graphics** options as needed:
 - Specify in millimeters the default interval between grid lines in the **Grid size** field. This value can be individually modified for any Diagram View, State Machine View or Symbol View.
 - Check **Create in specific symbol directory** to create new symbols in the first directory specified in the **Graphical symbol directories** field (see [“Editing Layout of Graphical Symbols”](#) on page 295).
 - Specify the directory where SCADE Suite searches first for symbol files in the **Graphical symbol directories** field. Use comas (,) to separate directory names if you specify several ones.
 - Check **Print frame** to print out sheets surrounded by a frame. If unchecked, there is no visible frame on print-outs.

Selecting Views Options

After loading a project, one can set up what kind of information and assistance the design views display and how information is displayed in views. Some options are also available contextually from diagrams, see [“Enabling Display Options in Diagram Views”](#) on page 143.

To select Views options

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Views** or `Ctrl+Tab` key to switch option display.

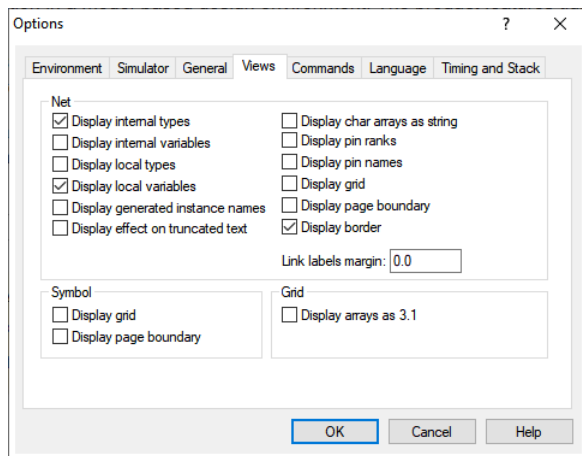


Figure 2.3: Options related to SCADE Suite Editor views

- 3 Check/uncheck the **Net** options as needed:
 - **Display internal types** to show/hide the type of internal variables over the connections (whether types are already hidden or displayed).
 - **Display internal variables** to show/hide the name of internal variables over the connections (whether names are already hidden or displayed).
 - **Display local types** to show/hide the type of local variables over the connections (whether types are already hidden or displayed).

- **Display local variables** to show/hide the name of local variables over the connections (whether names are already hidden or displayed).
- **Display generated instance names** to show/hide all instance names including those automatically generated by SCADE Suite in textual expressions used in diagrams.
- **Display effect on truncated text** to highlight truncated information in graphical diagrams with a light-gray hatch background in addition to the ellipsis (...) mark.
- **Display char arrays as string** to display char arrays as text string in graphical diagrams, Constants Views, and simulation windows (applies only to displayed fields and not to semantic features like copy as text, values given to the Tcl interpreter). If unchecked, the character by character representation is used. For instructions, see [“To define char array values”](#) on page 210. To enter strings smaller than the specified array dimension, enable the use of padding characters as explained in [“To edit the layout properties of a project”](#) on page 428.
- **Display pin ranks** to show/hide the rank of operator calls pins.
- **Display pin names** to show/hide the input/output names corresponding to each operator call pin.
- **Display grid** to show/hide a background grid in all graphical diagrams.
- **Display page boundary** to show/hide the diagram border based on default page format of the active project. Set default format from project Properties as explained in [“To edit the page format properties of a project”](#) on page 427.
- **Display border** to show/hide the size of states and actions with a red dotted line when states/actions displayed in separate diagrams. For changing state/action display, see [“Switching Embedded/Hidden State Display”](#) on page 200 and [“Switching Embedded/Hidden Action Display”](#) on page 201.

- Set in pixels the bottom margin between the connecting wires and their labels in the **Link labels margin** field.
- 4 Check/uncheck the **Symbol** options with the following check boxes:
- **Display grid** to show/hide a background grid in Symbol Views.
 - **Display page boundary** to show/hide the page border in dotted lines based on the default page format of the active project.
- 5 Check/uncheck the **Grid** options with the following check boxes:
- **Display arrays as 3.1** to switch the column-row display of array constants to the row-column display while editing array constant values. For editing array constants, see [“To set/edit values of constants of two-dimensions array type from a table”](#) on page 221.

Selecting Commands Options

These options enable/inhibit some of the commands available from SCAD Suite application menus and contextual menus when a project is loaded.

To select Commands options

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Commands** or use **Ct r l+Tab** key to switch option display.
- 3 Uncheck the command(s) to deactivate from menus and toolbars.

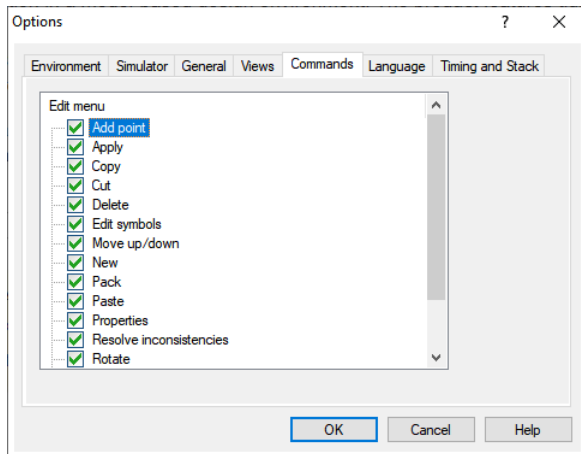


Figure 2.4: Options related to SCAD Suite Editor commands

- 4 Edit command selection as follows:
 - Hold **Ct r l**-key to select separate commands and press **Space** bar to check/uncheck the highlighted list.
 - Hold **Shift**-key to select consecutive commands and press **Space** bar to check/uncheck the highlighted list.
 - To check/uncheck by mouse click a multiselection list, keep the **Ct r l** or **Shift** keys pressed while clicking on check boxes.

Selecting Language Construct Options

These options control if Scade constructs are available at project loading.

To select Language options

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Language** or use **Ct r l+Tab** key to switch option display.
- 3 Uncheck the construct(s) to deactivate from menus and toolbars. Toolbars, menus, and Shortcuts Window display deactivated constructs as grayed.

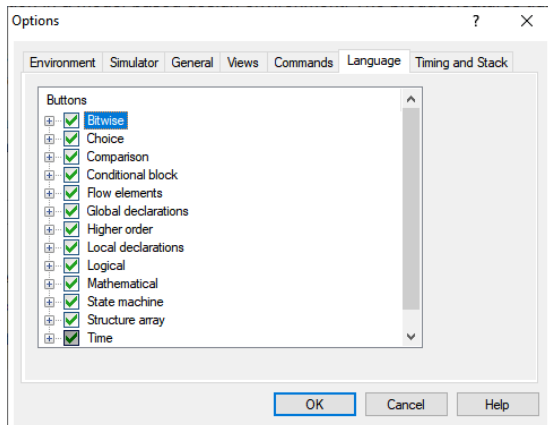


Figure 2.5: Options related to Scade language constructs

- 4 Edit command selection as follows:
 - Check a category of constructs to check all its content.
 - Hold **Ct r l**-key to select separate constructs and press **Space bar** to check/uncheck the highlighted list.
 - Hold **Shift**-key to select consecutive constructs and press **Space bar** to check/uncheck the highlighted list.
 - To check/uncheck by mouse click a multiselection list, keep the **Ct r l** or **Shift** keys pressed while clicking on check boxes.

Selecting Advanced Options

These options allow to organize the priority of option files in projects, create user [option files](#), and set options from any option file. Predefined option files are distributed with SCADE, but several option files can be used in projects. Project and user option files set the redefinition of option values. SCADE Suite merges the values set in the option files of a project with respect to priority order. In case of conflict, like two files setting the same option with different values, a priority rule applies: the file listed on top of the file tree sets the value. You can always change the order of option files.

- [“To create user option files and insert in active project”](#)
- [“To change the priority order of option files”](#)
- [“To select and set tools options from any option file”](#)
- [“To reset options to default in any option file”](#)
- [“To select and set options directly from user option files”](#)
- [“To lock options set in user option files”](#)

Check **Show advanced options page** in **Environment** options to access **Advanced** options for option file management (see [“Selecting Environment Options”](#) on page 74).

Remember that values in **Options** tabs are grayed and non-editable if an option file with higher priority redefines the option.

To create user option files and insert in active project

- 1 Select **Tools > Options** to view applicable options.
- 2 Display **Advanced** or use **Ctrl+Tab** key to switch option display.
- 3 Click **Save As** to open the **Save As** window. Enter the name of the new option file in the **File name** field (format is .t o t). Click **Save**. A pop-up asks whether to insert the new file in the current project.
- 4 Click **Yes** to insert the option file in **File View** and add the newly created option file in project directories. Click **OK** to validate. Alternatively, set tools options from user option files directly from **File View**.

To select and set tools options from any option file

- 1 Select **Tools > Options** to view applicable options.
- 2 Display **Advanced** or use **Ctrl+Tab** key to switch option display.
- 3 In **Servers** list, select SCADÉ modules to filter the categories of corresponding options to be set in option file.
- 4 In **Managers** list, select a category of options to filter between options.

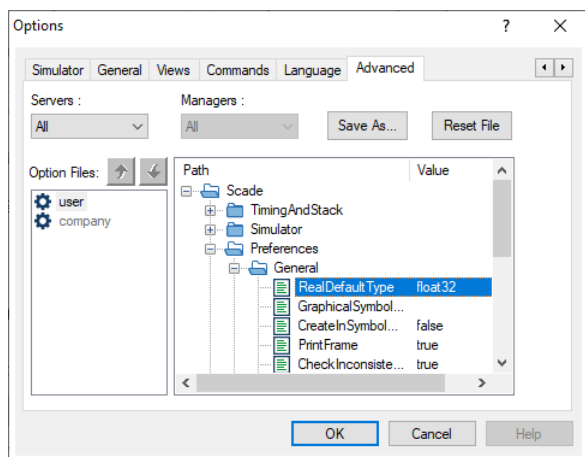


Figure 2.6: Options related to advanced file management

- 5 Select a file in the **Option Files** list (file needs to be in write access).
- 6 Browse the option tree on the right for the value(s) to edit:
 - **Red**: value modified
 - **Orange**: value redefined in a file of higher priority (editing ignored)
 - **Grey**: value cannot be redefined (read-only or locked option)
 - **Black**: value unchanged
- 7 Select the option, click in **Value** column and press <F2>. Edit the value by entering new value or selecting from drop-down list. Click **OK** to validate.

To select and set options directly from user option files

- 1 In **File View**, locate the appropriate user TOT option file and double click the file to open its editing window. SCADE Suite displays a tree view with option categories on the left and the corresponding options on the right.

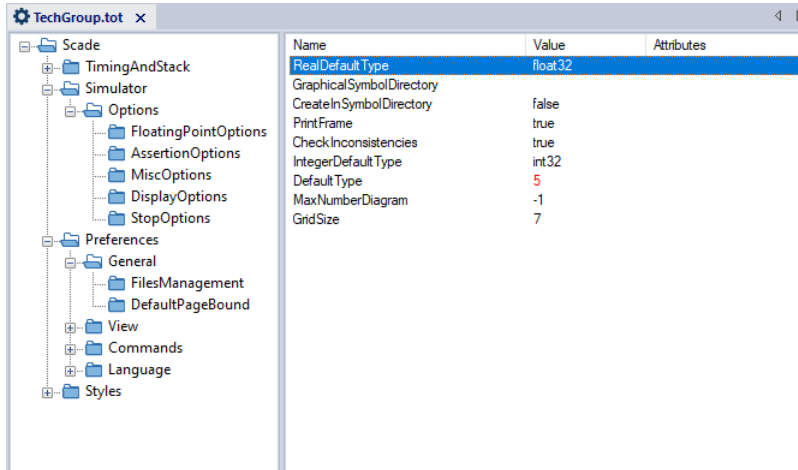


Figure 2.7: Setting options in user option file

- 2 Expand the tree of categories and select a folder to browse for options:
 - **Red:** value modified
 - **Orange:** value redefined in file with higher priority (editing ignored)
 - **Grey:** value cannot be redefined (read-only or locked option)
 - **Black:** value unchanged
- 3 Select the option, click in the **Value** column and press <F2>.
- 4 Edit the value by entering new value or selecting from drop-down list. It is possible to prevent options from being edited in files with higher priority (see ["To lock options set in user option files"](#) on page 88).
- 5 Select *File* > *Save* to save changes.

To change the priority order of option files

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Advanced** or use C t r l+Tab key to switch option display.
- 3 Select an option file in the **Option Files** list. It is not possible to change the priority of predefined option files. The user . tot and company . tot files are located respectively in <user>/AppData/Roaming/Ansys/SCADE and ProgramData/Ansys/SCADE (Windows 10). Predefined files can be set in init . tot located next to the company . tot file.

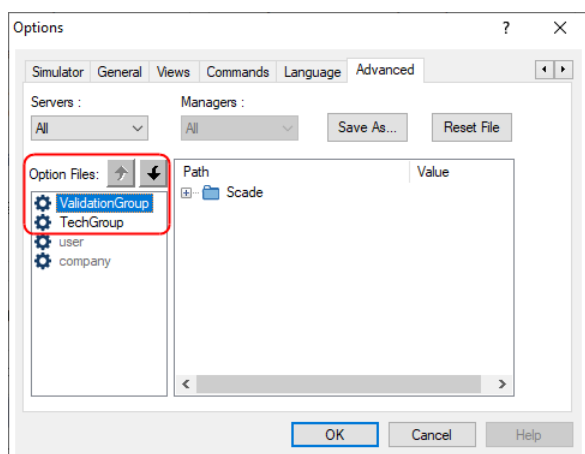


Figure 2.8: Arranging order of option files

- 4 Click **Up**  or **Down**  to change priority order. Click **OK** to validate.

To reset options to default in any option file

- 1 Select *Tools > Options* to view applicable options.
- 2 Display **Advanced** or use C t r l+Tab key to switch option display.
- 3 Select an option file and click **Reset File** to reset options to default values.

To lock options set in user option files

- 1 From **File View**, double click the TOT option file to start editing.
- 2 Browse for the option to be locked in the category folders.
- 3 Right-click the option to lock: the **Attribute** column is set to `forced=true`. The option cannot be defined again.
- 4 Select *File > Save* to save changes.

Customizing Tools Menu

You can associate new commands with any program running on the operating system. You may specify arguments for all new commands that display in *Tools* menu.

To add custom commands in Tools menu

- 1 Right-click in toolbar area and select *Customize* (or *Tools > Customize* menu) to display **Tools**.

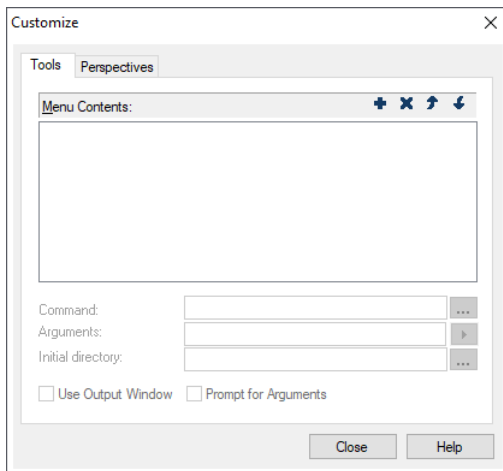


Figure 2.9: Defining commands to insert custom menus

- 2 In **Menu Contents** list, click **New +** to define a new command.
- 3 Enter the menu title in the blank line and press Enter.
 - To specify a letter in the menu as a shortcut key, type ampersand (&) in front of the letter. By default, the first letter in the title is the keyboard shortcut.
 - To specify sub-menu items, use the \ character.

- 4 In **Command** field, browse and specify the file path of the program or Tcl file associated with the new command.
- 5 In **Arguments** field, do one of the following to insert argument syntax:
 - Type the argument(s) to be passed to the program.
 - Click the arrow to select an argument from the drop-down list:

Argument	Syntax	Definition
File Path	\$(FilePath)	file path of the selected object
File Name	\$(FileName)	file name of the selected object
File Directory	\$(FileDir)	directory of the selected object file
File Extension	\$(FileExt)	extension of the selected object file
Project Path	\$(PrjPath)	file path of the project containing the selected object
Project Name	\$(PrjName)	name of the project containing the selected object
Project Directory	\$(PrjDir)	directory of the project containing the selected object
Active Project Path	\$(APrjPath)	path of the active project
Active Project Name	\$(APrjName)	name of the active project
Active Project Directory	\$(APrjDir)	directory of the active project
Current Directory	\$(CurDir)	current directory path
Workspace Path	\$(WsPath)	path of the current workspace
Workspace Name	\$(WsName)	name of the current workspace
Workspace Directory	\$(WsDir)	directory of the current workspace

- Leave field empty and check **Prompt for Arguments** to be prompted each time you select a custom command. Upon **Tools Arguments** prompt, click the arrow to set an argument from the list, check **Redirect to Output Window** to display in the Output Window, and click **OK** to run the program with the selected argument.
- 6 In **Initial Directory** field, browse to set the directory where the command is located.
 - 7 Check **Use output window** to specify that an application related outputs display in the Output Window rather than opening a “Console window”.
 - 8 Close to add the new command in the *Tools* menu for selection.

To edit custom Tools menu

- 1 Select the menu in **Menu Contents** list and do the following:
 - Modify menu title, command line, argument, or initial directory in matching fields.
 - Press **Delete**  to remove a command from *Tools* menu.

Managing Perspectives

You use [perspectives](#) to switch between various displays of the menu bar, toolbars, and docking windows using preset location and size. Default perspectives are available but one can create new ones. Each perspective stores a configuration of display. It can be set to optimize SCADE workspace organization and fit one’s needs in many contexts of use. At product launch, the last perspective active in previous session is loaded. The active perspective can store any changes in workspace layout when showing/hiding the menu bar, toolbars, and docking windows, or when changing their display size and/or position. Any modification is automatically saved.

- [“Default Perspectives”](#)
- [“Creating New Perspectives”](#)
- [“Switching Perspectives”](#)

Default Perspectives

Default perspectives are part of SCADE environment and cannot be deleted.

Table 2.1: Default perspectives available

Perspective	Modes of usage
Architect Design	Designing system architecture in SCADE Architect
Coverage	Running coverage analysis with SCADE Test Model Coverage
Diff	Running diff analysis on SCADE Suite models or objects
Scade Design	Designing control flow models in SCADE Suite
Simulation	Simulating SCADE Suite models (automatically loaded at launch)
Standard	Walking through models (applied at first application launch)
Testing	Designing and running tests with SCADE Test Environment
Timing and Stack	Running timing and stack analysis tools (Optimizer or Verifiers)

Switching Perspectives

You switch between perspectives to change the display of the design environment with respect to the tools and the workspace setup you need.

Note

Full screen mode is associated with the active perspective. A perspective is saved when leaving full screen mode.

To switch active perspective

- Select *View > Perspectives* and select the perspective name from the list (the active perspective shows a check mark in front). Alternatively, right-click in toolbar area and select *Perspectives*, or use the drop-down selection in **Perspective** toolbar.

Creating New Perspectives

You can create as many perspectives as needed to easily switch the user interface display between various contexts of use.

To create new perspectives

- 1 Select *Tools > Customize* or *View > Perspectives > Customize*, or right-click in toolbar area and select *Perspectives > Customize*, display **Perspectives**.

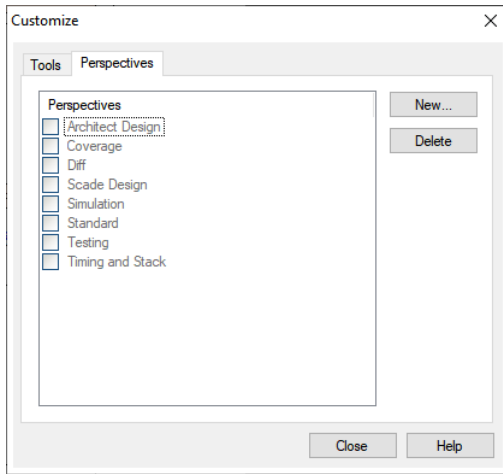


Figure 2.10: Perspectives options for creating/deleting perspectives

- 2 To delete perspectives, check a perspective name and click **Delete**. Grayed perspectives cannot be deleted: they correspond to the active one or to default ones.
- 3 Click **New** to open the **Create Perspective** window.

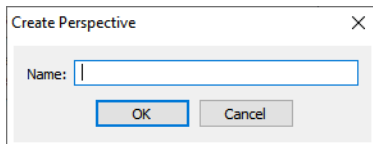


Figure 2.11: Create Perspective window

- 4 Enter the name of the perspective in the **Name** field.
- 5 Click **OK**. The new perspective is added and based on the active perspective; it becomes the active perspective.

Managing Projects

SCADE Suite Editor includes tools and features which facilitate the management of model design projects.

- [“Creating Blank Workspace”](#)
- [“Creating New Project”](#)
- [“Creating New Python Files”](#)
- [“Creating New Tcl Files”](#)
- [“Switching Active Project in Same Workspace”](#)
- [“Renaming Projects”](#)
- [“Managing Project Archives”](#)
- [“Inserting Existing Project or Files”](#)
- [“Creating New Folders in Projects”](#)
- [“Saving Files in UNIX Text Format”](#)

For details about project files, see [“SCADE Suite Files”](#) on page 60.

Creating Blank Workspace

You create blank workspaces (. vsw file extension) where you insert existing project files. When creating a new workspace, any active workspace is closed.

To create a blank workspace

- 1 Click **New**  in **Standard** toolbar or select *File > New*.
- 2 Display **Workspaces** options on the **New** window.

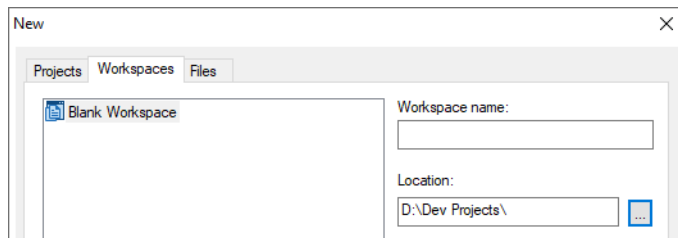


Figure 2.12: Defining blank workspace

- 3 Enter the name of the workspace in the **Workspace name** field. The **Location** field is updated automatically while entering the name.
- 4 Browse in project folders and set the file path in the **Location** field.

Creating New Project

You create new project files whenever you need to start building and designing a model in SCADE Suite.

To create a new project

- 1 Click **New**  in **Standard** toolbar or select *File > New*.

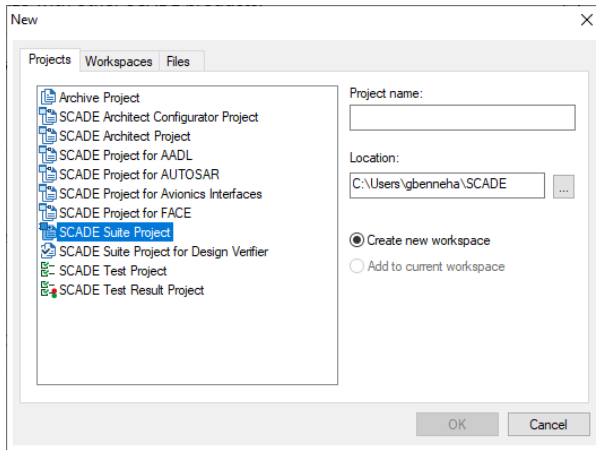


Figure 2.13: Opening New window to create new project

- 2 Select the type of project to create:
 - **SCADE Suite Project:** empty project for model design.
 - **SCADE Suite Project for Design Verifier:** project (including the current model as a library) to perform formal verification with Design Verifier without polluting the original model with observer operators and proof objectives (see [“Creating Verification Project”](#) on page 736).
 - **Archive Project:** empty archive project (see [“Managing Project Archives”](#) on page 103).

- **SCADE Architect Configurator Project:** empty project for defining, managing, and generating configuration (see [“Creating Configurator Project”](#) on page 15 in *SCADE Architect Configurator Guidelines*).
 - **SCADE Architect Project:** empty project for architecture design (see [“Managing Projects”](#) on page 68 in *SCADE Architect User Manual*).
 - **SCADE Test Project:** empty project for preparing test data before testing sessions. You must install SCADE Test Environment for Host with SCADE Suite to get access to this type of project (see [“Creating Test Projects”](#) on page 11 in *SCADE Test User Manual*).
 - **SCADE Test Result Project:** empty project for storing test results from testing sessions. You must install SCADE Test Environment for Host with SCADE Suite to get access to this type of project (see [“Creating Result Projects”](#) on page 23 in *SCADE Test User Manual*).
- 3 Enter the name of the project in the **Project name** field. The **Location** field is updated automatically while entering the name.
 - 4 Click the browsing button to specify the file location in the **Location** field.
 - 5 Select **Create new workspace** to generate the project in a new workspace or **Add to current workspace** to insert it in the active workspace.
 - 6 Click **OK** to launch the SCADE Suite Project wizard.

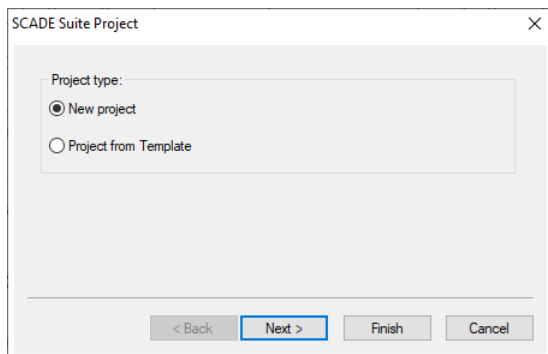


Figure 2.14: Setting project type with SCADE Suite project wizard

- 7 For SCADE Suite project creation, select between empty and template-based projects from following options:
 - **New project:** an entirely new and empty project.
 - **Project from template:** a template-based project to copy options, files, and libraries from the template file (.et t) into the new project. For details, see [“To save project as template”](#) on page 58.
- 8 Click **Next** to continue project configuration with project libraries. If you decide to create the project from a template or descriptor file, SCADE Suite opens a window where you can specify the location of the template or descriptor file to use. Click **Finish** to validate the new project.

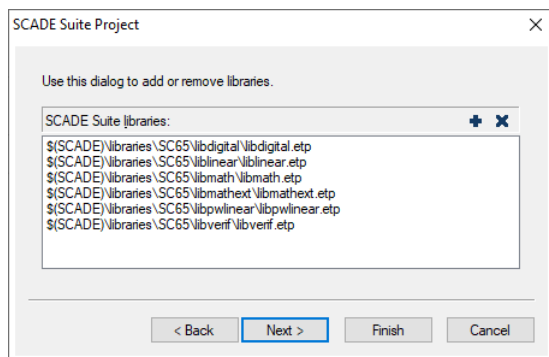


Figure 2.15: Setting libraries to be included in new project

- 9 Edit the list of libraries if needed. You can add new library files to the list or delete libraries from the default list. Relative paths and environment variables are allowed in the libraries path.
- 10 Click **Next** to continue project configuration with model properties.

Note

The properties set for the model in this wizard can always be edited later from the project properties. See [“Editing Project Properties”](#) on page 424.

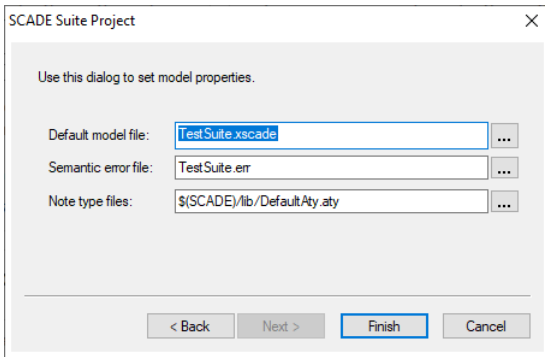


Figure 2.16: Setting model properties in the new project

- 11** In **Default model file**, specify the path of the default model file. The constants, types, and sensors declared in a model outside any package, if any, are stored in this file (see [“Multifile Management”](#) on page 66).
- 12** In **Semantic error file**, specify the path of the semantic error file. This field may contain environment variables.
- 13** In **Note type files**, specify the path of the note type file(s) . This field may contain environment variables.
- 14** Click **Finish** to create the project.

You can specify the properties of projects at any time. Project files (.etp) have editable properties accessible from the [“Scade View”](#) and the [“File View”](#). For user instructions, see [“Editing Project Properties”](#) on page 424, and [“Editing Project Files Properties”](#) on page 431.

Creating New Python Files

You create empty Python script files for writing scripts to access SCADE models and generated data. For scripts and data extraction, see [“Using Script Wizard for Python Scripting”](#) on page 28 in *SCADE Python API Guide*. For Python scripts and customizing SCADE Suite, see Chapter 4 about [“Scripting Capabilities in SCADE Environments”](#) in *SCADE Suite Technical Manual*.

To create an empty Python file

- 1 Click **New**  in the **Standard** toolbar or select *File > New*.
- 2 Display **Files** options and select **Python File** in the list of files.

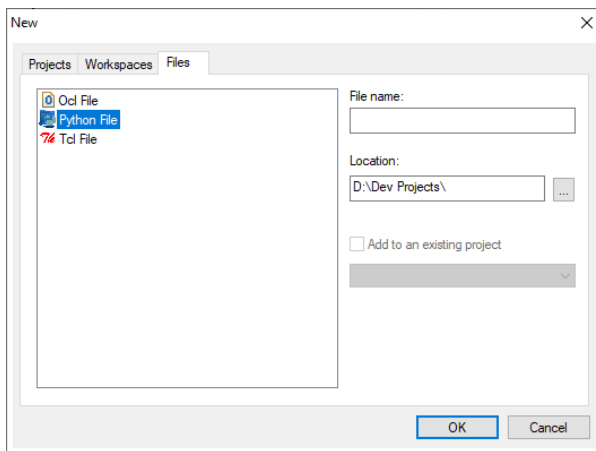


Figure 2.17: Defining empty Python file

- 3 Enter the name of the file in the **File name** field.
- 4 Browse in project folders and set the file path in the **Location** field.
- 5 Check **Add to an existing project** to insert the file into a project currently opened in SCADE Suite. Select the project from the drop-down list.
- 6 Click **OK** to create the file. SCADE Suite automatically opens the empty file where you can start writing the script.

Creating New Tcl Files

You create empty Tcl script files for customizing SCADE Suite. For scripts and data extraction, see [“Using Script Wizard for Tcl Scripting”](#) on page 169 in *SCADE Java/Tcl API Guide*. For customizing, see Chapter 4 about [“Scripting Capabilities in SCADE Environments”](#) in *SCADE Suite Technical Manual*.

To create an empty Tcl file

- 1 Click **New**  in the **Standard** toolbar or select *File > New*.
- 2 Display **Files** options and select **Tcl File** in the list of files.

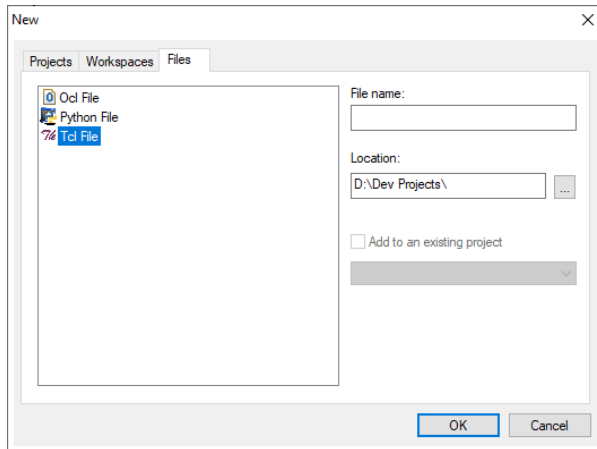


Figure 2.18: Defining empty Tcl file

- 3 Enter the name of the file in the **File name** field.
- 4 Browse in project folders and set the file path in the **Location** field.
- 5 Check **Add to an existing project** to insert the file into a project currently opened in SCADE Suite. Select the project from the drop-down list. If no project is active, you cannot insert a Tcl file in any existing project.
- 6 Click **OK** to create the file. SCADE Suite automatically opens the empty file where you can start writing the script.

Switching Active Project in Same Workspace

You can switch the active project at will when several projects are included in the same workspace. The active project appears in bold in the model browsers.

To switch the active project

- Use either of the following:
 - Select the project to activate from the file name list at *Project > Set Active Project*.
 - Right-click project file in **File View** or **Scade View** and select *Set as Active Project*.
 - Select the project to activate from a file name list available in the **Project** toolbar.

Renaming Projects

You can rename existing projects with respect to all SCADE Suite files they reference.

To rename SCADE Suite projects

- 1 Copy manually the project directory and all related files in a new location.
- 2 Open the project and select the .etp project file in **File View** or **Scade View**.
 - From **General** properties, rename the project in the **Persist As** field, press Enter.
 - Press F2 and rename the file directly in **File View**.
- 3 Save the project with its new name.

Managing Project Archives

Archive management in SCADE Architect, SCADE Suite, and SCADE Test allows to export an archive from any project, to import an archive and load its content, or to create a project dedicated to specifying an archive containing multiple projects.

- [“Exporting Projects as Archives”](#)
- [“Setting Project Archiving Properties”](#)
- [“Creating Archives from Archive Projects”](#)
- [“Importing Archives”](#)
- [“Customizing Archive Files”](#)
- [“Defining Applications for Archive Project”](#)
- [“Archive Project Command Lines”](#)

Exporting Projects as Archives

It is possible to export any SCADE Architect, SCADE Suite, or SCADE Test project with all sub-projects, libraries, referenced files, and ALM Gateway files into an archive.

To export a project into an archive

- 1 Select the project (. etp) to archive from any SCADE browser.
- 2 Select *File > Export as Archive* to produce the archive file.

A <project_name> . zsp file is created in the same directory as the selected project. By default, the archive file contains the selected project, all files referenced by the project (SCADE files, C files), as well as all needed implicit files. By default, sub-projects are only referenced in the archive and

need to be exported separately in their own archive. The default content of archives can be modified as explained in [“Setting Project Archiving Properties”](#) on page 104.

Note

Exporting archives is possible in command line using a script located in SCADE installation at `SCADE\Scripts\Archive\exportarchive.py`. For details, see [“Archive Export/Import Command Line”](#) on page 921.

Notice you can modify the content of archives by adding/removing files to/from the project using **File View**.

Only files which are located under the project folder or its sub-folders are considered for the archive file. Other files are ignored and not added to the archive file. When files are contained outside the project folder and are needed for archiving, it is possible to use an archive project as explained in [“Creating Archives from Archive Projects”](#) on page 105.

Setting Project Archiving Properties

It is possible to modify the way a file/subproject is considered in an archive by modifying the default archiving properties before exporting an archive.

To modify properties for project archiving

- 1 In **File View**, select a file (referenced file or any project file included in the active project) contained in the project and display **Archive** properties.

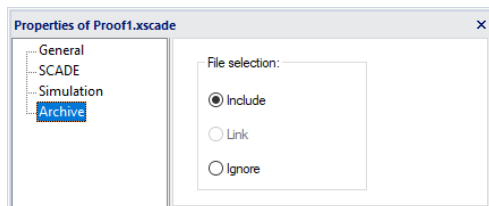


Figure 2.19: Setting properties for file archiving

2 Set **File selection** for archiving as follows:

- **Include:** the selected file (referenced or project) is added to the archive file and can be uncompressed when unzipping the archive at import. By default, all files except project (.etp) files are included in the archive. When setting project files as included, the whole content of the project defined in the .etp file is recursively included in the archive.
- **Link:** the selected file (project only) is not added to the archive but is referenced as having its own archive file. The archive of all linked projects must be done by exporting these projects separately. By default, all sub-project files contained in the archived project are specified as a link in the archive.
- **Ignore:** the selected file (referenced or project) is not added to the archive file. If a file is set as **Ignore**, a warning can be raised when importing the project, as the project cannot find the file. By default, projects and files referenced with \$(SCADE) are set as **Ignore**.

Creating Archives from Archive Projects

In case users have to archive projects and files which are not located under the project directory, in case users want to add several projects to the same archive file, or in case users want to include in the archive file additional files that are not part of the project itself (*i.e.*, documentation, etc...), it is possible to use an archive project.

To export projects into archive from an archive project

- 1 Click **New**  in **Standard** toolbar or select *File > New*.
- 2 Select **Archive Project**, name the project, set the file location and click **OK** to create the project. The file location must be in a folder such that all files to be included in the archive are located in that folder or in a sub-folder.

- 3 In **File View**, right click the archive project and select *Insert Files* to insert all project files and any other files that must be part of the archive.
- 4 In **File View**, select all projects to be included in the archive, display **Archive** properties, and set to **Include** (see [“Setting Project Archiving Properties”](#) on page 104)
- 5 In **File View**, select the project file to be used as main project by the archive, display **Archive** properties. In **Root project** list, select one of the main project’s sub-projects (only sub-projects set to **Include** are listed)

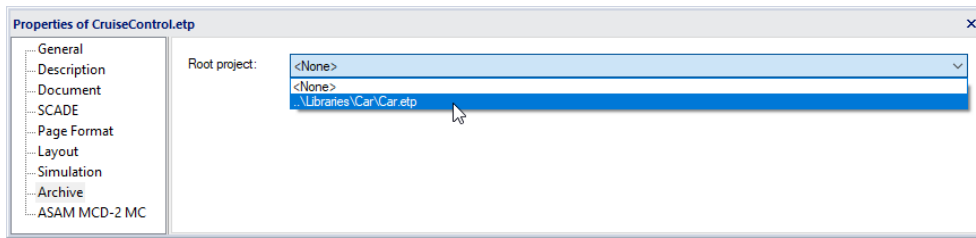


Figure 2.20: Setting the root project of an archive

- 6 Select each project in **File View** and set the properties for file archiving as needed (see [“Setting Project Archiving Properties”](#) on page 104).
- 7 Select the archive project in **File View**.
- 8 Select *File > Export as Archive* to produce the archive file.

Importing Archives

Archives created from SCADE Architect, SCADE Suite, and SCADE Test projects can be imported by users in their own SCADE environment as detailed below.

To import archive file and load content

- 1 In SCADE, select *File > Import > Archive file* to open the **Import an archive file** window.



Figure 2.21: Setting archive import

- 2 In **Archive location**, browse for files and specify the path to the archive file (.zsp) to be imported.
- 3 In **Target directory**, specify the directory where the archive content is uncompressed.
- 4 Check **Open root project** (default) to open the project set as root project in the imported archive. If unchecked or no root project is found in the archive, the ETP project with the same pathname as the selected archive file is loaded.
- 5 Click **Import**. If specified options are valid, the selected archive file is uncompressed in the specified target directory. The archive files of linked projects must be in the directory of the selected project to allow archive import; such archive files are uncompressed in the directory of the linked projects. The root project of the imported archive is loaded in SCADE.

Note

Importing archives is possible in command line using a script located in SCADE installation at SCADE\Scripts\Archive\extract.py. For details, see [“Archive Export/Import Command Line”](#) on page 921.

Customizing Archive Files

In some particular cases, it is easier to complete the content of an archive defined within an archive project using pattern matching. It is also possible to define an archive content completely without creating an archive project. Users can customize their `<project_name>.archive` file with a dedicated syntax.

The following template can be used to customize archive files:

```
# List of files to be included in the project archive
# Line format:
# <operation_char> <path_pattern>
# <operation_char> = + to add files
# <operation_char> = - to exclude files
# <path_pattern> = path specification. It can be absolute or relative but
# only files under the project directory will be added.
#
# Example 1: to include the content of directory TMP except .txt files:
# + TMP/*
# - TMP/*.txt
#
# Example 2: to include all .c files of all the subdirectories of the
# project
# + **/*.c
#
# Example 3: include files with numerical names located in the project
# directory
# + [0-9].*
```

For example the archive file can be as follows:

```
+ CruiseControl/TwinBuilder/**/*
- CruiseControl/TwinBuilder/*.png

+ **/*.docx
```

Defining Applications for Archive Project

It is possible to define applications that can be executed on archive projects using SCADE or Python command lines. Such definition is contained in an `<archive_name>.batch` file. By default, this file defines SCADE applications like TEST and COVERAGE.

The following template can be used to define applications:

```
# List of Applications that can be applied to an archive file
# An application is identified by a section defining the command line
# associated to this application
#[<application_name>]
#command={scade, python}
#args = list of command arguments

[TEST]
command=scade
args = -test "<PROJECT>" -tee -conf Test -result_project "<PROJECT>"

[COVERAGE]
command=scade
args = -test "<PROJECT>" -mc -conf Test -result_project "<PROJECT>"
```

Archive Project Command Lines

Use either of the following commands to execute an application on archive projects:

Executing application defined in `<archive_name>.batch` file:

```
<install_dir>/contrib/Python37/python <script_path>/zscade.py <archive_file> [-d <dir>] [-l <libpath>] [-u] -a <APP>
```

Executing application defined in command line:

```
<install_dir>/contrib/Python37/python <script_path>/zscade.py <archive file> [-d <dir>] [-u] <cmd> -- <args>
```

Possible command options and parameters:

- `<script_path>` to specify the path of the Python script at `install_dir/scade/scripts/`
- `<archive_file>` to specify the path of the archive project (.zsp file) containing the `<archive_name>.batch` file where applications are defined (see [“Defining Applications for Archive Project”](#) on page 109)
- `-d <dir>` (optional) to specify the directory where the archive project is unzipped (by default, the archive project is unzipped in “./WS” directory)
- `-l <libpath>` (optional) to specify the path of library archives linked to the archive. By default, this path is “.”, use `-l` option to change or set a path. It is possible to use several instances of `-l` in the command.
- `-u` (optional) to update the archive after command execution (for example if the project is updated by the command execution)
- `-a <APP>` to name the application section defined in `<project>.batch` file (TEST, COVERAGE, etc.).
- `<cmd>` to launch `scade.exe` or `python.exe` commands with arguments `<args>`. Such commands cannot be used with `-a` option All available Scade commands are described in Appendix A about [“SCADE Suite in Command Line”](#). All available Python commands are described in Python API Guide (see [“SCADE Python API Guide Overview”](#)).
- `-- <args>` to specify the list of arguments needed by the SCADE or Python command `<cmd>`

Note: `scade` or `python` sub-commands can use the generic parameter `<PROJECT>`. When the SCADE archive project defines a root project, the `<PROJECT>` parameter is replaced by the root project, otherwise it is replaced by the name of the archive project itself.

Example 1: Executing tests on an archive project

```
<install_dir>/SCADE/contrib/Python/python <install_dir>/scade/scripts/zscade.py Dev/  
ProjectA/MyArchProj.zsp -a TEST
```

```
<install_dir>/SCADE/contrib/Python37/python <install_dir>/scade/scripts/zscade.py Dev/  
ProjectA/MyArchProj.zsp -u scade -- -test "<PROJECT>" -tee -conf Test -result_project  
"<PROJECT>"
```

Example 2: Retrieving model coverage from an archive project

```
<install_dir>/SCADE/contrib/Python37/python <install_dir>/scade/scripts/zscade.py Dev/  
ProjectA/MyArchProj.zsp -a COVERAGE
```

```
<install_dir>/SCADE/contrib/Python37/python <install_dir>/scade/scripts/zscade.py Dev/  
ProjectA/MyArchProj.zsp -u scade -- -test "<PROJECT>" -mc -conf Test -result_project  
"<PROJECT>"
```

Use the following command line to perform a diff between archive files:

```
python <script_path>\zdiff.py <archive_file1> <archive_file2> [-h] [-d <directory>]
[-no_libraries] [-report <report>] [-format <format>] [-script <script>]
```

Possible command options and parameters:

- **<script_path>** to specify the path of the Python script at `install_dir/scade/scripts/`
- **<archive_file1>** and **<archive_file2>** to specify respectively the path to the first and second archive projects (.zsp file) to be compared
- **-h** (optional) to display help information about command line usage and options.
- **-d <dir>** (optional) to specify the directory where the archive projects are unzipped (by default, the archive projects are unzipped in ". /WS" directory).

Specific diff options:

- **-no_libraries** (optional) to prevent any library comparison. By default, both projects are compared including their libraries.
- **-report <report>** to specify the pathname of the generated report. By default, the report naming convention is: `<project1/object1/file1>_diff_<project2/object2/file2>.htm`. The report is located in the directory of the first project.
- **-format <format>** to set the report in RTF or HTML (default) format
- **-script <script>** to specify the script used to compute the differences and generate the report (by default, the script is located at `<install_dir>/Scripts/Diff/ScadeDiff.tcl`)

Example: Comparing two archive projects

```
<scade_python_path>/python <script_path>\zdiff.py -d Test/tmp Test/Project1.zsp Test/
Project1_Changed.zsp -report Test/tmp/report.rtf -f rtf
```

The above command produces in the specified `Test/tmp` directory:

```
P1_Project1/extracted_contents_of_Project1.zsp
P2_Project1_Changed/extracted_contents_of_Project1_Changed.zsp
```

Then, the command launches:

```
<scade_bin_path>/scade.exe -diff Test/tmp/P1_Project1_Changed/Project1.etp Test/tmp/
P2_Project1_Changed/Project1_Changed.etp [diff options]
```

and produces a report containing diff operation results in `Test/tmp/report.rtf`.

Inserting Existing Project or Files

You can insert several projects or files in the active project provided you have read-access.

- When inserting a project, the workspace is updated and the project added to its content. The added project becomes the active one (in bold in model browsers).
- When inserting SCADE files (.xscade and .scade) or any other external files like H or C-files, the project is updated and inserted files displayed in the **File View**. All objects defined in inserted files are loaded and appear in the model browsers.

To insert an existing project into the current workspace

- 1 Select *Insert > Project into Workspace*.
- 2 Browse for project files from the **Open** window. Click **Open** to insert file. In case of errors, the browsing window remains open: enter another project name or click **Cancel**.

To insert existing file(s) in the current project

- 1 Select *Insert > Files*.
- 2 Browse for files or library files from the **Open** window.
- 3 Click **Open** to insert file. In case of errors, the browsing window remains open: enter another file name or click **Cancel**.

Messages about loaded projects or files display in the **Messages** output tab.

Blocking errors can be due to the following situations:

Added project/file/library has same name as another project/file/library in the workspace.

Added project/library contains a file which is already in the model.

Declaration conflicts among entities of the added project/file or between an entity of the project/file and an entity of the current model or library.

Syntactical error detected in added project (one file), in added file or library (one file).

One of the new library files does not exist.

No user read-access to one of the new library files.

Detection of recursions in project (e.g., library A uses library B which uses library A).

Non-blocking errors display as warning messages and due to the following situations:

You do not have write-access to the project or file.

You do not have write-access to one of the new project or library files.

An annotation file cannot be read-accessed.

An annotation file is empty.

One annotation file is syntactically incorrect. Its annotations are discarded.

An annotation type is unknown. The annotation is not loaded.

Annotation is attached to unknown entity. It is not loaded.

The semantic error file cannot be write-accessed.

References to missing elements are not resolved.

The following errors may occur only when inserting projects:

Annotation type file is empty.

Annotation type file is syntactically incorrect and remains unloaded. Annotation files are discarded as well.

Creating New Folders in Projects

You can create new virtual folders to organize project files in the **File View**. This allows you to manage the content of the projects as needed. For details about folder configuration, see [“Editing File Folder Properties”](#) on page 430.

To create new folders in File View

- 1 Right-click a project or an existing folder in **File View** and select *Insert Folder*.

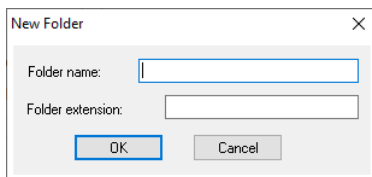


Figure 2.22: Creating new folder in File View

- 2 In **New Folder** window, enter the name of the folder in **Folder name**.
- 3 Enter the file extension(s) in **Folder extension** to help classifying automatically files added into a project without prior indication of the folder. Use semi-colon (;) to list several extensions.
- 4 Click **OK** to validate. A folder or sub-folder is created in **File View**.

Saving Files in UNIX Text Format

You can save SCADE Suite files in UNIX mode without using the extra CTRL-M at the end of each line. This is useful if you need SCADE Suite files in a UNIX environment.

To enable file saving in UNIX

- Either set the environment variable **ETBINARY** in Windows;
- or start SCADE Suite Editor with the option `-binary`.

Managing Libraries

SCADE Suite enables to manage libraries in projects to facilitate the reuse of operators and design definitions across multiple projects as described below.

- [“Inserting Library in Projects”](#)
- [“Replacing Libraries in Projects”](#)
- [“Setting Library Path for Sharing Projects”](#)

Inserting Library in Projects

You can create own libraries or reuse common operators and types that are defined in SCADE Suite libraries. To reuse the content of libraries in models, you must reference the library in the **File View** of any project opened in SCADE Suite Editor. Libraries are SCADE Suite projects, reused in read-only mode in the context of another project. They can contain type definitions, constant definitions, and operators. The content of inserted libraries is available in read-only mode and ready to be instantiated in the model.

To insert a library in projects

- 1 Display the **File View** of the current project.
- 2 Right-click the SCADE Libraries folder (or any other custom folder) and select *Insert Files*. The **Open** window opens.
- 3 Browse for files in local directories and select the library file in ETP format.
- 4 Click **Open** to insert the file in current project. The ETP file appears in **File View** and all resources of the inserted library are available for model designing in **Scade View**.

Replacing Libraries in Projects

It is possible to replace any operator reference to a library with the same operator reference(s) contained in another library provided the new library contains the operators configured with the same name. **Deleting a library does not delete its references.**

Setting Library Path for Sharing Projects

Whenever you insert a library from SCADE Suite distribution in projects, the library file path refers to a relative pathname and the `$(SCADE)` variable. This environment variable references the absolute pathname to the installation directory of the SCADE Suite version currently running. This allows SCADE Suite projects to be shared between users whether or not they have different installation directories.

Tip

If libraries inserted in a project are in locations that vary with users, one can define an environment variable (*i.e.*, `SCADELIB`) for each user and set how the project keeps a persistent link towards such libraries using this same variable. For instance, set `SCADELIB=C:\users\all\scade\lib` environment variable with the operating system and replace the absolute reference to `C:\users\all\scade\lib\mylib.etp` by `$(SCADELIB)\mylib.etp` in the Persist as field (see how [“To edit the general properties of a file”](#) on page 431).

Managing Default or Custom Configurations

Configurations provide a quick way either to enable the settings associated with some SCADE tools or to change the display of selected options and parameters when editing tool settings. They are a set of pre-selected settings applicable to the following tools:

- SCADE LifeCycle Reporter
- SCADE Suite Timing and Stack Analysis Tools
- SCADE Architect Synchronization
- SCADE Suite KCG/MCG Code Generators
- SCADE Suite Simulation
- SCADE Test Model Coverage

SCADE Suite is installed with a few default configurations. Each new project is generated with [default configurations](#). Additionally, you can create custom ones in models and enable them like the default configurations when required by tool usage.

- [“Enabling Configurations”](#)
- [“Editing Default and Custom Configurations”](#)
- [“Creating and Setting Configurations”](#)

Default Configurations

Default configurations are associated with SCADE Suite tools that may require specific settings for SCADE environment and development purposes. All configurations can be edited as described in [“Editing Default and Custom Configurations”](#) on page 123. Switching default configurations and applying pre-selected settings becomes easy. They facilitate the following usage:

- [“Generating Project Documentation”](#) on page 40 in *SCADE LifeCycle User Manual*
- [“Launching Testing Sessions on Host”](#) on page 32 in *SCADE Test User Manual*
- [“Launching Simulation Session”](#) on page 596
- [“Launching Timing and Stack Analysis Tools”](#) on page 809
- [“Analyzing Model Coverage Results \(Suite\)”](#) on page 155 in *SCADE Test User Manual*
- [“Launching KCG Code Generation from UI”](#) on page 864

The default configurations available at SCADE Suite model creation can be used in various context as described in the next table.

Table 2.2: Default configurations available in SCADE Suite

For report generation using SCADE LifeCycle Reporter	
RTF	This configuration enables the generation of reports in RTF format with respect to report customization options applicable to this format.
HTML	This configuration enables the generation of reports in HTML format with respect to report customization options applicable to this format.
Cert. Reporter	This configuration enables the generation of qualified reports in RTF format with respect to report customization options compliant for certification. For details, see “Documentation Generation for Certification” on page 40 in <i>SCADE LifeCycle User Manual</i> .
For model synchronization with SCADE Architect	
Synchronization	This configuration enables synchronization of SCADE Suite models with SCADE Architect models.
For simulation code generation using SCADE Suite Code Generator	
Simulation	This configuration enables model simulation and debugging by enhancing the visibility of variables in simulation without optimizing the code. For user guidelines, see “Setting Simulation Code for Simulation Sessions” on page 586.
For timing and stack analysis using SCADE Suite Code Generator	
Timing and Stack	This configuration enables the generation of code for timing and stack usage analysis. For user guidelines, see Chapter 17 about “Verifying Timing and Stack Usage” .

Table 2.2: Default configurations available in SCADE Suite (Continued)

For embedded code generation using SCADE Suite Code Generator	
KCG	This configuration enables the generation of code to embed with Code Generators. Note: To generate qualified code, you must generate code using SCADE Suite KCG in batch commands. Contact SCADE Products Support for more details about the entire SCADE Suite qualification process.
For coverage measurements using SCADE Test Model Coverage	
Coverage	This configuration enables instrumentation for model coverage measurement. For user guidelines, see “Setting Test Tool for Test Execution on Host” on page 28 in <i>SCADE Test User Manual</i> . It also enables the generation of reports with options applicable to coverage data reporting. For details, see Chapter 5 about “Reporting about Test Data” in <i>SCADE LifeCycle User Manual</i> .

Enabling Configurations

You activate configurations available in project by menu commands or directly from their corresponding toolbar. Once selected, the configuration automatically enables the tool options editable from the **Settings** window of the tool.

Warning

Selecting configuration from drop-down list in the upper right corner of Settings windows only allows to edit the configuration. It does not make it active at next tool execution.

To enable a configuration from toolbars

- 1 Display the appropriate toolbar from *View > Toolbars* or locate it in the workspace.

- 2 Select any default or custom configuration name in the drop-down list. Custom configurations need to be defined first in the project as described in [“Creating and Setting Configurations”](#) on page 121.

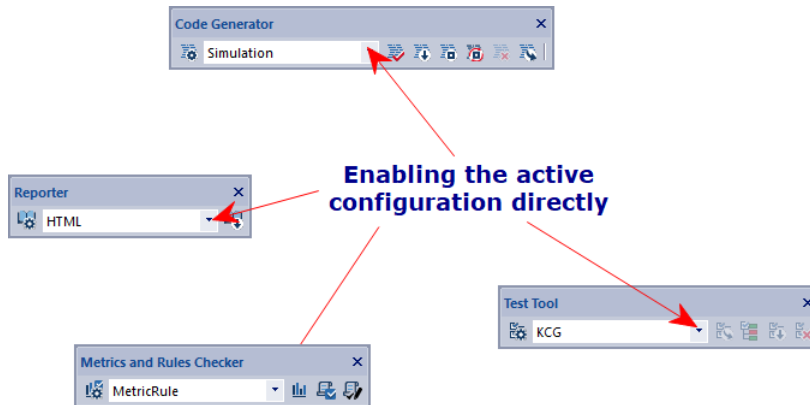


Figure 2.23: Selecting the active configuration from any toolbar dedicated to SCADE tools

To enable a configuration from Project menu

- 1 Select **Project > [Tool Name] > Set Active Configuration** to open the **Set Active Configuration** window.
- 2 Select the configuration to activate from the list and click **Set Active**. It turns to bold.
- 3 Click **OK**. The toolbar of the corresponding tool is updated.

Creating and Setting Configurations

Custom configurations contain preselected tool parameters or parameters copied from default configurations. Each custom configuration is associated with specified tool(s) and attached to a specific project.

To create custom configurations

- 1 Select **Project > Configurations**.

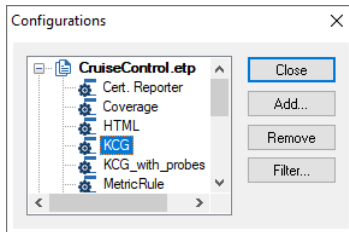


Figure 2.24: Opening Configurations window

- 2 Click **Add** to open the **Add Configuration** window. Instead, to delete a configuration from the project, select it in the list and click **Remove**.

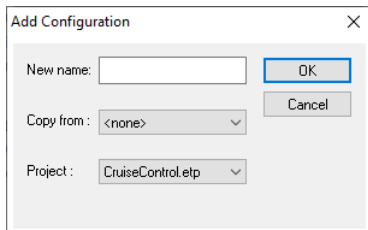


Figure 2.25: Adding new configuration

- 3 Enter the name of the configuration in **New name**.
- 4 Select a configuration in the **Copy from** list to initialize the new configuration with tool settings already set or leave the selection on <none> to use the default tool settings.

- 5 Select the project to which the configuration is attached in the **Project** list.
- 6 Click **OK** to create the new configuration.

To associate configurations with tools

- 1 Select *Project > Configurations*.

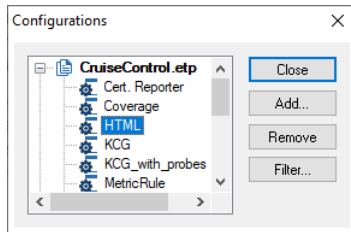


Figure 2.26: Selecting configuration to be associated

- 2 Select a configuration and click **Filter** to open the **Configurations Filter** window.

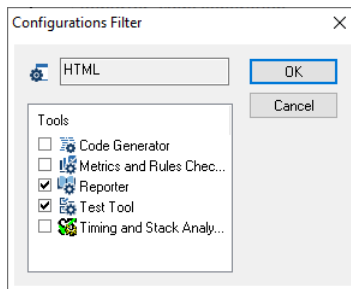


Figure 2.27: Associating configuration with SCADE Suite tool

- 3 Check/uncheck the tools to be associated or not with the configuration and click **OK**.

Editing Default and Custom Configurations

You can always edit the parameters set for any custom or default configuration.

To edit tool parameters in configurations

- 1 Open the **Settings** window of any tool associated with a configuration.

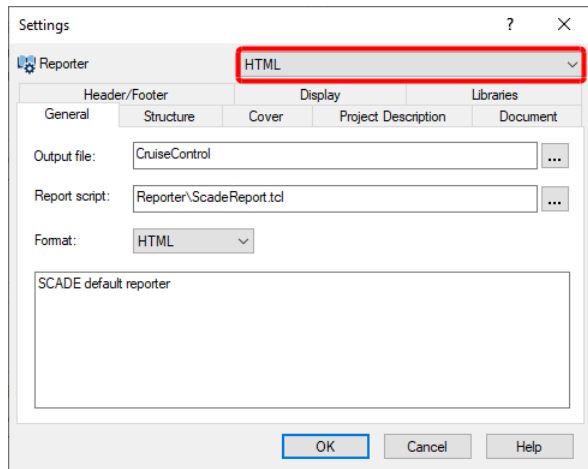


Figure 2.28: Setting configuration parameters with Reporter tool

Warning

The configuration drop-down list does not activate configurations.

- 2 Select the configuration to edit in the configuration list in upper right corner of the **Settings** window. The list contains all configurations associated with the tool.
- 3 Set the tool parameters as appropriate and click **OK** to save the edited configuration.

Organizing Model Design with Packages

Packages support the organization of the software design as a bundle of separate blocks containing model declarations. They can contain elements like operators, constants, sensors, types, or sub-packages. SCADE Suite models can contain as many packages as needed. Packages correspond to *namespaces*.

Each package identifies a namespace where one can group model elements together. Different design objects can be declared with the same name in separate packages. The properties of packages enable to extend the scope of visibility from a current package onto the definitions contained in any other package. It is possible to open namespaces and integrate them into the current namespace. SCADE Suite Editor ensures that there are no scope conflicts when opening the namespace of packages: a model set with the definition of a type T cannot open a package where a constant is called T as well.

In addition, you may modify the default visibility of any operator declaration outside their package. Based on package scope management, SCADE Suite Editor is able to compute the access name of objects used in the design and identify object definitions by their access path. Named objects are accessed through their path and can be displayed with this information where their definition is used in the model.

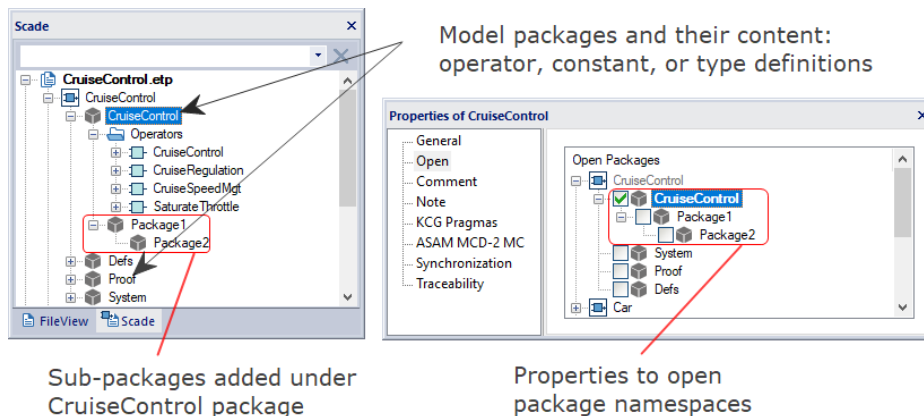


Figure 2.29: Model packages in the Scade View and package scope properties

Model packages are displayed in **Scade View** as illustrated above. Depending where they are created, they are inserted right under the model or right under a parent package.

- [“Creating New Packages”](#)
- [“Managing Visibility of Package Declarations”](#)
- [“Opening Packages”](#)

Creating New Packages

You can create new packages or sub-packages from the **Scade View** to organize the architecture of the application design and group operators as needed. Each new package allows to declare operators, constants, sensors, and types within their own namespace, or to subdivide packages using sub-

packages. You may also declare operators, constants, types, and sensors directly under the project without using packages as shown in [“Multifile Management”](#) on page 66.

Warning

Package constraints in Ada and Scade being different, code generation in Ada requires to pay attention to package structure. Please refer to [“Direct Package Hierarchy Correspondence”](#) on page 232 in *SCADE Suite Technical Manual*.

- [“To declare new packages”](#)
- [“To declare sub-packages”](#)

To declare new packages

- 1 Display **Scade View** and expand existing folders.
- 2 Right-click the project folder where to declare a package, and select *Insert > Package*. Alternatively, select a project folder and click **New Package**  in **Create** toolbar to create a package.

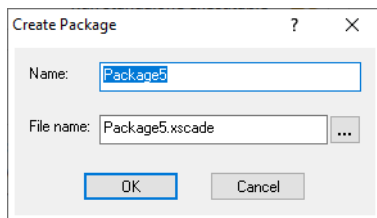


Figure 2.30: Naming newly created package

Note

SCADE prompts you for package and file name if the Use a creation dialog box option is enabled (see [“Selecting General Options”](#) on page 77).

- 3 Enter the name of the package in **Name**. The file name is updated automatically.
- 4 Click **OK** to create the package.

To declare sub-packages

- 1 Display the **Scade View** and expand existing folders.

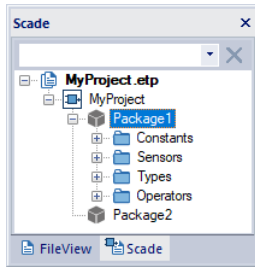


Figure 2.31: Selecting package folder before declaring sub-packages

- 2 Right-click the package folder where you want to declare the sub-package, and select *Insert > Package*.

Tip

Select the package folder (or any of its declaration folders) and click New Package  in the Create toolbar to create a sub-package.

- 3 Rename the sub-package as required.

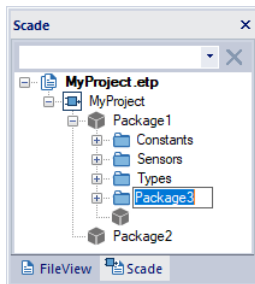


Figure 2.32: New sub-package for declaring elements in a sub-namespaces

Opening Packages

You can open packages to facilitate the usage of package declarations in SCAD Suite models and reduce the length of declaration path names from top-level package to sub-packages. When opening a package, SCAD Suite applies a directive to the package that allows to use declarations belonging to the package as if declared locally. In addition, the complete name with package access path (e.g., `Package1::Package2::OperatorA`) is not displayed.

Note

The public declarations contained in an opened package can be accessed directly. For user guidelines about public declarations, see [“Managing Visibility of Package Declarations”](#) on page 130.

For more technical details about packages and the open directive, consult Chapter 3 about [“Program and Packages”](#) in *Scade Language Reference Manual*.

To open a package

- 1 Right-click a package in **Scade View** and select *Properties*.
- 2 In **Properties** window, select **Open** and set required properties.

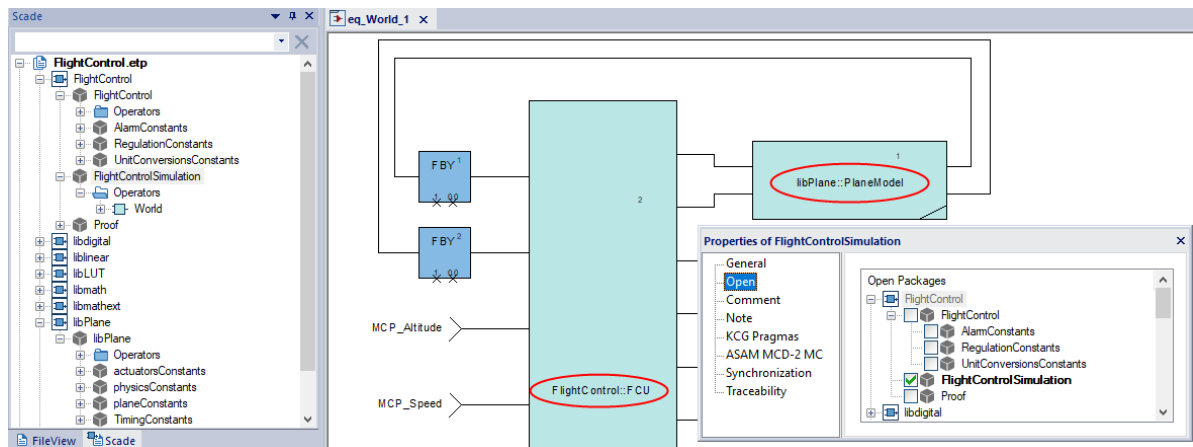


Figure 2.33: Opening packages

- 3 In the **Open Packages** tree, select the name of the package with which you want to associate the current package. Click anywhere in the interface to validate the selection.

Warning

Packages cannot be opened if this action results in name collision.

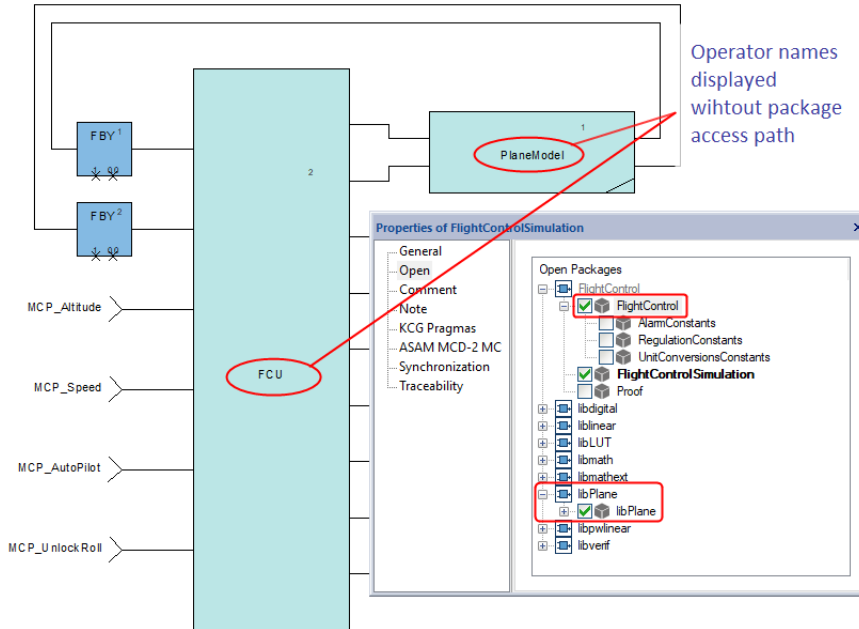


Figure 2.34: Opened package access path

Note

Opening a package is not transitive, but is inherited by sub-packages. In the above example, the namespace of Package2 is opened into the namespace of Package1, but the reverse is not true, unless you enable the open clause from the Package2 properties.

Managing Visibility of Package Declarations

You can change the visibility of any declaration outside the package that contains it. By default, all the declarations of a package are visible outside this package through the package access path notation. Changing the visibility status applies to **operators**, **constants**, and **types**.

Note

Read more about packages in Chapter 7 about [“Structuring Models”](#) of the *Scade Language Primer*.

To edit visibility rules of declarations outside packages

- 1 Right-click a declaration in **Scade View** and select *Properties*.

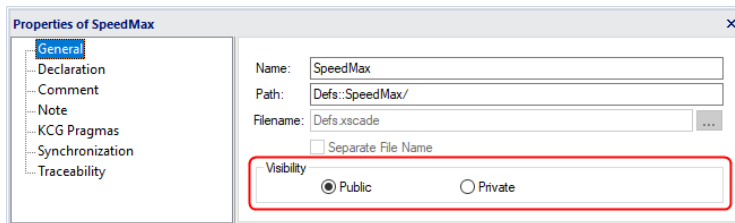


Figure 2.35: Editing visibility status of a declaration

- 2 Select **General** on the left and set required properties on the right.
- 3 Select the **Visibility** radio button as needed:
 - **Public:** the declaration can be used in any namespace (default).
 - **Private:** the declaration can be used only inside the package and sub-package that contains it and does not exist in the rest of the model.

Note

Private declarations are recognizable in the Scade View by a little key symbol on top of the package, operator, constant, or type icons.

- 4 Close the window to validate or click anywhere.

Managing Favorites and Using Shortcuts

You can create shortcuts for favorite toolbars usage. It is possible to manage the list of favorites that can be used from the **Shortcuts Window**.

To add toolbars in Shortcuts Window

- Right-click on the toolbar to be added to the **Shortcuts Window** and select *To Shortcut*. Toolbar buttons appear inside a new shortcut folder named as the toolbar.

To remove toolbars from Shortcuts Window

- Right-click a shortcut folder and select *As Toolbar*. The folder is removed from the **Shortcuts Window** and the toolbar appears back in toolbar area.

To add user-defined operators in Shortcuts Window

- 1 Right-click one or several operators in **Scade View** and select *Add to favorites*.

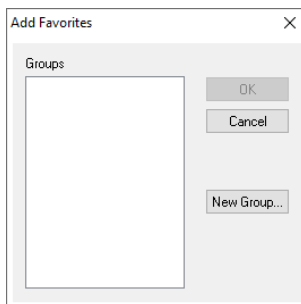


Figure 2.36: Add Favorites dialog box

- 2 In **Add Favorites** dialog, click **New Group**, enter a name for the group or select an existing one in the **Groups** list, then click **OK** to validate.

The icon and name of the operator(s) previously selected appear in the **Shortcuts Window** under the specified favorites group.

Tip

Right-click on Operators or Libraries folder in Scade View and select *Add to favorites*, to insert the complete content of the folders in the Shortcuts Window.

To remove user-defined operators from Shortcuts Window

- 1 Select *Tools > Organize Favorites* to open the **Organize Favorites** window.
- 2 In the Groups list, double click the shortcut folder that contains the operator(s) you want to remove from the **Shortcuts Window**.

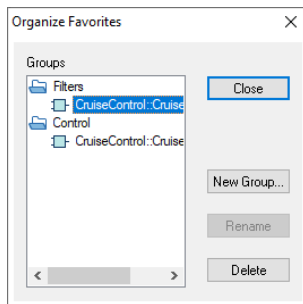


Figure 2.37: Organize Favorites window

- 3 Select the operator or a folder (to remove all favorites it contains), and click **Delete**. **Close** the window to validate.

To customize icons in Shortcuts Window

- 1 Right-click inside the current shortcut folder and select either:
 - *Small Icons*: to display favorites with small buttons.
 - *Large Icons*: to display favorites with large buttons.

To create design elements from Shortcuts Window

- 1 Select *View > Docking Windows > Shortcuts* to display the **Shortcuts Window**.
- 2 Click on shortcut folder corresponding to [predefined operator toolbars](#) or **Create** toolbar.
- 3 Create elements in the active **Diagram View** using one of the following means:
 - Click and drag predefined operator icon over the diagram, then drop it in the diagram.
 - Click any design element icon and click again in the diagram.

Creating and Modifying Styles

You can modify the general layout and display style of design elements using predefined styles or create custom styles. Each style defines font properties, line properties, and fill colors applicable to the graphical display of specific items (*i.e.*, lines, names, instance names, hidden parameters, pin ranks, pin names) on design element symbols. A style is defined by a name, the name of its parent style, and display properties. New styles inherit values from selected parent style. The inheritance mechanism is dynamic: any change in parent styles is applied to all child styles.

Table 2.3: List of predefined styles

Style name	Effect
Action	Applies line and color fill properties to action symbols on If and When block branches
Activation Block	Applies font, line, and color fill properties to State Machines, If and When Blocks
Assertion	Applies font and line properties to assume and guarantee elements
Branch	Applies font, line, and color fill properties to conditional branches in If and When block
Edge	Applies font and line properties to connecting wires between data-flow elements
Equation Set	Applies font, line, and color fill properties to equation set elements
Operator	Applies font, line, and color fill properties to user operator symbols
Predefined Op.	Applies font, line, and color fill properties to predefined operator symbols
Signal	Applies font, line, and color fill properties to signal symbols
State	Applies font and fill color properties to state symbols

Table 2.3: List of predefined styles (Continued)

Style name	Effect
Transition	Applies font and line properties to state transitions and actions in State Machines
Variable	Applies font, line, and color fill properties to I/O, probes, local variables, or expressions

Possible tasks to set the layout and display style of design elements are:

- [“Creating User-Defined Styles”](#)
- [“Applying Styles in Diagrams”](#)
- [“Modifying Style Definitions”](#)
- [“Deleting Styles”](#)

Creating User-Defined Styles

You can complete the set of predefined styles by configuring one’s own styles to highlight selected design element symbol(s) in **Diagram Views**.

To create user-defined styles

- 1 Select *Edit > Style* to open the **Style** window.

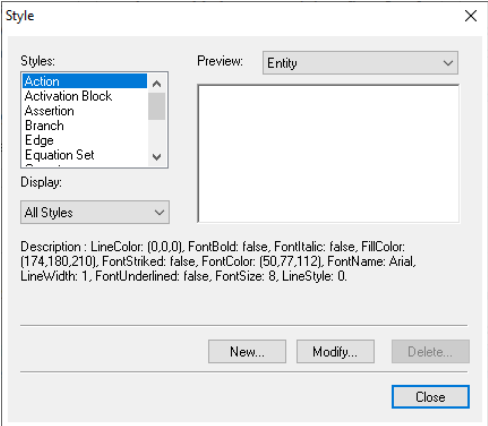


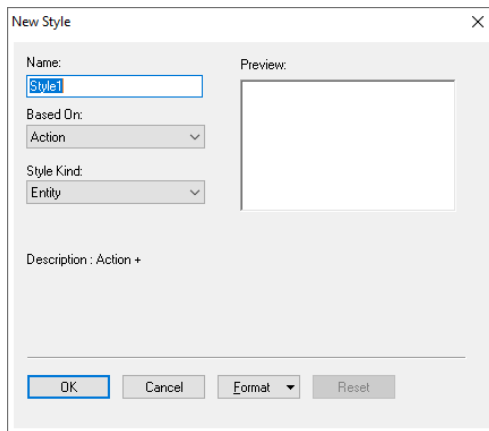
Figure 2.38: Creating user-defined styles from Style window

- 2 If needed, in **Display** drop-down, filter the list of styles by:
 - **All Styles**: to list all styles.
 - **User Styles**: to list only user-defined styles.
- 3 In **Styles** list, select the style you want to inherit from. The values describing the parent style are displayed below.

Tip

The Preview list allows you to display the style description for each category of display objects that can be defined. See step 7 below.

- 4 Click **New** to open the **New Style** window.



The 'New Style' dialog box is shown. It has a title bar with a close button. Inside, there are four main sections: 'Name' with a text input field containing 'Style1'; 'Based On' with a dropdown menu showing 'Action'; 'Style Kind' with a dropdown menu showing 'Entity'; and 'Preview' which is a large empty rectangular box. Below these is a 'Description' field containing 'Action +'. At the bottom, there are four buttons: 'OK', 'Cancel', 'Format' (with a small dropdown arrow), and 'Reset'.

Figure 2.39: New Style window

- 5 Enter the name of the new style in **Name**. The name must be different from all existing styles (case insensitive).
- 6 Set the parent style you inherit from in **Based On** drop-down. Any predefined or user-defined style can be the parent of another style (see [Table 2.3](#) about predefined styles).
- 7 Set the category of symbol items to be defined from **Style Kind** drop-down:

- **Entity:** to define the style for symbol lines, element names and labels (*i.e.*, internal variables on edges, action on transitions), or symbol fill (if any) used to display any element except higher-order constructs or conditional If and When blocks.
- **Instance name:** to define the style for instance names and labels displayed on user-defined and predefined operator symbols.
- **Assign:** to define the style for hidden parameter values displayed on some predefined operators and usable on user-defined operators.
- **PinRank:** to define the style for pin ranks or names displayed on operator symbols.
- **Priority:** to define the style for transition priorities (number in circle) in State Machines.
- **Action:** to define the style for action description on transitions in State Machines.
- **Activation:** to define the style for symbol lines, element names and labels (*i.e.*, iteration size, values on branches), or symbol fill used to display higher order constructs or conditional If and When blocks.

8 Click **Format** and select *Font* to open the **Font** window.

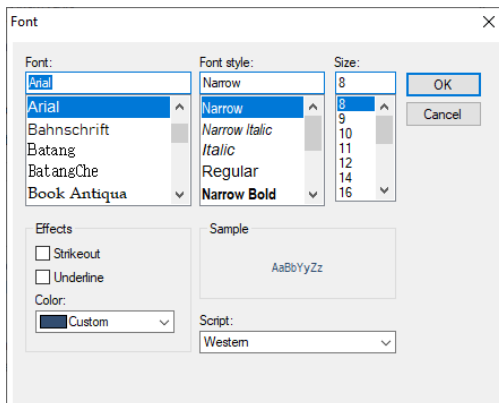


Figure 2.40: Setting font properties

- 9 Set the properties of the font applicable to the selected category of symbol items.
- 10 Click **OK** to validate font properties.
- 11 Click **Format** and select *Line and Fill* to open the **Line and Fill** window.

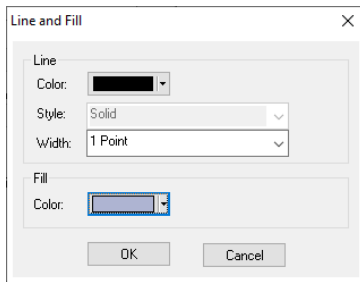


Figure 2.41: Setting line and fill properties

- 12 Set the line properties (set line width to 0 to be able to change line style) and fill color applicable to the selected category of elements.
- 13 Click **OK** to validate properties. Repeat from step 7 to set specific style properties for each category of symbol items.
- 14 Click **OK** to validate the style, choose to store the style in new option file (**Yes**) or in default option file (**No**), then click **Close** to close the **Style** window.

Applying Styles in Diagrams

You apply predefined or user-defined styles to design element symbols displayed in **Diagram Views**. It is possible to apply styles to all symbols representing a category of design elements across all diagrams or to any selected symbol provided you have created a specific style applicable to its category.

The categories of design elements that can be associated with any style are:

- Constant and signal references
- Input/Output references
- Hidden input references
- Local variable and probe references
- Connection labels and edges
- Predefined operator calls
- Operator calls
- Assumes/Guarantees
- Textual expressions
- Conditional blocks, nodes, and actions
- State Machines, states, and transitions
- Equation sets

To apply and modify styles in diagrams

- 1 Open a diagram and select *View > Toolbars > Style* to display the **Style** toolbar.
- 2 Select one or several symbols from one design element category in the **Diagram View**.
- 3 Select a style in **Style** drop-down from the **Style** toolbar to apply style properties to the selected symbol(s).

Tip

You can modify predefined and user-defined styles applied to a selection of symbols from Properties. For details, see [“Changing Design Element Layout”](#) on page 482.

- 4 Modify only some of the style properties on the selected symbol(s) from the **Style** toolbar using the following possibilities (if applicable to the selection):
 - Change to a new font in **Font** drop-down
 - Change to a new font size in **Font Size** drop-down
 - Set the font style as **Bold**, **Italic**, or **Underlined** by clicking the appropriate button
 - Change to a new font color in **Font Color** drop-down
 - Change to a new line color in **Line Color** drop-down

- Change the line width in **Line Width** drop-down
- Change to a new line style in **Line Style** drop-down (if line width set to 0)
- Change to a new fill color in **Fill Color** drop-down

Style selection applies immediately to all selected instances as illustrated in [Figure 2.42](#).

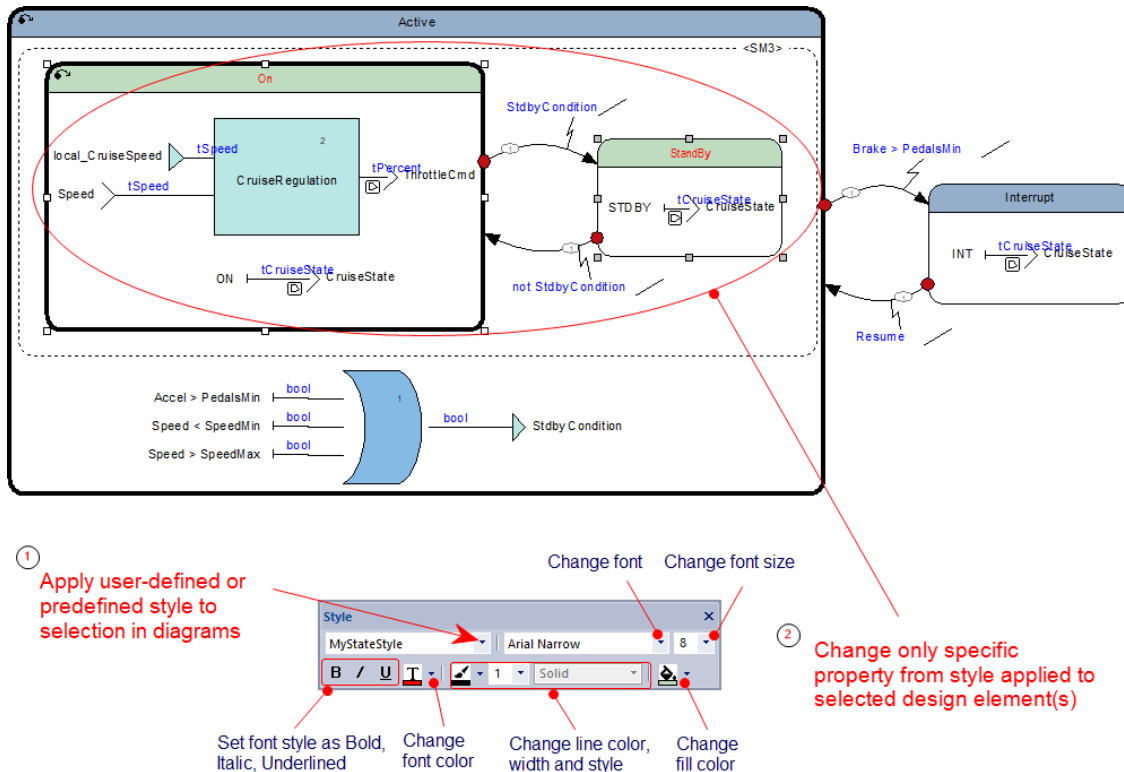


Figure 2.42: Applying style to design elements of the same category selected in diagrams

For semantic reasons, some style properties are not applicable to some selected elements.

Modifying Style Definitions

You can modify the style definitions of both predefined and user-defined styles. The layout of design element symbols associated with a modified style is updated accordingly.

Tip

If you select a design element in a Diagram View, you can modify the style for the display of the design element name via the Style toolbar and apply changes to all design elements of the same category. See [“Applying Styles in Diagrams”](#) on page 138.

To modify style definitions

- 1 Select *Edit* > *Style* to open the **Style** window.
- 2 Select the style to modify from **Styles** list and click **Modify** to open the **Modify Style** window.

Warning

When editing a design element in Symbol Editor (see [“Editing and Inserting Text in Symbols”](#) on page 299) and setting new font values for names, instance names, and hidden parameters, any style modification except for colors is ignored.

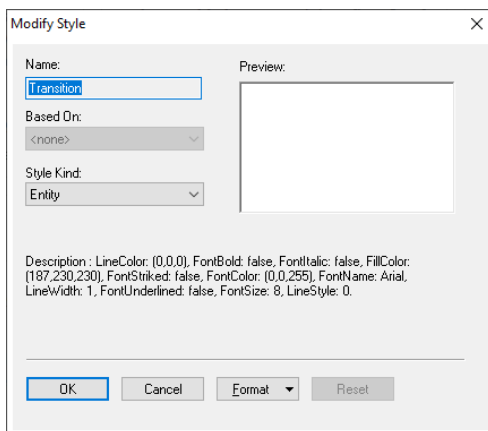


Figure 2.43: Modifying the definition of a layout style

- 3 Edit the category of elements from **Style Kind** drop-down.
- 4 Click **Format** and select *Font* to edit font properties on the **Font** window ([Figure 2.40](#)).
- 5 Click **Format** and select *Line and Fill* to edit line and fill properties on the **Line and Fill** window ([Figure 2.41](#)).
- 6 Click **OK** to save settings, then click **OK** to enable the style modifications. **Reset** to suppress all definitions set by category of object and redefine them.
- 7 Click **Close** to close the **Style** window.

Deleting Styles

The predefined styles provided at installation cannot be deleted from projects. When you delete a user-defined style, the symbols of the design elements associated with the deleted style are updated according to the properties defining the parent style of the deleted one.

To delete user-defined styles

- 1 Select *Edit > Style* to open the **Style** window.
- 2 In **Styles** list, select the style to delete.
- 3 Click **Delete** and answer **Yes** to confirm deletion of the selected style.

Enabling Display Options in Diagram Views

While working on model design in **Diagram Views**, you may need to look at various information like local and internal variables/types on connections, or pins and notes on operator calls. You can enable display options on the active view, but closing the active view cancels the last option selection.

To enable display options

- 1 Right-click in the **Diagram View** of an operator and select *Display Options*.
- 2 Check/uncheck the following options to show/hide information:
 - **Display Local Variables:** name of local variables.
 - **Display Internal Variables:** internal name of connections.
 - **Display Local Types:** type of local variables.
 - **Display Internal Types:** internal type used on connections.
 - **Display Pin Ranks:** rank of each operator call pin.
 - **Display Pin Names:** name of inputs/outputs for each operator call pin.
 - **Display Hidden Notes:** “HiddenGraphical” type of notes with text in tooltips only.
 - **Display Border:** borders of the embedded design area when editing a state or an action in hidden display.

Note

For details, see [“Switching Embedded/Hidden State Display”](#) on page 200 and [“Switching Embedded/Hidden Action Display”](#) on page 201.

You can also set default display in SCADE Suite environment, see [“Selecting Views Options”](#) on page 79.

Organizing and Working with Model Editing Views

All views (Diagrams, Constants, Types, Sensors, Textual Diagrams) you open in SCADE Suite offer a wide range of possibilities for organizing and facilitating model editing. You can adjust the layout of tabbed views and switch option display, open several instances of the same view, close views selectively or at once, grab and move design in diagrams, zoom in/ out in diagram content, use full screen display, or enable grids in diagrams.

- [“Managing Views in Multi-Tab Display”](#)
- [“Setting Model Editing Views for Multi-Screen Display”](#)
- [“Duplicating Tabbed Views”](#)
- [“Closing Tabbed Views”](#)
- [“Grabbing and Moving Design Representation”](#)
- [“Zooming Design Content in Views”](#)
- [“Underlining Truncated Text in Views”](#)
- [“Displaying Active View on Full Screen”](#)
- [“Setting Grid Properties to Assist Design Positioning”](#)
- [“Printing Active Views”](#)

Managing Views in Multi-Tab Display

In SCADE Suite, the [Model Editing Views](#) open as tabs displaying content across the entire view area. If the list of tabs does not allow to see all opened views, use top-right arrows as illustrated below. In addition, it is possible to organize tabbed views as follows:

- Splitting tabbed views into vertical or horizontal groups
- Switching tabbed views between groups

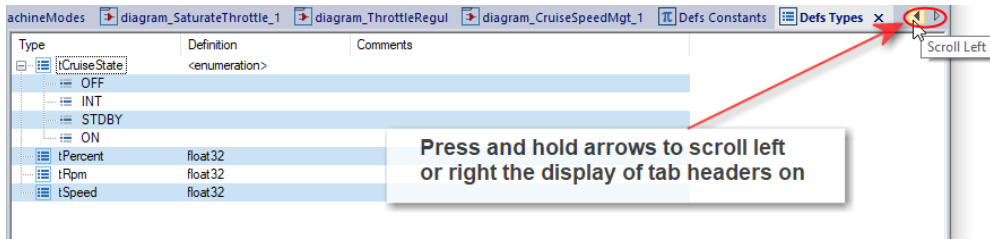
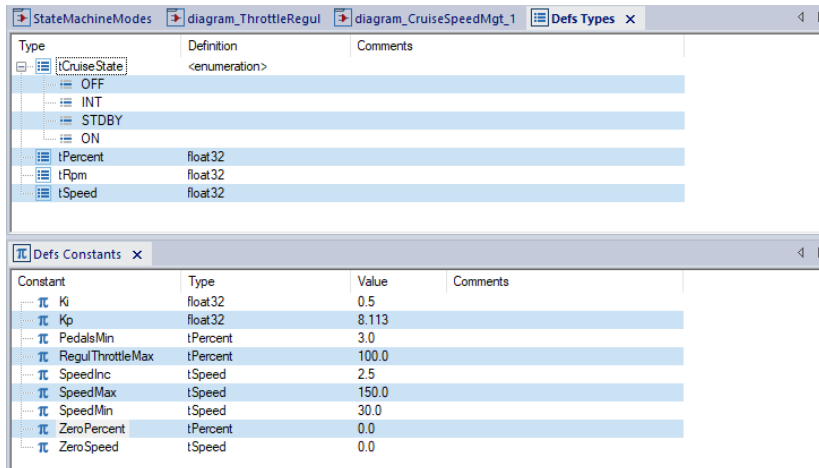


Figure 2.44: Standard display of model editing views as tabs

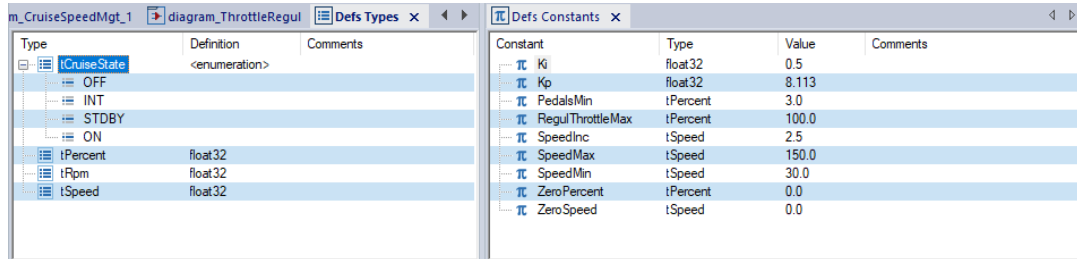
To split tabbed views horizontally/vertically

1 Select and hold the tab to be split:

- To create a tabbed view with horizontal split, move the selected tab to the bottom side of the view area to display a frame outline and release to drop the tabbed view in a new tab group.



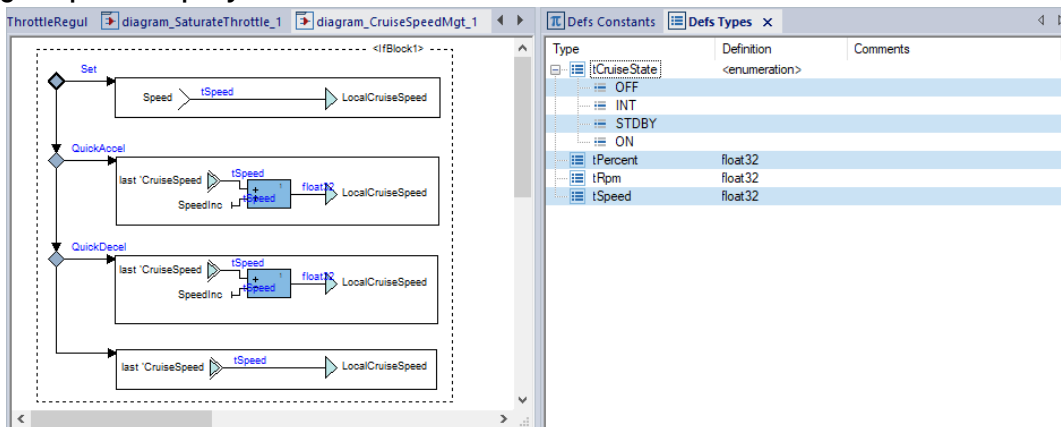
- To create a tabbed view with vertical split, move the selected tab to the right side of the view area to display a frame outline and release to drop the tabbed view in a new tab group.



All of the above can be performed contextually from tabs. Right-click any tab and select *New Horizontal Tab Group* or *New Vertical Tab Group* (if no tab group). Mixing vertical and horizontal tab groups is not possible.

To switch tabbed views between tab groups

- 1 Select and hold the tab to be moved to another group.
- 2 Move the selected tab unto the tab header area (or tab) of the destination group to display a frame outline and release the tab.



All of the above can be performed contextually from tabs. Right-click any tab and select *Move to Next Tab Group* or *Move to Previous Tab Group* (if tab group exists).

Setting Model Editing Views for Multi-Screen Display

All model editing views and file views (displayed as non dockable tabs) can be opened as floating windows outside SCADE Suite application and moved for display on multi-screen environments.

To set views as floating windows for multi-screen usage

- Right-click the tab of an opened diagram or file and select *Float*, or grab the tab of an opened diagram or file and drop it outside its display area. The selected view now displays as a floating view which can be moved onto another screen. Note it is possible to grab and drop other tabs on the tab bar area of any floating window.

To set floating windows back to application window

- Right-click any tab displayed in a floating window and select *Back to Main*, or grab any floating view (or tab in a floating view) and drop it in the main application window on the tab bar area (if any view opened) or on the empty display area of editing views (if no view opened). The dropping area is highlighted in blue.

Duplicating Tabbed Views

You can duplicate any active view representing model design and definitions.

To duplicate tabbed views

- 1 Set the tabbed view to duplicate as the active tab.
- 2 Select *Window > New Window*. SCADE Suite opens a new tab with the same content as the selected one. Repeat again to duplicate several times.

Closing Tabbed Views

You can close selectively the views opened in tabs.

To close tabbed views selectively

- Press the close button in the right corner of an active tab (or non-active tab when mouse hovering tab), press the mouse wheel over any non-active tab, or right-click the tab of any opened view and select either:
 - *Close Tab* to close the selected tabbed view.
 - *Close All Tabs* to close all tabbed views currently opened.
 - *Close Other Tabs* to close all tabbed views except the selected one.

Grabbing and Moving Design Representation

You can grab design pages (Diagram Views and Symbol Editor views) and move them around in any direction to display other areas of the active view, especially after zooming (see [“Zooming Design Content in Views”](#) on page 148).

To grab the design and move it

- 1 Click **Pan**  in **Zoom/Pan** toolbar to enable the Pan mode.
- 2 Click on view and drag the mouse to move the design page.

To quit the pan mode, click **Select Mode** .

Zooming Design Content in Views

You can zoom in the content of Diagram Views and Symbol Editor views. Continuous zooming by mouse wheel is available in Selection or Pan mode.

To zoom in/out in continuous on active view

- Hold **Ctrl**-key down and use mouse wheel up/down for zooming in/out. Hold **Shift**-key down and use mouse wheel up/down to scroll horizontally.

To zoom in/out by incremental steps

- 1 Click **Zoom**  in **Zoom/Pan** toolbar to enable the zoom mode.
- 2 Click left/right mouse buttons for zooming in/out by steps (about 25%).

To set zoom factor on active view

- Right-click inside the active view and select *Zoom* to magnify to: **50%**, **75%**, **100%** and **200%**. Click  in **Zoom/Pan** toolbar to magnify view to 100%.

To fit the content of the active view on screen

- Click **Zoom to Fit**  in **Zoom/Pan** toolbar to fit the content of the active view to the display size of the view.

To zoom on specific element in the active view

- Select the design element to magnify in the view and click **Zoom to Selection**  in **Zoom/Pan** toolbar to magnify the selected item.

Note

Zoom to Fit/Zoom to Selection are active in selection, zoom, or pan modes.

Underlining Truncated Text in Views

Within the framework of certification activities, it is of utmost importance to be able to read and check any information that describes design elements in the graphical representation of SCADE Suite models.

Truncation occurs when text cannot be fully displayed. By default, any truncated text is indicated graphically by an ellipsis (...) mark. It is possible to highlight truncated data by means of a hatch background.

Note

All views inserted in generated reports to illustrate model design also display the ellipsis mark and background hatching when enabled in Diagram Views.

CONTEXT OF TRUNCATED INFORMATION

Operators	Any value used in symbol files to display name, instance name, signature, parameters, or hiddens. Iterator size
Control blocks	Name of State Machines, If blocks, and When blocks
States	Name of states and textual diagrams embedded in states
Annotations	Single line and multiline graphical annotations
Types and Constants Views	Any text in columns (only using ellipsis)

To underline visibility of truncated text in Diagram Views

- 1 Go to *Tools > Options* and display the **Views** tab.
- 2 Check **Display effect on truncated text** to insert the hatch background.

Displaying Active View on Full Screen

You can display the active view in full screen mode for maximum visibility and better readability. By default, the full screen mode hides all docking windows and toolbars, and shows only the menu bar on top and the **Full Screen** toolbar.

To display any active view in full screen

- Select *View > Full Screen* (or press *Shift+Alt+Enter*). In full screen mode, display docking windows and toolbars from the respective *View* menus. To return to normal view, click Close Full Screen in **Full Screen** toolbar or press Esc on the keyboard.

Setting Grid Properties to Assist Design Positioning

You can modify the graphical properties and the default behavior of the grid that is available in Diagram Views and Symbol Editor to facilitate design positioning. For using grid, see [“Snapping Design Elements”](#) on page 293.

To set the properties of the grid

- 1 Right-click in **Diagram View** and select *Grid Properties*.

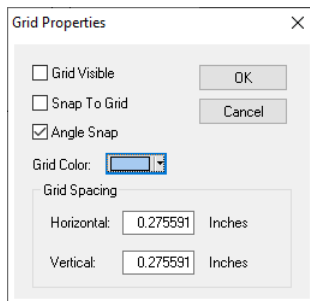


Figure 2.45: Grid Properties window

- 2 In the **Grid Properties** window, select/unselect properties as needed:
 - **Grid visible:** to show/hide the grid on the current view.
 - **Snap to grid:** to snap automatically objects added in the current view.
 - **Angle snap:** no effect.
- 3 Pick a color from the selection arrow to change the color of the grid.
- 4 Specify a new vertical/horizontal line spacing in the corresponding field.
- 5 Click **OK** to validate.

Printing Active Views

Model editing views like Diagram Views, Textual Views, Types Views, Constants Views, or Sensors Views are printable once opened in SCADE Suite. Whenever you need to print the current view, you have access to all standard printing features for printer setup and print-out previewing. Printing may be blocked due to insufficient access rights, check with the administrator.

To set up printer

- Select *File > Print Setup*, modify printer options in the **Print Setup** window, and click **OK** to validate changes.

To preview print-out

- 1 Select the view to print as the active one and select *File > Print Preview* to display a print-out preview.
- 2 Scroll through pages and/or zoom in/out where needed.
- 3 Click **Print** to open the **Print** dialog and launch printing. Surround print-outs with a frame with the **Print frame** option. If unchecked, no frame is visible on print-outs. See [“Selecting General Options”](#) on page 77.

To print the active view

- 1 Select the view to print as the active one and select *File > Print* (Ct r l+P) to open the **Print** dialog. Modify the following settings as required:
 - Select printer and set its properties.
 - Choose to print the active view to a file (. pr n) and specify the target file name and location in **Print to File** dialog.
 - Choose to print the whole view or some pages.
 - Specify the number of printed copies.
- 2 Click **OK** to launch printing.

Organizing and Working with Docking Windows

All docking windows you open offer a wide range of possibilities for organizing SCADE Suite working environment.

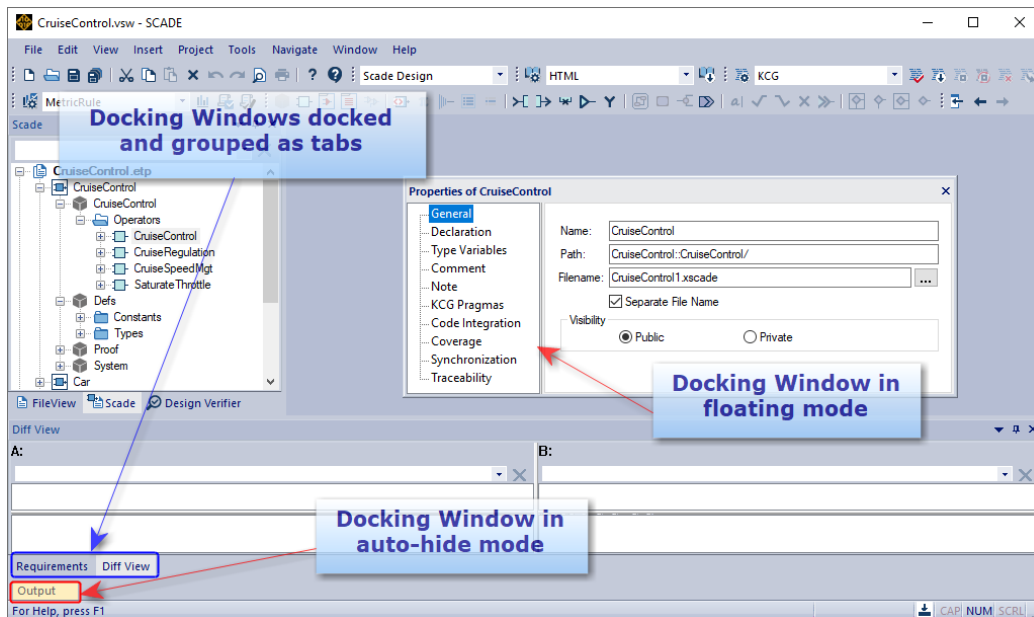


Figure 2.46: Different possibilities for docking window usage

Docking windows can be displayed using either of the following states:

- *Docked* at any place within SCADE Suite user interface. Docking windows can be docked alone or together on the same place using tabs.
- *Floating* anywhere outside SCADE Suite user interface. Docking windows can be floating alone or together on the same place using tabs.
- *Tabbed document* to display docking windows in tab groups with model editing views.

- *Hidden* from SCADE Suite user interface. Docking windows can be shown/hidden by menu control.
- *Auto-hidden* from SCADE Suite user interface. Docking windows appear as simple buttons on the edges of the user interface and expand automatically their content when setting mouse focus over the buttons.

To position docking windows

- 1 Select and hold a docking window (docked or floating).
- 2 Drag it around the interface to display position helper buttons.

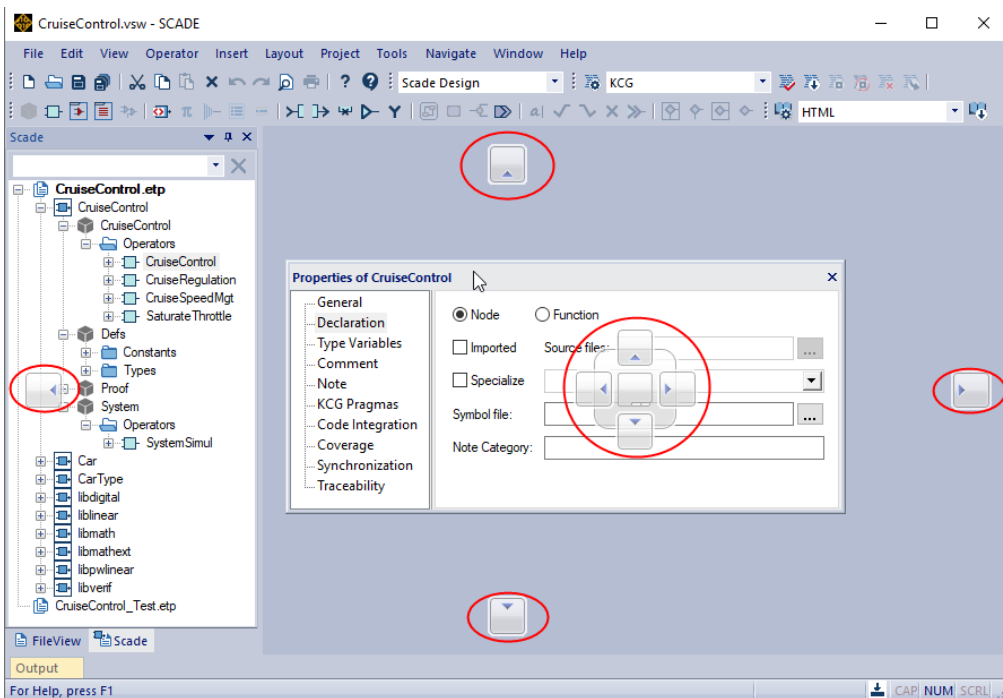
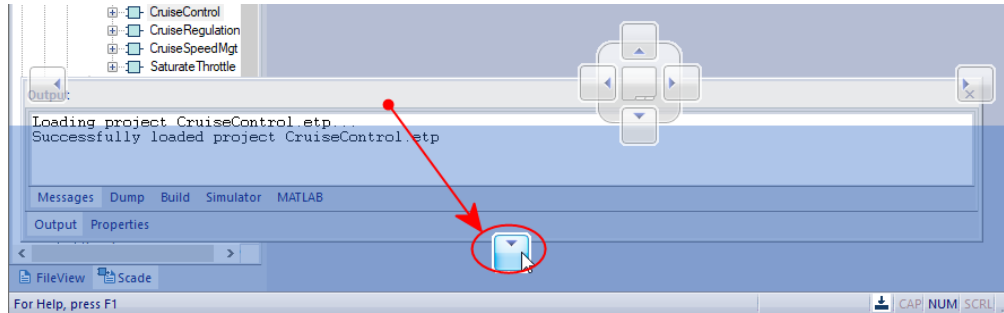


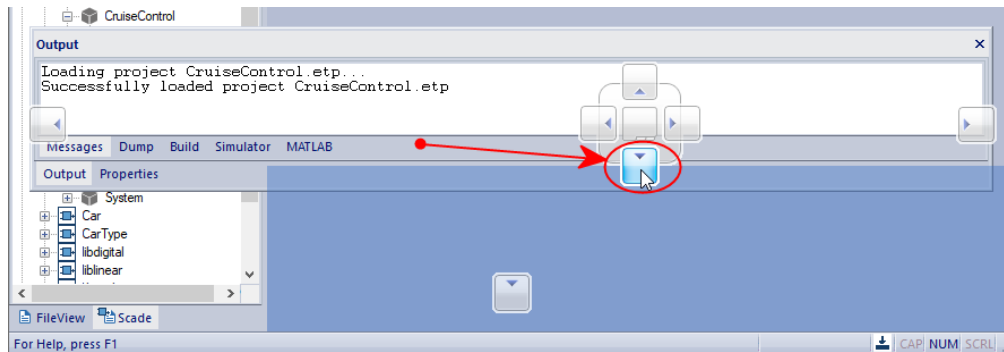
Figure 2.47: Display of position helpers for selecting docking location

- 3 Move over buttons to activate the position helper (turning to blue) and highlight the position where window can be docked. You can choose among buttons as follows:

- Top/Bottom/Left/Right side buttons to dock the window respectively on top/bottom/ left/right sides of the SCADE Suite user interface.



- Group of Top/Bottom/Left/Right buttons to dock the window on top/ bottom/left/right sides of the corresponding display area. You can use the center button to turn the dragged window into a new tab in the display area.



- 4 Release the dragged window to dock it at the selected position.

To group docking windows

- 1 Select and hold a docking window (docked or floating).
- 2 Drag and drop it on the title bar of a docked window to dock them together. Docking windows are docked together and display as tabs.

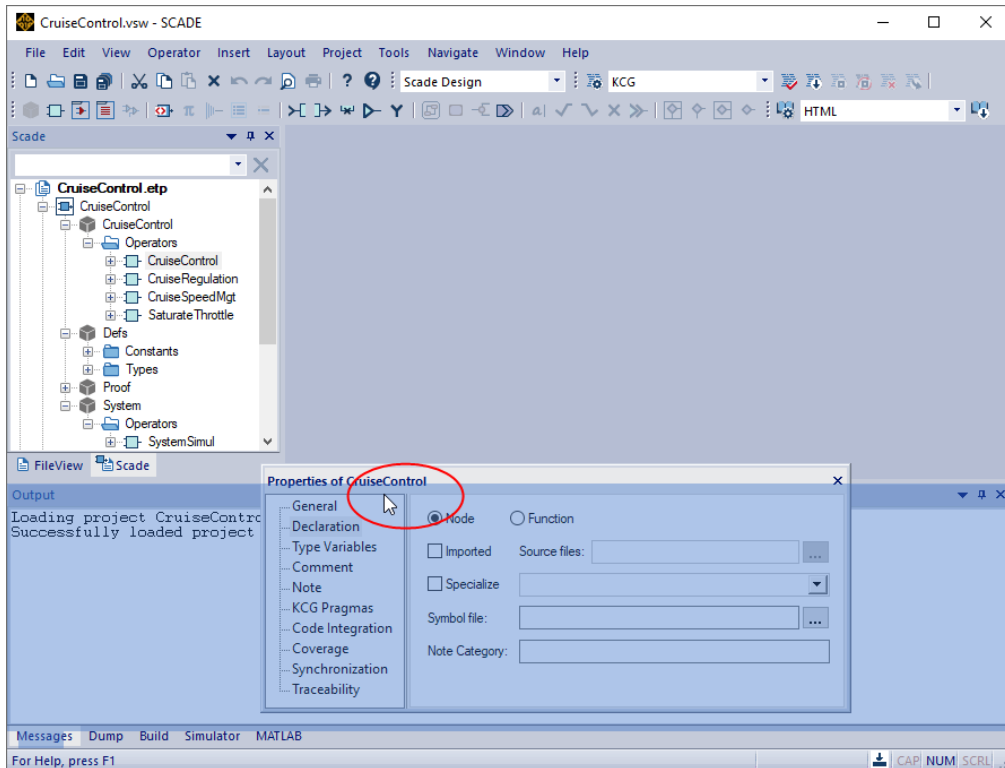


Figure 2.48: Grouping docking windows on same docked position

To turn docked windows into floating windows

- 1 Double click the title bar of a docked window. If you double click a group of docked windows, all windows are floating together and display as tabs. You can also drag the group of docked windows to an empty place.
- 2 Move the floating window as needed.

To change docking window display state

- 1 Use the buttons displayed in the title bar of docked windows.



- 2 Click the arrow button and select either:

- **Floating** to set the window in floating mode
 - **Docking** to set the window in docking mode
 - **Tabbed Document** to set the window as a tabbed view
 - **Auto Hide** to set the window in auto-hide mode
 - **Hide** to remove the window from user interface display
- 3 Click the pin button to set the docking window in auto-hide mode. To return to normal display mode, display the window and click again on the pin button.
 - 4 Click the cross button to hide the docking window.

To reorder tabbed views within docking window

- Grab the tab of a view and move it along other views to exchange place.

3 / Designing Models

This chapter covers all design features and items SCADE Suite provides to help design, configure and edit models with data-flow and control-flow elements.

- [“Declaring Design Elements in Model”](#)
- [“Defining Data Types”](#)
- [“Defining Constants”](#)
- [“Defining Sensors”](#)
- [“Setting Design Instances in Diagram Views”](#)
- [“Setting and Defining Equation Sets”](#)
- [“Setting Operator Parameterized by Size”](#)
- [“Designing Model with Textual Definitions”](#)
- [“Checking Model Semantics”](#)
- [“Modifying Design Elements”](#)
- [“Optimizing Design Layout”](#)
- [“Editing Design Elements with Symbol Editor”](#)
- [“Developing Control and Display Software”](#)

Declaring Design Elements in Model

When designing models in SCADE Suite, you first have to create the building blocks of the design.

- [“Declaring Model Operators”](#)
- [“Declaring Operator Interface”](#)
- [“Declaring Operator Local Data”](#)
- [“Creating and Designing State Machines”](#)
- [“Creating and Designing If Blocks”](#)
- [“Creating and Designing When Blocks”](#)
- [“Declaring and Editing Imported Operators”](#)
- [“Declaring and Editing Specialized Operators”](#)
- [“Setting States/Actions in Embedded/Hidden Mode”](#)
- [“Splitting Operator Design in Multiple Diagrams”](#)
- [“Inserting Existing Item”](#)

Once design elements are declared, use them in **Diagram Views** to design models as explained in [“Setting Design Instances in Diagram Views”](#) on page 232.

Declaring Model Operators

You always have to declare the operators necessary to design the model. Several types of operators are available for design purposes:

- Nodes (operators with memory)
- Functions (operators without internal memory)
- Imported nodes or imported functions (operators with external definitions)
- Specialized operators (nodes or functions) with specialization operators.

You can create and declare model operators within packages as explained [“Organizing Model Design with Packages”](#) on page 124, or you may declare operators, constants, types, and sensors without using packages.

To declare model operators

- 1 Display the **Scade View** and expand existing folders.

- 2 Right-click any package folder or an Operators folder (under a package folder) and select either:
 - *Insert > Node Operator* (Ctrl+Shift+N) or *Insert > Function Operator* (Ctrl+Shift+F) to declare an operator as node or function. You can switch node to function from the declaration properties of the operator (see [“Editing Operator Properties”](#) on page 444).
 - *Insert > Imported Operator* to declare operators (node or function) defined in external source files or defined by specialization. Resume creation in [“Declaring and Editing Imported Operators”](#) on page 194 or in [“Declaring and Editing Specialized Operators”](#) on page 196.
-

Note

If the operator is saved in its own file, SCADE Suite prompts you for operator name and filename provided the Use a creation dialog box general option is enabled (see [“Selecting General Options”](#) on page 77).

- 3 Define the operator interface as needed. Operators are inserted with an empty interface and an empty diagram each having a default name. Imported operators are created without diagram. For user instructions, see [“Declaring Operator Interface”](#) on page 162

Declaring Operator Interface

You can define the interface of the operators you create in models with inputs, outputs, and hidden inputs.

- [“Declaring Inputs/Outputs”](#)
- [“Declaring Hidden Inputs”](#)

Declaring Inputs/Outputs

You can define the interface of any operator from **Scade View** with inputs and outputs. Such elements can be dragged in the Diagram View of the corresponding operator.

Note

The top-down ordering of inputs and outputs in Scade View corresponds to the order of pins on the operator call representation in diagrams.

To declare and define new inputs/outputs

- 1 Right-click the operator to be defined in **Scade View** and select *Insert > Input* (Ctrl+Shift+I) or *Insert > Output* (Ctrl+Shift+O). An input or output is inserted under the **Interface** folder of the selected operator with a default incremental name and a default type.

Tip

Right-click the **Interface** folder of the operator to declare its inputs/outputs.

- 2 Edit the name of the new input/output and press Tab-key to switch to type definition. Start typing text and press Ctrl+Space keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to input/output name definition if needed.

For editing all properties of the inserted element, see [“Editing Input/Output Properties”](#) on page 453. You can also declare new inputs/outputs and simultaneously complete the graphical design using the **Create** toolbar.

Tip

Click the appropriate icon in the toolbar and click again in any active Diagram View to add the reference. The element appears in the operator interface in Scade View, and a reference is created in the Diagram View.

Declaring Hidden Inputs

You can define the interface of any operator from **Scade View** with hidden inputs. Hidden inputs allow to enter directly an input value and reduce the number of links for model readability. Such elements can be dragged in the **Diagram View** of the corresponding operator.

Hidden inputs are not part of Scade language semantic. They are graphical properties displayed on the bottom of an operator call representation which allow to set their value with any expression directly in the diagrams. The top-down ordering of hidden inputs in **Scade View** corresponds to the left to right ordering on the operator representation in diagrams.

To declare and define new hidden inputs

- 1 Right-click the operator to be defined in **Scade View** and select *Insert > Hidden* (Ctrl+Shift+H). A hidden input is inserted under the **Interface** folder of the selected operator with a default incremental name and a default type.

Tip

Right-click the **Interface** folder of the operator to declare its hidden inputs.

- 2 Edit the name of the new hidden input and press Tab-key to switch to type definition. Start typing text and press Ctrl+Space keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to hidden input name definition if needed.

For editing all properties of the inserted element, see [“Editing Input/Output Properties”](#) on page 453. For assigning a value to the new hidden input, see [“Defining Value of Hidden Inputs on Operators”](#) on page 250. You can also declare new hidden inputs and simultaneously complete the graphical design using the **Create** toolbar.

Tip

Click the appropriate icon in the toolbar and click again in any active Diagram View to add the reference. The element appears in the operator interface in Scade View, and a reference is created in the Diagram View.

Declaring Operator Local Data

You can define the local data of the operators you create in models with local variables, probes, and signals.

- [“Declaring Variables”](#)
- [“Declaring Probes”](#)
- [“Declaring Signals”](#)

Declaring Variables

You can define local variables on any operator or state from **Scade View**. Local variables can be seen only by the operator where you declare them. Such elements can be dragged in the **Diagram View** of the corresponding operator. The instances of local variables in **Diagram Views** can be set as input or output flows. Before using local variables in diagrams, check about [“Creating References to Local Data”](#) on page 240.

To declare and define new local variables

- 1 Right-click the operator to be defined in **Scade View** and select *Insert > Local Variable* (Ctrl+Shift+L). A local variable is inserted under the **Locals** folder of the selected operator with a default incremental name and a default type.

Tip

Right-click the **Locals** folder of the operator to declare new local variables.

- 2 Edit the name of the new local variable and press Tab-key to switch to type definition. Start typing text and press Ctrl+Space keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to variable name definition if needed.

For editing all properties of the element, see [“Editing Local Variable Properties”](#) on page 459. You can also declare new local variables and simultaneously complete the graphical design using the **Create** toolbar.

Tip

Click the appropriate icon in the toolbar and click again in any active Diagram View to add the reference. The element appears in the operator declaration in Scade View, and a reference is created in the Diagram View.

To declare local variables at state or transition levels, proceed as described above but select a state or a transition in **Scade View**. For details about the use of local variables in transitions, see [“Setting Transition Conditions and Actions”](#) on page 178.

Declaring Probes

You can define probes on any operator or state set in **Scade View**. Probes allow to define a new variable and associate it with a given flow. They enhance observation during simulation or execution of the generated code. Such elements can be dragged in the **Diagram View** of the corresponding operator. Probes can be used only as output flows.

To declare and define new probes

- 1 Right-click the operator to be defined in **Scade View** and select *Insert > Probe* (Ctrl+Shift+P). A probe is inserted under the `Locals` folder of the selected operator with a default incremental name and a default type. Alternatively, Right-click the `Locals` folder of the operator to declare new probes.
- 2 Edit the name of the new probe and press Tab-key to switch to type definition. Start typing text and press Ctrl+Space keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to probe name definition if needed.

For editing all properties of the element, see [“Editing Probe Properties”](#) on page 456. You can also declare new probes and simultaneously complete the graphical design using the **Create** toolbar.

Tip

Click the appropriate icon in the toolbar and click again in any active Diagram View to add the reference. The element appears in the operator declaration in Scade View, and a reference is created in the Diagram View.

To declare probes at state or transition levels, proceed as described above but select a state or a transition in **Scade View**. For using probes, see [“Creating References to Local Data”](#) on page 240. For using probes in transitions, see [“Setting Transition Conditions and Actions”](#) on page 178.

Declaring Signals

You can define signals on any operator set in **Scade View**. Signals enable State Machines to interact with the model by broadcasting signal emission or testing signal presence.

To declare new signals at operator level

- 1 Right-click the operator to be defined in **Scade View** and select *Insert > Signal* (Ctrl+Shift+A) to add the element in the Locals declaration of the selected operator.

Tip

Right-click the Locals folder of the operator to declare new signals.

- 2 Set the properties of the element as described in [“Editing Signal Properties”](#) on page 462.

You can also declare new probes and simultaneously complete the graphical design using the **Create** toolbar.

Tip

Click the appropriate icon in the toolbar and click again in any active Diagram View to add the reference. The element appears in the operator declaration in Scade View, and a reference is created in the Diagram View.

To declare signals at state or transition levels, proceed as described above but select a state or a transition in **Scade View**. For using signals, see [“Creating References to Local Data”](#) on page 240. For using signals in transitions, see [“Setting Transition Conditions and Actions”](#) on page 178.

Creating and Designing State Machines

You design models with the help of State Machines to introduce control flow in the design. You define such elements with states and transitions directly in **Diagram Views**.

SCADE Suite Editor automatically inserts in **Scade View** all the State Machines and their design elements you create and edit graphically.

- [“Creating State Machines”](#)
- [“Creating States”](#)
- [“Drawing and Editing State Transitions”](#)
- [“Drawing and Editing Fork Transitions”](#)
- [“Setting Transition Creation Modes”](#)
- [“Setting Transition Conditions and Actions”](#)

Creating State Machines

You create State Machines directly in **Diagram Views** to define them in models without having to declare them in the **Scade View**. A State Machine can contain only a collection of states connected by transitions.

State Machines are identified by a unique default name <SMn> (where n is an incremental integer) assigned by SCADE Suite with respect to the scoping rules of the Scade language. If required, default names may be edited from the properties of the State Machine as shown in [“Editing State Machine Properties”](#) on page 470. Default names are unique in the parent operator definitions.

To create State Machines

- 1 Open the **Diagram View** of the operator where to define a State Machine.
- 2 Click **New State Machine**  in **Create** toolbar.
- 3 Click again in the drawing area to insert the element. Alternatively, click and drag immediately to size the element as needed. Resizing by grabbing handles after creation is possible if the State Machine content can be displayed by the new size.

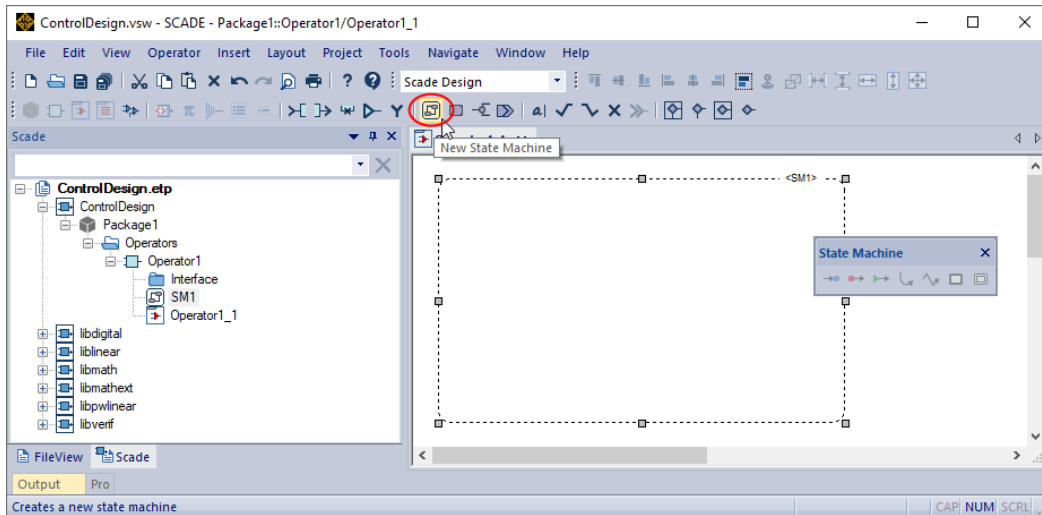


Figure 3.1: Creating State Machines

The dashed border identifies the drawing area where you can design the behavior of the State Machine by combining control and data flow elements. The created State Machine is automatically added in the **Scade View** with its default name under the definition folder of the active operator.

To create parallel State Machines

- 1 Click **New State Machine**  in **Create** toolbar.
- 2 Click again on the same parent level in the **Diagram View**.

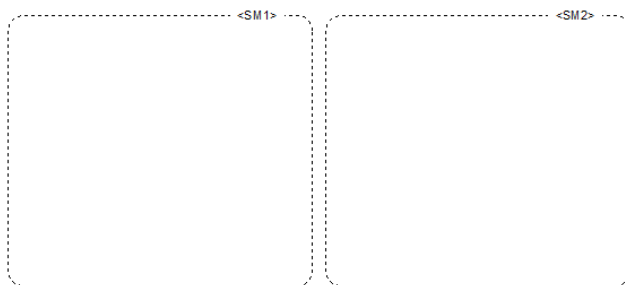


Figure 3.2: Creating parallel State Machines

You can also design operators with data flow in parallel with State Machines as below.

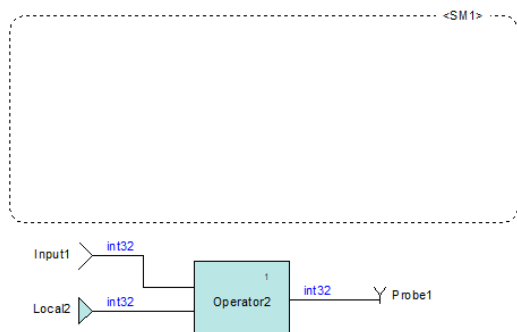


Figure 3.3: Creating State Machine combined with data flow

State Machines can be moved, resized, copied/pasted, aligned, or distributed with certain conditions and effects. For details about design and layout activities, see [“Optimizing Design Layout”](#) on page 283.

To move State Machines

- Grab and hold State Machine symbol by the top or by any side and move it with all its content to a new place that can contain it entirely. If the parent changes during the move, the operation is equivalent to cut/paste.

Creating States

You create states into State Machines to define the control flow and the system behavior. States can contain any combination of data flow elements, assertions, or State Machines. Remember, you can switch displays to facilitate design and reading of state content as described in [“Switching Embedded/Hidden State Display”](#) on page 200.

To create states in State Machines

- 1 Open an operator **Diagram View** where you defined a State Machine.
- 2 Click **New State**  in **Create** toolbar.
- 3 Click again in the State Machine drawing area to insert the element.
Alternatively, click and drag immediately to size the element as needed.

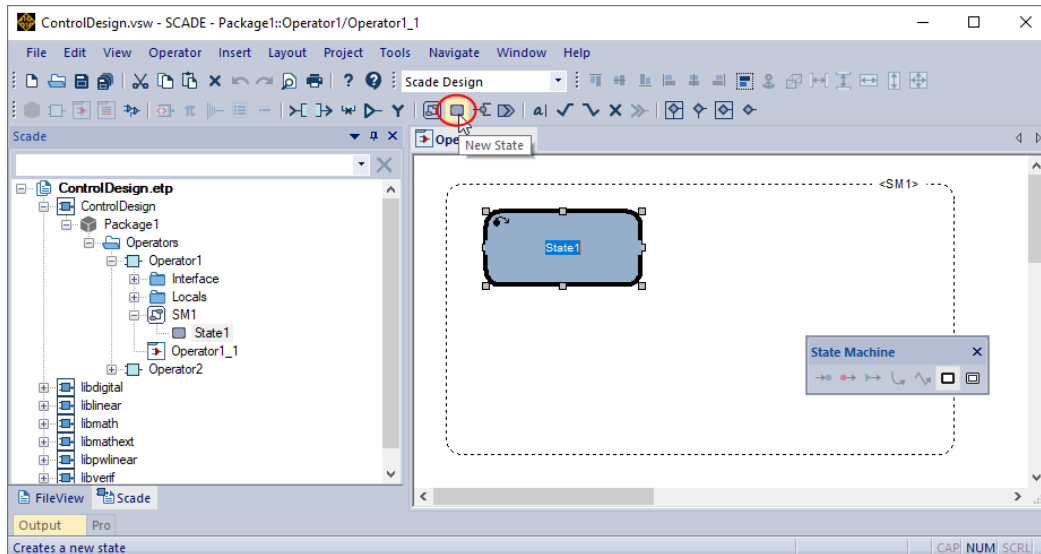


Figure 3.4: Creating states

The newly created state is automatically added in **Scade View** with its default name under the parent State Machine.

- 4 Rename the created state directly from its symbol. States must have a unique name in the parent operator definitions. By default, states are named with respect to the scoping rules of the Scade language. If required, state names can be edited from the Properties of state elements as shown in [“Editing State Properties”](#) on page 471.
- 5 From **State Machine** toolbar, set the state as required:

- Click **Set state as initial**  for initial states. The first state created in a state machine is initial. When setting a new state as initial, any initial state already in the state machine is reset.
 - Click **Set state as final**  for final states.
 - Click both to set the state as initial and final.
 - Leave the state as created.
- 6 Continue the design and connect elements as specified for the system.
- States can be designed with textual content as explained in [“Defining Textual States or Actions”](#) on page 266. It is also possible to create, paste, or drop any data flow element or State Machine into created states. States can be moved, resized, copied/pasted or dropped in State Machines, aligned, or distributed with certain conditions and effects. For details about layout activities, see [“Optimizing Design Layout”](#) on page 283.

Drawing and Editing State Transitions

You create transitions in State Machines to connect states with each other. Transitions contain trigger conditions and possibly actions. For details, see [“Setting Transition Conditions and Actions”](#) on page 178.

To draw state transitions in State Machines

- 1 Position the cursor over the border of the source state (crosshair cursor) and click the border to start drawing the transition.
- 2 Click over the target state (if empty or in hidden mode) or over the header of the target state (if in embedded mode). Specify the exact path of the transition by clicking one or several times between source and target.

Tip

Right-click on wire or transition line where to add a point and select *Add/Remove Point* or hold C t r l-key down and click on empty place to add point.

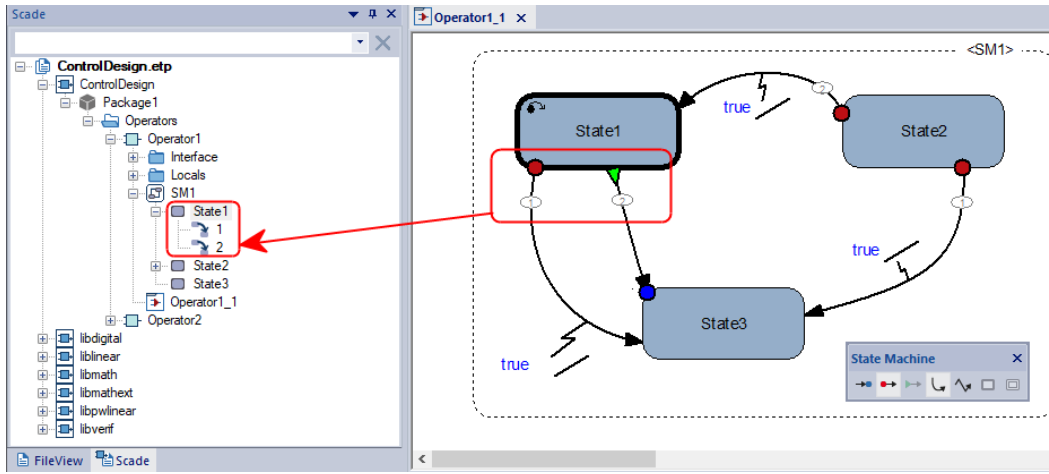


Figure 3.5: Creating state transitions

The newly created state transition is automatically added in **Scade View** with its priority number under the corresponding source state declaration.

3 Set the transition kind from **State Machine toolbar:**

- Click  for weak transition.
- Click  for strong transition.
- Click  for synchro transition.

4 Set the transition layout from **State Machine toolbar:**

- Click  for Bezier style.
- Click  for Polyline style.

The default transition kind and default transition layout can be adjusted as described in [“Setting Transition Creation Modes”](#) on page 178. You can set transition kind and layout by contextual menus on the transition. For semantic details about transitions, consult [“State Transitions”](#) on page 32 in Chapter 3 / of *Scade Language Primer*.

5 Adjust the graphical display of the transitions for enhanced readability.
Moving states' position modifies the shape of transitions in Bezier style.

- 6 Define the transition actions and conditions as explained in [“Setting Transition Conditions and Actions”](#) on page 178.
- 7 Set the transition as resuming (if needed): see the general properties of transitions in [“Editing Transition Properties”](#) on page 474. Transitions with history allow to freeze the execution of a state when exited and to resume execution when reentered again. You can attach any number of history transitions to a state.
- 8 Edit the transition priority (if needed): see the general properties of transitions in [“Editing Transition Properties”](#) on page 474. Priorities are displayed in a circle next to the transition base. Strong transitions have a higher priority than weak ones. Managing all transition priorities from a source state is possible from the transition properties of the state as explained in [“Editing State Properties”](#) on page 471.

Transitions can be reconnected to other states or moved using tangents or intermediate points displayed when selected in **Diagram Views**.

To reconnect transitions

- 1 Select the source or target point of a transition.
- 2 Grab and hold with the cursor and move to another state.
- 3 Unhold the cursor on the border of the selected state to attach.

To move transitions

- Select the transition and use one of the following possibilities:
 - Grab and hold the connecting point to move it to the desired position. In Bezier style, tangents are automatically positioned while moving.
 - Select a tangent point (Bezier lines) and move it to position the transition.

- Select an intermediate point (polylines) and move it to position the transition. Transitions in polyline can be moved from intermediate points. Intermediate points on Bezier line have two tangent points that can be moved independently.

To move transition labels

- 1 Grab and hold the label describing transition activation.
- 2 Move the text to the desired position.
- 3 Resize the left or right part of the label as needed.

You can also set created transitions from the properties of transition elements as described in [“Editing Transition Properties”](#) on page 474.

Drawing and Editing Fork Transitions

You can create fork transitions in State Machines to group transitions and design decision branches by setting different conditions and priorities on transitions.

To create fork transitions in State Machines

- 1 Open the view of the operator where you defined states and transitions.
- 2 Click **New Fork Symbol**  in **Create** toolbar.
- 3 Click on an existing transition to fork. The fork symbol can be moved freely in the **Diagram View**.
- 4 Click over the border of the fork symbol when the cursor is a circled plus, then click on the border of the target state to attach a new fork transition: create a transition with intermediate points by clicking in the view where to add such points, or right-click on wire or transition line where to add a point and select *Add/Remove Point* or hold C t r l-key down and click on empty place to add a point.

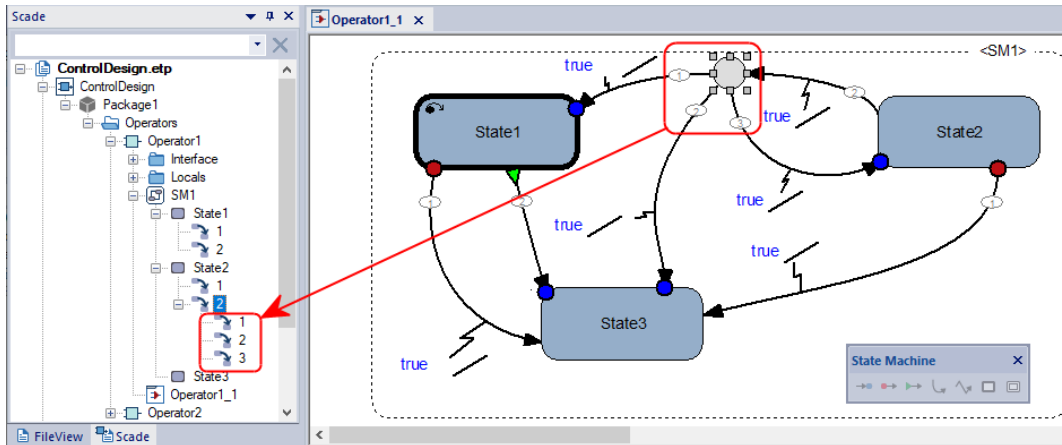


Figure 3.6: Creating fork transitions

The newly created fork transition is automatically added in **Scade View** with its priority number under the icon of the corresponding parent transition.

5 Set the transition kind from **State Machine** toolbar:

- Click  for weak transition.
- Click  for strong transition.
- Click  for synchro transition.

For weak fork transitions, the blue dot is displayed for each leaf transition unlike strong and synchro transitions highlighted only on the main transition from the source state.

6 Set the transition layout from **State Machine** toolbar:

- Click  for Bezier style.
- Click  for Polyline style.

The default transition kind and default transition layout can be adjusted to one's own needs as described in [“Setting Transition Creation Modes”](#) on page 178. You can set transition kind and layout by contextual menus on the transition.

- 7 Set the fork transition as an “else” branch with no condition: see the general properties of transitions in [“Editing Transition Properties”](#) on page 474.

Warning

The priority of an else branch is the lowest in the set of fork transitions. You cannot set several else branches in the same set of fork transitions: setting a branch as else while one already exists turns the existing branch into a normal fork transition with a true condition; the priority of the new else branch increments by one.

- 8 Edit the fork transition priority: see from the general properties of transitions in [“Editing Transition Properties”](#) on page 474. Priorities are displayed in a circle close to the base of the transition. Strong transitions have always a higher priority than weak transitions. Managing all fork transition priorities is possible from the transition properties of the parent transition as explained in [“Editing State Properties”](#) on page 471.

As state transitions, fork transitions can be reconnected (under strict conditions) or moved with the help of tangents and intermediate points displayed when selecting transitions in **Diagram Views**. Fork transitions are reconnected as follows:

- Target points of terminal forked transitions (from final fork symbol) to another state.
- A forked transition to a fork symbol or state only if it is not the last incoming or outgoing transition of its fork symbol.
- Source points of main transition (from source state to fork symbol) to another state.

Deleting a fork transition when it is the last one results in merging the conditions and actions of the transition with the conditions and actions of the parent transition. The parent transition is automatically reconnected to the target. Conditions are merged with an “And” operator; actions are simply concatenated.

Setting Transition Creation Modes

You can modify transition creation default by setting the creation mode for default kind (weak, strong or synchro) and default layout (Bezier or polyline).

To set transition creation modes

- 1 Display the **State Machine** toolbar and click in any **Diagram View** to highlight the currently active creation modes.

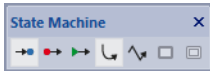


Figure 3.7: State Machine toolbar showing selected modes for transition creation

- 2 Click inactive buttons to switch the default kind or default layout. Active modes are highlighted by a light frame.

Setting Transition Conditions and Actions

You can define transitions with the help of actions and conditions. Actions and conditions can be specified directly in **Diagram Views** and edited separately. The syntax of actions and conditions must comply with Scade language:

- Conditions are expressed with the help of `expr`;
- Actions are expressed with the help of the `eqs` rule, namely `{{equation;}}` without `let/tel` keywords.

For details about condition and action syntax, consult Chapter 8 about [“Expressions”](#) and [“Scope Declarations”](#) on page 47 in *Scade Language Reference Manual*.

The condition and action text is separated by a slash and attached to the transition by a zigzag line.

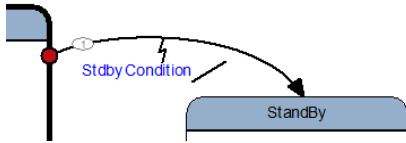


Figure 3.8: Display of transition conditions

You can set conditions and actions from the activation properties of the transitions as described in [“Editing Transition Properties”](#) on page 474) or directly from the graphical diagrams as explained below.

To set condition on created transitions

- 1 Move the cursor in front of the / symbol and click to activate the text field.
- 2 Specify the expression of the condition in the correct syntax. No condition can ever be defined on Synchro transitions.

To set action on created transitions

- 1 Move the cursor behind the / symbol and click to activate the text field.
- 2 Specify the action in the correct syntax. Separate with ; all actions you specify as a list of equations.

Except within strong transitions, conditions and actions can use local variables and signals if defined in the scope of their source state:

To declare local variables, probes, and signals in the transition scope

- 1 In the **Scade View**, expand the source state that contains the transition.
- 2 Right-click the transition identified by its priority and select *Insert > Local Variable*, *Insert > Probe*, or *Insert > Signal* from the contextual menu.
- 3 Define inserted element, see [“Declaring Operator Local Data”](#) on page 164.
- 4 Use variables, probes, and signals to define transition actions and conditions.

Creating and Designing If Blocks

If blocks provide another way to introduce control flow in the design as with State Machines (see [“Creating and Designing State Machines”](#) on page 168) or When blocks (see [“Creating and Designing When Blocks”](#) on page 188). The activation of conditional If branches depends on Boolean conditions, computable in the current cycle (the previous state has no influence on the current state).

You define such elements directly in the Diagram Views.

- [“Creating If Blocks”](#)
- [“Editing If Blocks and Nodes”](#)
- [“Creating and Setting If Nodes”](#)

Creating If Blocks

You create If blocks directly in the **Diagram Views** to define them in models without having to declare them in the Scade View.

To create If blocks

- 1 Open the operator **Diagram View** where to define an If block.
- 2 Click **New If Block**  in **Create** toolbar.
- 3 Click again in the **Diagram View** of the operator to insert the element. Alternatively, click and drag immediately to size the element as needed.
- 4 Rename the created If block directly from its symbol.

If blocks are identified by a unique default name <IfBlockn> (where n is an incremental integer) assigned by SCADE Suite with respect to the scoping rules of the Scade language. If blocks must have a unique name in the parent operator definitions. If required, default names may be edited from the properties of the If block as shown in [“Editing If Block and If Node Properties”](#) on page 464.

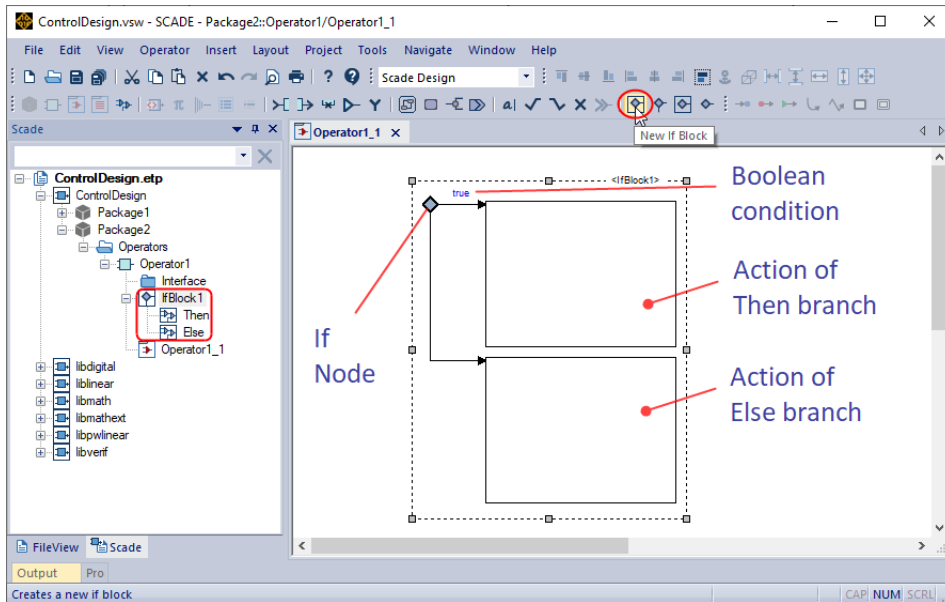


Figure 3.9: Creating If blocks

An If block contains at least one **If node** with two decision branches:

- the horizontal decision branch is the **Then** branch
- the vertical decision branch is the **Else** branch

Each decision branch leads to an **Action** symbol that defines the action to be performed, or to another If node.

Each If node is associated with a Boolean condition set to `true` by default. Whether the condition is evaluated to true or false, the Then branch or the Else branch is activated. The created If block is automatically displayed with its default name and its Then and Else branches in **Scade View**.

After creating and naming an If block, you must express the condition(s) of its activation by designing Boolean decision branches with one or several If Nodes and specifying explicitly the action to perform for each Then/Else branch activation.

Creating and Setting If Nodes

You can create new If nodes to introduce new conditions and thus define more complex If blocks. You create either blank If nodes or paste existing ones. All If nodes under the starting If node are evaluated one after the other until the action to be performed. You can switch displays to facilitate design and reading of action content as described in [“Switching Embedded/Hidden Action Display”](#) on page 201.

To create If nodes

- 1 Open the operator **Diagram View** in which you defined an If block.
- 2 Click **New If Node**  in **Create** toolbar.
- 3 Move the pointer over a decision branch and click as follows:
 - over *Then* branch (horizontal link) to create new If node and *Else* branch with empty action as illustrated on next figure.

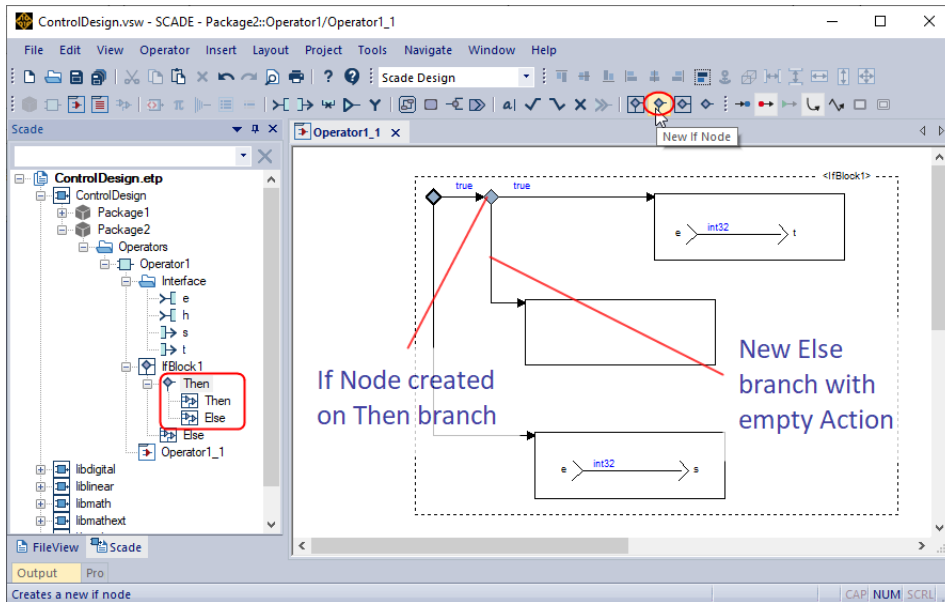


Figure 3.10: Creating new If node on Then branch

Note

In Scade View, new If node replaces the selected branch under parent If node.

- over *Else* branch (horizontal link part) to create new If node and *Else* branch with empty action as illustrated on next figure.

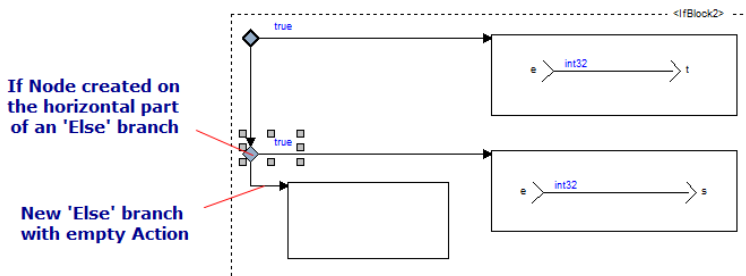


Figure 3.11: Creating new If node on horizontal part of Else branch

- over *Else* branch (vertical link part) to create new *If* node and *Then* branch with empty action as illustrated on next figure.

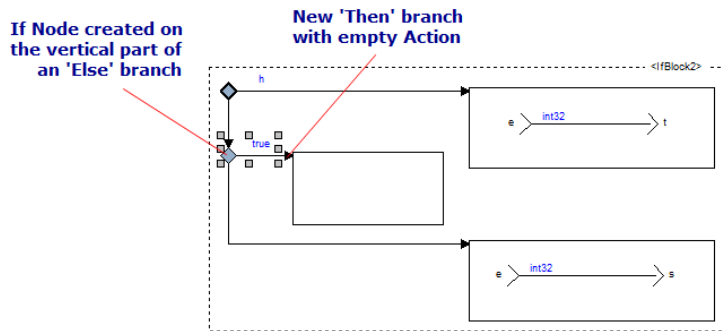


Figure 3.12: Creating new If node on vertical part of Else branch

You can create If nodes only if enough space is available. If required, make space in the If block by resizing it and by moving other elements. You can also switch branches of If nodes (see [“Editing If Blocks and Nodes”](#) on page 185).

To set condition expression of If nodes

- 1 Move the pointer over the label of the condition of an If node and click twice slowly (or select it and press <F2>) to enter in editing mode.



Figure 3.13: Setting Condition of an If node

- 2 Specify the condition expression in the correct syntax. You can also set a condition from If node properties (see [“Editing If Block and If Node Properties”](#) on page 464).

To define actions of If nodes

- Insert design elements in the action design area and connect them as specified for the system. You can also design actions with textual content (see [“Defining Textual States or Actions”](#) on page 266).

Editing If Blocks and Nodes

You can move, resize, or copy/paste any block items used in the design of conditional blocks. You can also delete branches selectively.

- [“To move If blocks, If nodes, and decision actions”](#)
- [“To switch branches of If nodes”](#)
- [“To copy/paste If blocks or If nodes”](#)
- [“To delete If node decision branches”](#)
- [“To copy/paste actions from If nodes”](#)

To move If blocks, If nodes, and decision actions

- 1 Move the pointer over the element symbol to move as follows:
 - Over the dashed border of an If block to move the If block.
 - Over an If node to move the If node (created If nodes can be moved only orthogonally to the incoming link).
 - Over any side of an action to move the action.
- 2 Click to grab the element symbol.
- 3 Hold and drag the symbol and all the elements that depend on it to a new place that can contain all the moved elements. You can move any element as long as it does not intersect with another object in the new place. If the parent changes during the move of an If block, the operation is equivalent to cut/paste.

You can resize If blocks and actions manually or use *Layout* commands to optimize the design of the created blocks as described in [“Optimizing Design Layout”](#) on page 283.

To copy/paste If blocks or If nodes

- 1 Right-click an If block or If node symbol and select *Copy* to copy the selected element with all its content.
- 2 Open the **Diagram View** where to paste.
- 3 Paste the selected item as follows:
 - For *If block selection*, right-click on an empty space and select *Paste* to paste the selection with all its content. It is possible to paste an entire block into an empty action of another block provided the action design area displays sufficient space.
 - For *If node selection*, right-click an *empty* action and select *Paste* to replace the action by the selection with all its content.

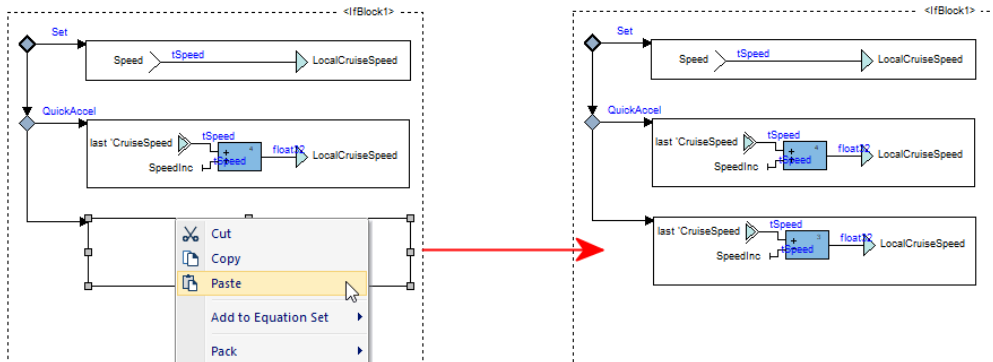


Figure 3.14: Copy/pasting an If node on empty action

To copy/paste actions from If nodes

- 1 Right-click an action in If block and select *Copy*.
- 2 Display the operator or the block where to paste.
- 3 Right-click an *empty* action and select *Paste* to paste the action and its content.

To switch branches of If nodes

- Right-click an If node symbol in If block and select *Switch branches*.

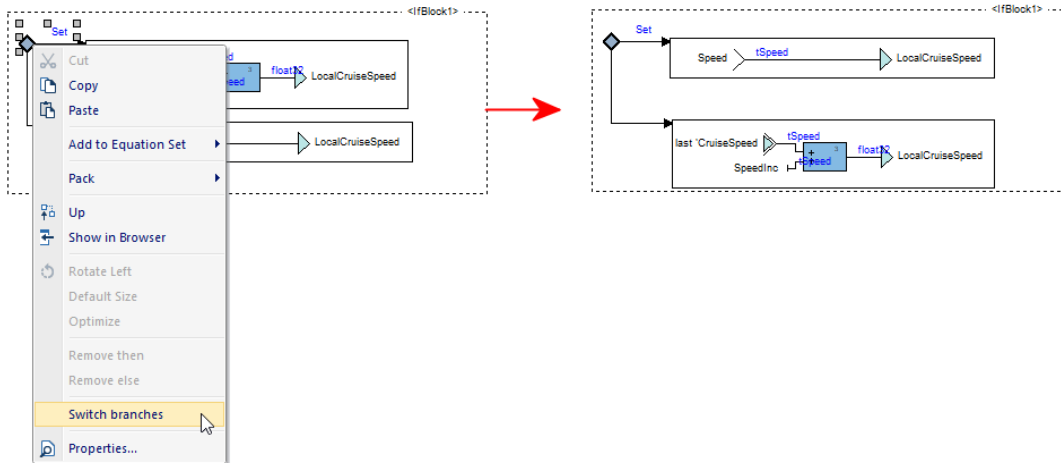


Figure 3.15: Switching branches of an If node

To delete If node decision branches

- Right-click an If node and delete one of the If node else/then branches as follows:
 - Select *Remove then* to remove the If node and its Then branch content. The Else branch is left as the new branch of the parent If node.
 - Select *Remove else* to remove the If node and its Else branch content. The Then branch is left as the new branch of the parent If node.

If you click **Delete**  from **Standard** toolbar after selecting any If node symbol, all decision branches are deleted. An empty action symbol replaces the deleted node. You can never delete the initial If node of the block.

Creating and Designing When Blocks

When blocks provide another way to introduce control flow in the design as with State Machines (see [“Creating and Designing State Machines”](#) on page 168) or If blocks (see [“Creating and Designing If Blocks”](#) on page 180). The activation of conditional When branches depends on the enumerated condition, computable in the current cycle (the previous state has no influence on the current state).

You define such elements directly in the Diagram Views.

- [“Creating and Setting When Blocks”](#)
- [“Editing When Blocks and Branches”](#)
- [“Creating and Setting When Branches”](#)

Creating and Setting When Blocks

You create When blocks directly in the Diagram Views to define them in models without having to declare them in **Scade View**.

After creating and naming a When block, you must express the enumerated condition of its activation. All cases of the enumeration shall be then specified explicitly by designing When branches.

To create When blocks

- 1 Open the operator **Diagram View** where to define a When block.
- 2 Click **New When Block**  in **Create** toolbar.
- 3 Click again in the **Diagram View** of the operator to insert the element. Alternatively, click and drag immediately to size the element as needed. Resizing by grabbing handles after creation is possible if the When block content can be entirely displayed by the new size and if it does not intersect with any other object.

- 4 Rename the created When block directly from its symbol. When blocks are identified by a unique default name <WhenBlockn> (where n is an incremental integer) assigned by SCADE Suite with respect to the scoping rules of the Scade language. When blocks must have a unique name in the parent operator definitions. When blocks names can also be edited from their Properties as shown in [“Editing When Block and When Branch Properties”](#) on page 467.

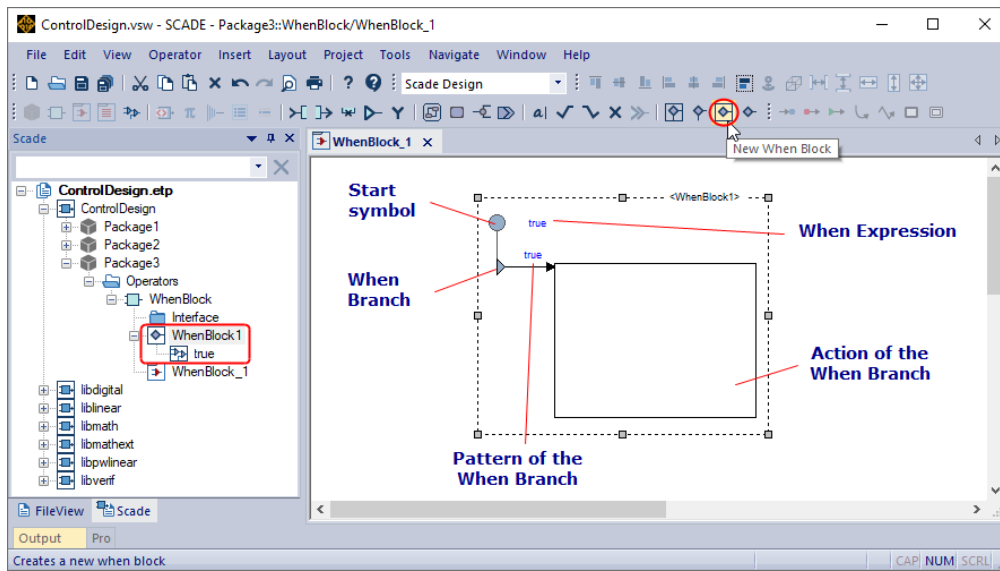


Figure 3.16: Creating When blocks

A When block has a **Start** symbol and a vertical link with at least one **When branch**. The label next to the Start symbol (<true> by default) corresponds to the enumerated condition expression of the When block. A When branch is associated with a condition value from the enumerated condition expression and an **Action** symbol that defines the action to be performed when matching the specified value. The condition value(s) must be of

enumerated type. The created When block is automatically added with its default name and its action under the definition folder of the active operator in **Scade View**.

To set condition expression of When blocks

- 1 Move the pointer over the label of the **Start** symbol and click twice slowly (or select it and press <F2>) to enter in editing mode.

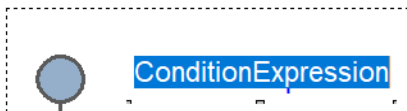


Figure 3.17: Setting the condition of a When block

- 2 Specify the activation condition with an enumeration type or set the condition from properties (see [“Editing When Block and When Branch Properties”](#) on page 467).

Creating and Setting When Branches

You can create new When branches to specify all enumeration cases of When block condition and define actions to be performed when matching each enumerated value of the condition expression. You can switch displays to ease design and reading of action content as described in [“Switching Embedded/Hidden Action Display”](#) on page 201.

To create When branches

- 1 Open the operator **Diagram View** in which you defined a When block.
- 2 Click **New When Branch** ♦ in **Create** toolbar.
- 3 Create a new When branch and its associated action as follows:

- Click the Start symbol to insert a new When branch between the Start symbol and the first following When branch.
- Click a When branch symbol to add a new When branch under that When branch.
- Click on the vertical link to insert a new When branch at the selected position.

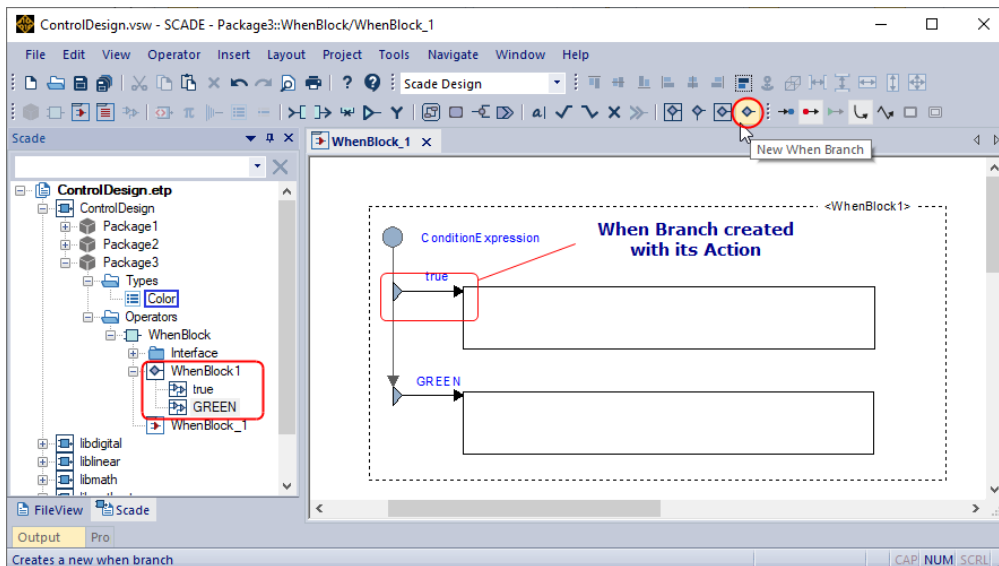


Figure 3.18: Creating new When branch

In **Scade View**, the created When branch is added under the definition of the active When block. Its hierarchical position in **Scade View** is the same as in the diagram.

To set condition value of When branches

- 1 Move the pointer over the label of the When branch and click twice slowly (or select it and press <F2>) to enter in editing mode.

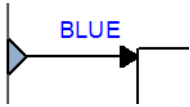


Figure 3.19: Setting the condition value of a When branch

- 2 Specify the condition value with which the activation condition has to match with. It must be an enumerated value. The value is updated next to the corresponding action in **Scade View**. You can also edit the pattern value from its When branch Properties (see [“Editing When Block and When Branch Properties”](#) on page 467).

To define actions of When branches

- Insert design elements in the action design area and connect them as specified for the system. You can also design actions with textual content as explained in [“Defining Textual States or Actions”](#) on page 266.

Editing When Blocks and Branches

You can move, resize, or copy/paste any block items used in the design of conditional blocks. You can also delete branches selectively.

- [“To move When blocks, When branches, and branch actions”](#)
- [“To copy/paste actions from When branches”](#)
- [“To copy/paste When blocks or When branches”](#)
- [“To delete When branches”](#)

To move When blocks, When branches, and branch actions

- 1 Move the pointer over the element symbol to move as follows:
 - Over the dashed border of a When block
 - Over the Start symbol of a When block
 - Over a When branch symbol
 - Over any side of an action
- 2 Click to grab the element symbol.

- 3 Hold and drag the symbol and the elements that depend on it to a new place that can contain all moved elements. You can move any element as long as it does not intersect with another object in the new place. In addition, if the parent changes during the move of a When block, the operation is equivalent to cut/paste.

You can resize When blocks and actions manually or use *Layout* commands to optimize the design of the blocks as described in [“Optimizing Design Layout”](#) on page 283.

To copy/paste When blocks or When branches

- 1 Right-click a When block or a When branch symbol and select *Copy* to copy the selected element with all its content.
- 2 Open the Diagram View where to paste.
- 3 Paste the selected item as follows:
 - For When block selection, right-click on an empty space and select *Paste* to paste the selection. It is possible to paste an entire block into an empty action of another block provided the action design area has sufficient space.
 - For When branch selection, right-click on an empty space within a When block and select *Paste* to paste the selection as the last branch.
 - For When branch selection, right-click a When branch symbol and select *Paste* to paste the selection right under the selected branch.

To copy/paste actions from When branches

- 1 Right-click an action in a When block and select *Copy*.
- 2 Display the operator or the block where to paste.
- 3 Right-click an *empty* action and select *Paste* to paste the action and its content.

To delete When branches

- 1 Select the symbol of the When branch to delete.
- 2 Click **Delete** .

Warning

When deleting a When branch, its action is deleted too. You can delete any branch as long as at least one branch exists in the When block.

Declaring and Editing Imported Operators

You can use imported operators (nodes or functions) to include external code in the model. SCADE Suite is able to execute external code specified in C/Ada source files. You need to identify such source files directly from the properties of the imported operator to include external definitions in the project.

Note

SCADE Suite compiles selected C/Ada files before generating the simulation executable.

After declaring imported operators in models, it is necessary to specify the implementation of imported operators with the external code descriptions.

Note

SCADE Suite is able to generate file templates to help creating and completing external code description as explained in Chapter 20 about [“Implementing Models with Imported Code”](#)

To declare new Imported Operator

- 1 Display the **Scade View** and expand existing folders.

- 2 Right-click any package or model folder, or an Operators folder (under a package folder), and select *Insert > Operator*.

Note

If the operator is saved in its own file, SCADE prompts you for operator name and filename provided the Use a creation dialog box general option is enabled in SCADE (see [“Selecting General Options”](#) on page 77).

- 3 Right-click the operator and select *Properties*.
- 4 Select **Declaration** on the left and set required properties on the right.
- 5 Check **Imported** and declare the imported operator either as node or function by setting the corresponding radio button.
- 6 In the **Source file** field, browse for files and specify the location of the external C/Ada source file containing the code to be implemented.

Warning

The function name in the external C file should be the same as the Imported Operator. Renaming an Imported Operator in the SCADE Suite project requires that you update manually the operator name in the corresponding source file. You may specify a Name pragma to declare the C function name. For user guidelines, see [“Setting Pragmas on Model Elements”](#) on page 859.

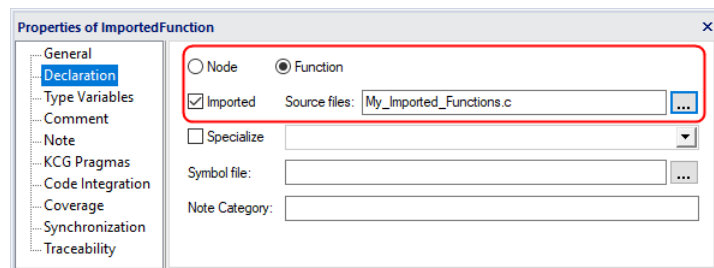


Figure 3.20: Setting imported operator as imported function or imported node

- 7 Close the **Properties** window or click anywhere in the workspace. The file is automatically added in the External Code folder of the **File View**. Double click on Imported Operators in **Scade View** to open their corresponding source file. The source file must be identified previously in the properties of the operator.
- 8 Specify the imported code implementation for functions or nodes as described in Chapter 20 about [“Implementing Models with Imported Code”](#).

RELATED TOPICS

- [“Setting Imported Types”](#) on page 214
- [“Setting Imported Constants with External Code”](#) on page 225
- [“Setting Imported Constants with Imported Types”](#) on page 227

Declaring and Editing Specialized Operators

You can design models with operators specialized by other operators. Specialization allows you to define different behavior for a polymorphic operator depending on the actual data type of the inputs of its instances.

- The *specialized* operator must be an imported operator (node or function).
- The *specialization* operator interface must have types compatible (same types or sub-types) with the types of the interface of the operator they specialize.
- Due to initialization analysis on KCG side, control blocks and assertions are not allowed when specializing imported functions.

Specialization aims at substituting an imported operator by a less general one. Scade language semantics defines typing relations that the specializing operator has to fulfill. These relations cover types, clocks, and initialization analyses. For further details about specialization semantics, see [“User-Defined](#)

[Operators](#)” on page 32 in *Scade Language Reference Manual*. When one instantiates a specialized operator in a model, SCADE Suite computes the correct specialization to set the behavior.

To declare specialized operators

- Declare the operator to specialize as *imported* as described in [“Declaring and Editing Imported Operators”](#) on page 194.

To declare specialization operators

- 1 For each specialization of the imported operator:
 - Create a new operator in the model
 - Specify its interface (compatible with the imported operator interface, but without generic types): inputs/outputs defined with generic types in the imported operator must be defined with non-generic and compatible types (same types or sub-types) in the specialization operator. It is impossible to have two specializations of the same operator with the same interface.
- 2 Right-click the specialization operator in the **Scade View** and select *Properties*.
- 3 Select **Declaration** on the left and set required properties on the right.
- 4 Set **Node** or **Function** to declare the operator as node or function. When the specialization operator is declared as a function, the referenced imported operator can have any kind. On the contrary, when declared as a node, the referenced imported operator can only be a node itself.
- 5 Check **Specialize** and select the operator to specialize in the drop-down list. If the specialized operator is declared in another package, specify the complete name of the package access path (e.g., `Package1::Package2::OperatorA`).

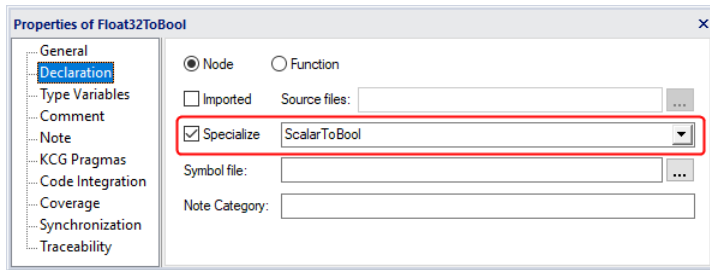


Figure 3.21: Specifying an operator to specialize

- 6 Close the window to validate.
- 7 Drag and drop the *specialized* operator in another operator diagram and connect it with other design elements that have compatible types.

The following is an example of specialized operator with two specializations:

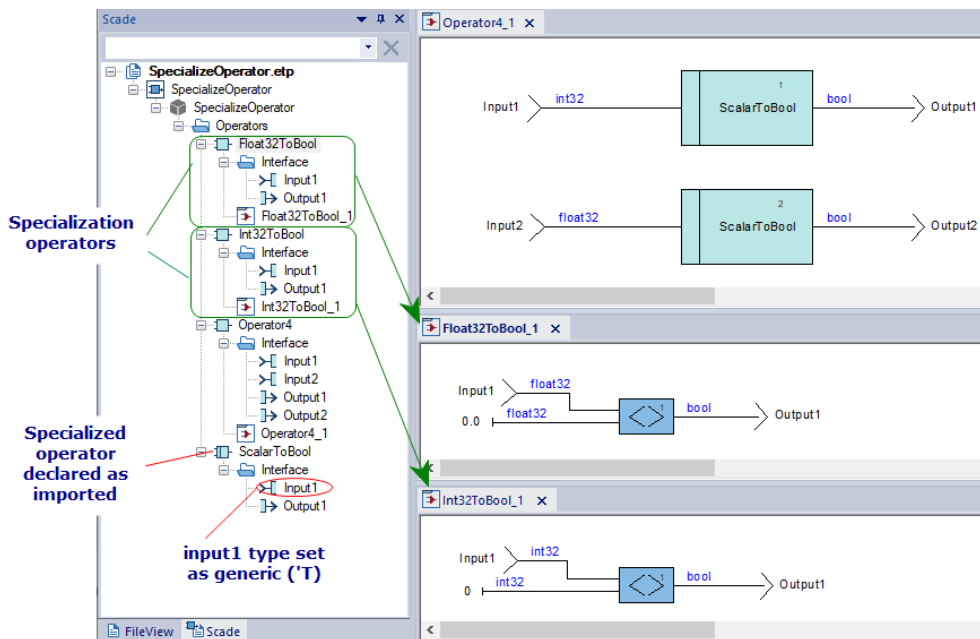


Figure 3.22: Example of specialized operator

- the specialized operator **ScalarToBool** is declared as imported. It has one input of type 'T (generic) and one output of type bool.
- the specialization operator **Float32ToBool** has one input of type float32 and one output of type bool.
- the specialization operator **Int32ToBool** has one input of type int32 and one output of type bool.

In the model, the specialized operator **ScalarToBool** is instantiated and connected to inputs of compatible types. The **ScalarToBool** operator and its specializations are available in the `LibsmLk` library.

Setting States/Actions in Embedded/Hidden Mode

For design and reading purposes, it is possible to switch the display of states and actions between *embedded* and *hidden* display modes. You show state/action content either embedded in the parent operator view or hidden in a separate diagram.

Note

The hidden mode allows to design state and actions using several graphical or textual diagrams as described in [“Inserting Graphical Diagrams”](#) on page 204 and [“Inserting Textual Diagrams”](#) on page 260.

The content of hidden states or actions is stored in a graphical diagram when designed graphically or in a textual diagram when designed textually. For details about textual states, see [“Defining Textual States or Actions”](#) on page 266. The corresponding diagram is automatically inserted under the state icon in the **Scade View** when hiding display.

Note

You can only embed graphical or textual content if symbol space is sufficient. It is impossible to show content as embedded if defined in several diagrams.

You edit the content of states/actions in hidden mode from the corresponding separate diagram(s). By default, each diagram is named `diagram_<name>_1`. The value of `<name>` can be either of the following: name of state; expression of When branch action; Then for Then branch action of If blocks; or Else for Else branch action of If blocks.

- [“Switching Embedded/Hidden State Display”](#)
- [“Accessing Content in Hidden Display”](#)
- [“Switching Embedded/Hidden Action Display”](#)

Switching Embedded/Hidden State Display

You change the representation of states in **Diagram Views** to show their design content as embedded in the parent operator view or hidden in a separate diagram.

To show state display as embedded

- Right-click a state in hidden display and select *Embed content*.
Alternatively, right-click from within the separate view of a state in hidden display and select *Embed content* to switch back to an embedded display.

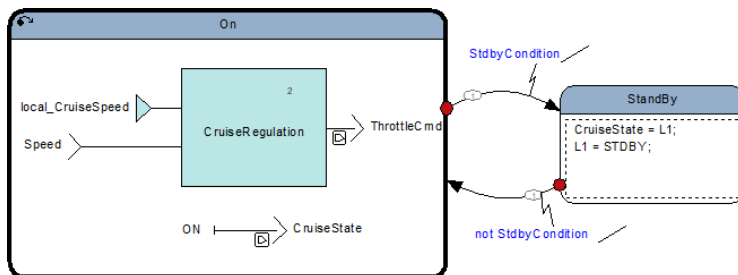


Figure 3.23: Embedded display of states

To hide state display in separate view

- Right-click a state in embedded display and select *Hide content*.

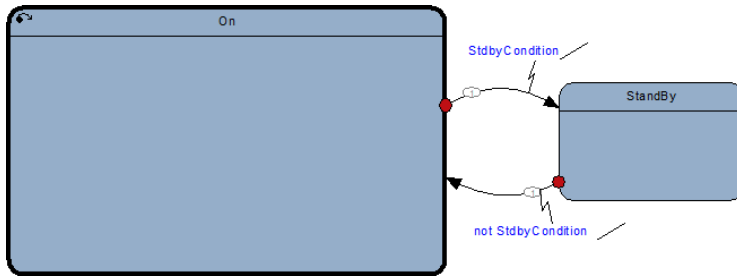


Figure 3.24: Hidden display of states

Tip

You can also set the state display from the properties of state elements as described in [“Editing State Properties”](#) on page 471.

Switching Embedded/Hidden Action Display

You change the representation of actions in Diagram Views to show their design content as embedded in the parent operator view or hidden in a separate diagram.

To show action display as embedded

- Right-click an action in hidden display and select *Embed content*. Alternatively, right-click from within the separate view of an action in hidden display and select *Embed content* to switch back to an embedded display.

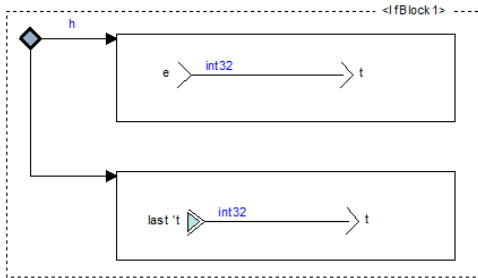


Figure 3.25: Embedded display of actions

To hide action display in separate view

- Right-click an action in embedded display and select *Hide content*.

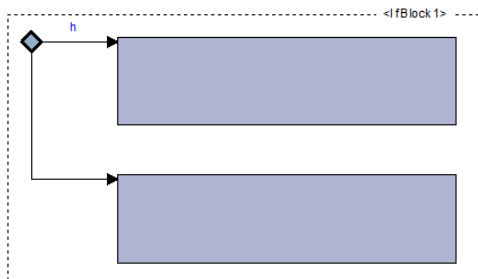


Figure 3.26: Hidden display of actions

Tip

You can also set the action display from the properties of If nodes and When branches as described respectively in [“Editing If Block and If Node Properties”](#) on page 464 and [“Editing When Block and When Branch Properties”](#) on page 467.

Accessing Content in Hidden Display

You can access the content of states/actions in hidden mode by double clicking to open the corresponding diagram(s). The area inside the red dashed border corresponds to the size used to display graphical content in the parent view. You may draw outside the border when design requires more space, but it becomes impossible to show the design as embedded again. If needed, the display border can be shown/displayed contextually or set optionnally.

Important

As long as all design elements do not fit within the red border, you cannot switch back from hidden to embedded display. If you need more space to fit in the embedded display, you can increase the size of the state in the parent view.

To display the separate view of states/actions in hidden mode

- Double click on states/actions in **Diagram View** or in **Scade View** to open them in a separate view. If the state/action content is designed in several diagrams, select and double click from proposed list. If you double click on state/action defined textually, the corresponding text diagram opens.

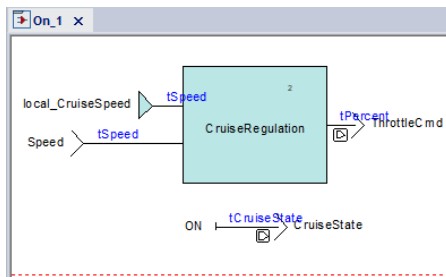


Figure 3.27: Separate view of a graphical state in hidden display

It is possible to control the border line display: right-click in diagram and select *Display Options > Display Border* or go to *Tools > Options* and set the default display by selecting/unselecting **Display Border** in **Views** options.

Splitting Operator Design in Multiple Diagrams

An operator is associated with a diagram to represent the graphical design ([“Diagram Views”](#)). You can insert additional diagrams to represent complex operators, states, or actions across several views. Additional diagrams can be connected with each other via local variables: this allows to split the graphical parts of operators without any hierarchy. You may also design operators with textual diagrams in addition to graphical ones. For details, see [“Designing Model with Textual Definitions”](#) on page 259.

- [“Inserting Graphical Diagrams”](#)
- [“Splitting Complex Operators Automatically”](#)
- [“Connecting Graphical Diagrams”](#)

Inserting Graphical Diagrams

You can design an operator, state, or action with as many diagrams as needed and split the design into several graphical parts. To use outputs from one diagram as inputs in another diagram, the diagrams must be connected (see [“Connecting Graphical Diagrams”](#) on page 205). States and actions must be declared in Hidden Display to allow diagram insertion (see [“Setting States/Actions in Embedded/Hidden Mode”](#) on page 199). The number of diagrams can be restricted from the **Maximum number of diagrams** option set in **General** options (*Tools > Options*). By default, there is no limit (-1).

To insert graphical diagrams

- 1 Display **Scade View** and expand the Operators folder.
- 2 Right-click an operator, state, or action and select *Insert > Diagram*, or click  in **Create** toolbar. An empty diagram is added in **Scade View** and named `diagram_name_i` where `i` is an incremental number and name is the name of the operator, state, or action.

Connecting Graphical Diagrams

You can design an operator with several diagrams provided you link them using local variable as input and output. Any local variable set as output is unique and can be used as many times as needed by input local variables in other diagrams (see [“Declaring Variables”](#) in **Scade View** or [“Creating References to Local Data”](#) in **Diagram Views**).

To connect graphical diagrams

- 1 Design the operator in each diagram from corresponding **Diagram Views**.
- 2 Insert local variables as appropriate on each diagram to enable transmission of flows between diagrams.

Splitting Complex Operators Automatically

You can split automatically operators with complex graphical representation spreading across several pages into several diagrams. Any design element located across page bound belongs to the diagram for the page that contains the upper left corner of the symbol. Data flow wires across page bounds are replaced by an internal local variable of the same type in each diagram.

To split a complex operator into diagrams

- 1 Display the **Diagram View** of the operator to split in several diagrams.
- 2 Right-click in the view and select *Page Bounds* to visualize where the graphical representation is going to split.
- 3 Select the parts to split and move them in distinct split areas.
- 4 Right-click in the view and select *Split*.

Splitted diagrams are named automatically based on the original name. The original diagram is not renamed and new diagrams named `<original_diagram_name>_cl` where `l` (line number) and `c` (column number) correspond to the position of the page of the new diagram in the current diagram. `l` and `c` begin at 0.

Inserting Existing Item

You can insert an existing item (operator or package) defined in a SCADE file (`.xscade` and `.scade`) directly from **Scade View**. When inserting an item, the associated file is automatically added to the active project's set of files.

To insert an existing item in current project

- Right-click a package or root folder (outside package declaration) in **Scade View** and select *Insert > Existing Item*. Browse for the file containing the item from the **Open** window. Click **Open** to validate the choice.

In case a root/non-root conflict is detected leading to change in file contents, user confirmation is requested. If no confirmation is given, the insert operation is aborted and no change is done to the model and project.

Defining Data Types

The data typing to use throughout the model must be declared explicitly. Types you define in SCADE Suite are organized in **Types Views** located in packages or at project root level.

Note

For project file management purposes, types are declared inside packages, but can be declared outside packages at project root level (see [“Multifile Management”](#) on page 66).

You may also use anonymous data types without using explicit declaration but directly in variable declarations. You may use generic types locally to operator inputs, outputs, or local variables, and set numeric types on operator declarations.

You can define data typing in models as follows:

- [“Declaring New Types”](#)
- [“Setting Array Types”](#)
- [“Setting Enumeration Types”](#)
- [“Setting Structure Types”](#)
- [“Completing Definition of Structured Types”](#)
- [“Setting Imported Types”](#)
- [“Declaring Operators Using Generic Types”](#)

Declaring New Types

You declare new types in models at package or model level. You create and declare types using *predefined types* (Boolean, character, numeric) or *user types* (imported, array, structure, enumeration, or any declared type). For editing the properties of declared types, see [“Editing Type Properties”](#) on page 438.

To declare new types

- 1 In **Scade View**, right-click a package, project folder, any type, or the Types folder, and select *Insert > Type*. A type with default name is inserted under Types folder. Alternatively, click **New Type**  in **Create** toolbar to create a type in the **Types View**.
- 2 Double click newly added type in **Scade View** to open the corresponding **Types View**.

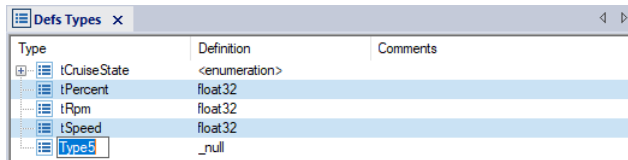


Figure 3.28: New type declared in Types View

- 3 Enter the name of the type in the Type column and define in Definition column:
 - Select the cell and press F2 to edit type definition from keyboard.
 - Switch editing focus between columns using Tab key or arrow-keys.
 - Start typing text and press Ctr l+Space keys to use auto-completion.
 - Enter keyword of predefined type:

bool	char	float32	float64
int8	int16	int32	int64
uint8	uint16	uint32	uint64

- Select the predefined or user type in drop-down list or navigate to opened packages.
 - Validate by pressing Tab key once or Enter key twice.
 - Delete types by selecting the type name and pressing Delete.
- 4 Enter a comment in the Comments column if needed.

Setting Array Types

You can define types as an array of data. Arrays are structured types which allow to ensure consistency between data modeling and the functional level.

Note

Array types are expressed by a base type and a dimension separated by '^'.

You set and edit array types from **Types Views**.

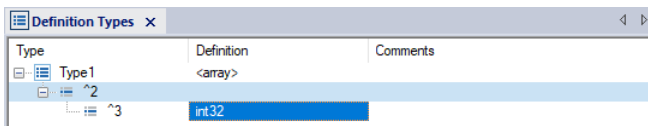


Figure 3.29: Setting a type as data array

Tip

Array types can be used in Diagram Views to define special operators as described in [“Composing/Decomposing Structured Types”](#) on page 236.

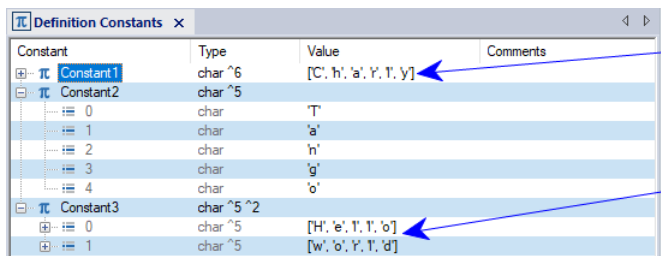
To set array types

- 1 Create a new type (see [“Declaring New Types”](#) on page 207).
- 2 Select **<array>** from the drop-down list available in **Definition** cells.
- 3 Press Enter to validate the selection or double click it. A definition element is added and set the array dimension (default is 1): it must be a static integer expression. Each dimension is referenced by an index; the index of the first dimension is 1. Any additional dimension is inserted above by incrementing index accordingly.
- 4 Specify the base type from the drop-down list of the **Definition** column. Base types can be of any type defined in the model. The type specified for array values is set only for the first dimension.

- 5 Click **New Definition Element**  in **Create** toolbar to add other dimensions to the array and set them as above. For editing definition elements, see [“Completing Definition of Structured Types”](#) on page 213.

When editing array types, you can delete dimensions starting from the top most dimension: select this dimension in Type column and press **Delete**.

You define char array values as if specifying string definitions. String types are not part of the Scade language. Char array values are expressed by a char type and a dimension separated by $\wedge$. Multi-dimensional char arrays can be defined as char^3^2 .



Constant	Type	Value	Comments
Constant1	char ⁶	['C', 'h', 'a', 'r', ' ', 'y']	
Constant2	char ⁵	['T', 'a', 'n', 'g', 'o']	
Constant3	char ⁵ ²	['H', 'e', 'l', 'l', 'o'], ['w', 'o', 'r', 'l', 'd']	

Char array values displayed as string

Multi-dimension char array displayed in character sequence

Figure 3.30: Different representations of char array values in Constants Views

Char array values can be represented in graphical diagrams, Constants Views, or simulation windows as text strings or in character sequence (default).

To define char array values

- Type single values in double quote like "He l l o" and multi-dimensional values ["He l l o", "wor l d"].
- Type the values character by character each in simple quote like ['H', 'e', 'l', 'l', 'o']. To enter a double quote or backslash character in char array values, the backslash character must be used with a double quote or backslash character. For example, "\"tag\\\" displays as [\"", 't', 'a', 'g', '\\'].
- Type the value in hexadecimal ASCII format like "New Line\\x0A".

Change the display mode of char arrays using the **Display char arrays as string** option. For details, see [“Selecting Views Options”](#) on page 79.

RELATED TOPICS

- [“Setting Array/Structure Type Constants”](#) on page 220
- [“Designing with Array Values”](#) on page 367

Setting Enumeration Types

You can configure types as an enumeration of values. It is possible to force integer value mapping. You set and edit enumeration types from **Types Views**.

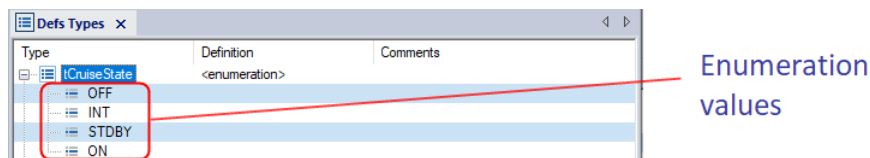


Figure 3.31: Setting a type as data enumeration

To set enumeration types

- 1 Create a new type (see [“Declaring New Types”](#) on page 207).
- 2 Select **<enumeration>** from the drop-down list available in Definition cells. A value is given by default where **i** is an incremental number.
- 3 Specify the value in the Type column and press Enter to validate. You cannot define two enumeration types with the same value name. Enumeration values cannot be integer values.
- 4 Click **New Definition Element**  in **Create** toolbar to specify other enumeration values as described above. For editing definition elements, read more in [“Completing Definition of Structured Types”](#) on page 213.

Setting Structure Types

You can configure types as data structure which corresponds to C structure. Structures are structured types which are useful to ensure consistency between data modeling and the functional level.

You set and edit structure types from **Types Views**.

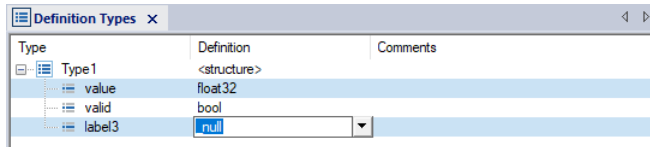


Figure 3.32: Setting a type as data structure

Tip

Structure types can be used in Diagram Views to define special operators as described in [“Composing/Decomposing Structured Types”](#) on page 236.

To set structure types

- 1 Create a new type (see [“Declaring New Types”](#) on page 207).
- 2 Select **<structure>** from the drop-down list available in **Definition** cells. A label value is given by default where *i* is an incremental number.
- 3 Rename the label in the Type column and press Enter to validate.
- 4 Specify a type from the drop-down list of the Definition column.
- 5 Click **New Definition Element**  in **Creation** toolbar to specify new structure fields as described above. For editing definition elements, read more in [“Completing Definition of Structured Types”](#) on page 213.

Completing Definition of Structured Types

When declaring types such as arrays, structures, or enumerations, you can always complete the definition with additional elements inserted below any of the selected components in the type definition.

By selecting the element named Green and inserting a new one with the **New Definition Element**  in **Create** toolbar, you are able to complete the type definition as illustrated in [Figure 3.33](#).

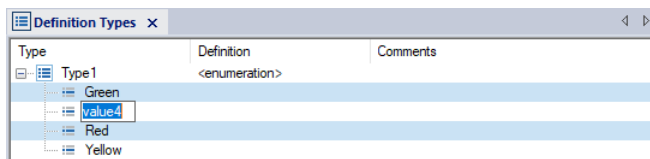


Figure 3.33: Adding definition element below selected item

Instead, if you select the name of the type before clicking the same button, new definition components are added at the end of all type definition components as illustrated in [Figure 3.34](#).

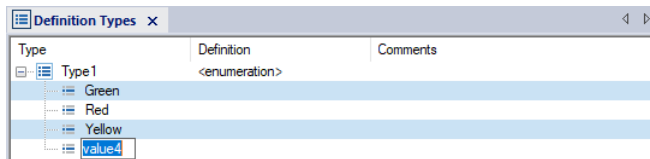


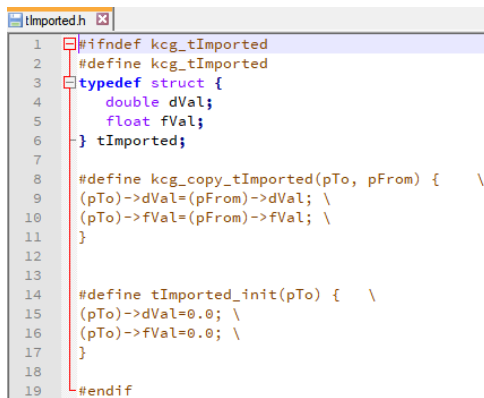
Figure 3.34: Adding definition element as last item

Adding new components for type definitions in the first position is impossible. Use *Edit > Move Up* and *Edit > Move Down* or the Shift+Alt- keys with up or down arrow keys to change the position of components in enumeration or structure types. The definition of array types is specific so that new dimensions are added to the first position.

Setting Imported Types

You can introduce in models user types defined externally as imported types. Only imported operators can process data from imported typed flows. You provide the definition of imported types in header files (.h) and set their declaration in models from **Types Views** and **Properties**. Such files contain definitions relative to the imported type, the initialization macro <TypeName>_init, the copy macro kcg_copy_<TypeName>, and the comparison macro kcg_comp_<TypeName>. The format for defining imported types is as follows:

```
typedef int Type1
```



```
1 #ifndef kcg_tImported
2 #define kcg_tImported
3 typedef struct {
4     double dVal;
5     float fVal;
6 } tImported;
7
8 #define kcg_copy_tImported(pTo, pFrom) { \
9     (pTo)->dVal=(pFrom)->dVal; \
10    (pTo)->fVal=(pFrom)->fVal; \
11 }
12
13
14 #define tImported_init(pTo) { \
15     (pTo)->dVal=0.0; \
16     (pTo)->fVal=0.0; \
17 }
18
19 #endif
```

Figure 3.35: Example of header file containing definitions of imported types

To set imported types

- 1 Create a new type (see [“Declaring New Types”](#) on page 207).
- 2 Right-click the imported type and select *Properties*.
- 3 Select **Declaration** on the left and check **Imported**.
- 4 Click the browsing button to look for the header file and specify its path in the **Definition file** field.

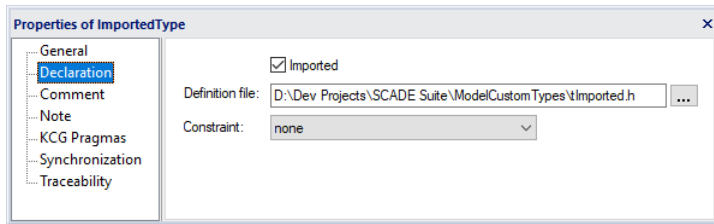


Figure 3.36: Specifying imported type with attached definition file and constraint

- 5 Set the type **Constraint**: select numeric, integer, float, signed, or unsigned in the drop- down list.
- 6 Validate the properties. If the definition file is not present in the project, SCADE Suite automatically adds the file in the active project.

For simulation purposes, type definition files are automatically declared as “**Type Definition File**” (see [“Simulating Models with Imported C Types”](#) on page 675).

For advanced details about imported types in simulation, consult [“Conversion and Comparison Functions for Types”](#) on page 105 in *SCADE Suite Technical Manual*.

- 7 Write the definition of the imported type in the header file.

At code generation, various files are generated by SCADE Suite with information about imported types. This allows to organize links to external declarations and implementations as wanted in C files using macros or a default mechanism. For details, see [“Implementing C Macro User Definitions”](#) on page 676 and Chapter 20 about [“Implementing Models with Imported Code”](#).

RELATED TOPICS

- [“Declaring and Editing Imported Operators”](#) on page 194
- [“Setting Imported Constants with External Code”](#) on page 225
- [“Setting Imported Constants with Imported Types”](#) on page 227

Declaring Operators Using Generic Types

You can define operators with generic types which are local to the operators. Such operators have their own generic type scope: generic types do not mask other types since they have distinct namespace.

- First, declare generic types while typing operator variables.
- Second, set generic types constraints from the operator properties.

To express generic types locally to an operator

- 1 Right-click a variable (input, output, local) of the operator and select *Properties*.
- 2 Select **Declaration** on the left and set required properties on the right.
- 3 Define a generic type by entering ' <TypeID> in the **Type** field. When replacing a generic type by another type, or deleting an object defined by generic type, you delete the generic type if no other object is using it.

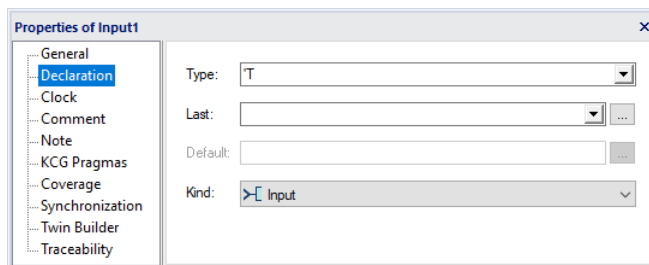


Figure 3.37: Expressing generic types on local variable

To set operators with generic types

- 1 Right-click the operator declaration in **Scade View** and select *Properties*.
- 2 Select **Type Variables** on the left and set required properties on the right.

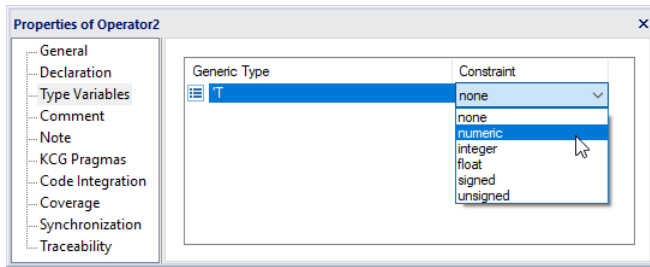


Figure 3.38: Setting generic types as numeric at operator level

- 3 Check in the list of defined generic types the one(s) to be set with a **Constraint**: select none, numeric, integer, float, signed, or unsigned in the drop-down list. You cannot add or remove generic types from the list. Creation and deletion of generic types is done through their usage on operator variables. Press F2 on selected type or double click in the list to edit its name.

Defining Constants

The constants you need to use throughout the model must be declared explicitly based on types declared in the model or using anonymous types defined textually (see [“Defining Data Types”](#) on page 207). Constants you define in SCADE Suite are organized in **Constants Views** located in packages or at project root level. For project file management purposes, types are declared inside packages, but can be declared outside packages at project root level (see [“Multifile Management”](#) on page 66).

- [“Declaring New Constants”](#)
- [“Setting Imported Constants with External Code”](#)
- [“Setting Array/Structure Type Constants”](#)
- [“Setting Imported Constants with Imported Types”](#)

Declaring New Constants

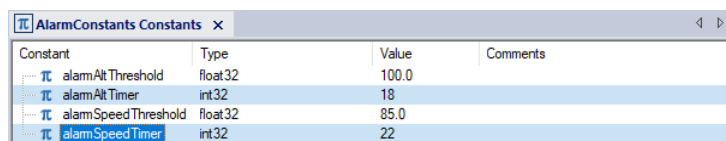
You declare new constants in models at package or model level. The constant values are defined based on predefined types or any type configured in the model and assigned to constant definitions in **Constants Views**. For editing the properties of declared constants, see also [“Editing Constant Properties”](#).

To declare and define new constants

- 1 Display the **Scade View** and expand existing folders.
- 2 Right-click the package folder (or any folder under this package or model) where to declare the constant, and select *Insert > Constant*. A constant is inserted under the Constants folder with a default incremental name and a default type. Alternatively, click **New Constant**  in **Create** toolbar to create a constant in the **Constants View** of the current package.
- 3 Edit the name of the new constant and press Tab-key to switch to type definition. Start typing text and press Ctrl+Space keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to constant name definition.

To edit constant definitions

- 1 Double click any constant in **Scade View** to open the corresponding **Constants View**.



The screenshot shows a window titled 'AlarmConstants Constants'. It contains a table with four columns: Constant, Type, Value, and Comments. The table lists four constants: alarmAltThreshold (float32, 100.0), alarmAltTimer (int32, 18), alarmSpeedThreshold (float32, 85.0), and alarmSpeedTimer (int32, 22). The 'alarmSpeedTimer' row is currently selected.

Constant	Type	Value	Comments
alarmAltThreshold	float32	100.0	
alarmAltTimer	int32	18	
alarmSpeedThreshold	float32	85.0	
alarmSpeedTimer	int32	22	

Figure 3.39: Editing declared constants from Constants View

- 2 Edit the name of the constant in the **Constant** column.
- 3 Edit the definition of the type in **Type** column:
 - For predefined types: enter or select the type name
 - For user-defined types: enter or select the type name
 - For array types: [“Setting Array Types”](#) on page 209
 - For imported types: [“Setting Imported Types”](#) on page 214

Edit **Type** column as follows:

- Select the cell and press **F2** to edit type definition by keyboard
 - Start typing text and press **Ctrl+Space** keys to use auto-completion
 - Select the type in drop-down list or navigate to opened packages
 - Validate by pressing **Tab** key once or **Enter** key twice
- 4 Set the values of the constant directly in the **Value** field.
 - 5 Enter a comment in the **Comments** column if needed.

Once constants are declared, you can add references to them in **Diagram Views**. For user instructions, see [“Creating References to Constants or Sensors”](#) on page 242.

Setting Array/Structure Type Constants

You set constants defined with array types by specifying each value of the data array, by entering values in a dedicated table for two-dimension arrays, or by importing matrix values contained in a textual file. Importing constant value is possible only when the constant type is an array with one or two dimensions applied to a scalar type. When editing the value of a constant used to define the dimension of an array type, all constants of this array type have their dimensions updated accordingly. Such an update is possible only if the dimension expression remains valid. In case the dimension of the array is unknown, the dimension is considered as 1. You set constants defined with structure types by specifying values for each label definition.

- [“To set/edit values of constants of array/structure type individually”](#)
- [“To set/edit values of constants of two-dimensions array type from a table”](#)
- [“To import array values of constants from CSV file format”](#)
- [“To restructure the values of array/structure constants”](#)

When modifying the size of an array, SCADE Suite detects whether it impacts any declared constant and sends a warning. A list of impacted constants displays in the **Messages** tab: double click in the list to allow quick location.

To set/edit values of constants of array/structure type individually

- 1 Expand the constant to display all its definition levels.
- 2 Select each cell corresponding to the definition of the data array/structure in the **Value** column as shown below in [Figure 3.40](#).

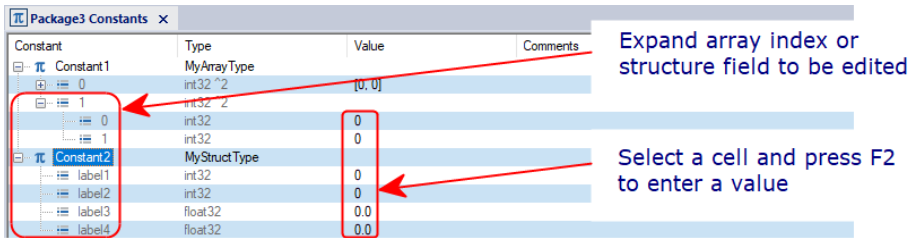


Figure 3.40: Setting the values of a constant of array/structure type

- Enter the value in the format required by the type. Values of char arrays can be entered as string or character by character. For details, see [“To define char array values”](#) on page 210.

To set/edit values of constants of two-dimensions array type from a table

- Expand the array structure of the constant to display a two-dimension data array level (TypeIDⁿ^p) as shown below in [Figure 3.41](#).

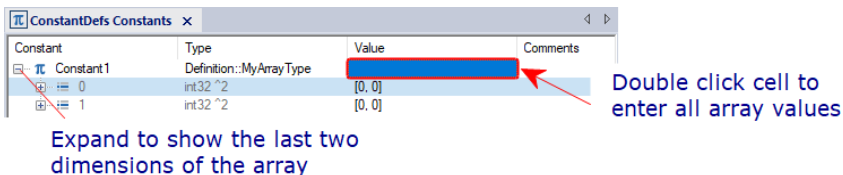


Figure 3.41: Setting the value of two-dimensional array constant in a table

- Double click on Value cell corresponding to the two-dimension array level to open the **Edit Constant Value** window.



Figure 3.42: Specifying values of constants set with array types

- Type the constant values in each cell of the tabular display.
- Click **OK** to validate the values and close the window.

To import array values of constants from CSV file format

- 1 Right-click a constant or a constant subpart in **Constants Views** and select *Import > Constant Value*. You can import values from any array constant (with one or two dimensions) listed in **Scade View**.

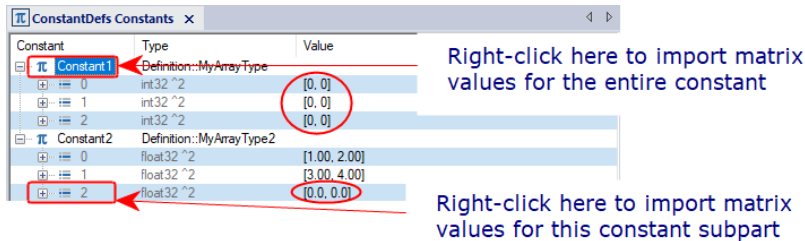


Figure 3.43: Importing values on array constant or subpart of array constant

- 2 Browse in project files from the **Open** window, select the file, and click **Open** to import its content. Make sure you import and define the CSV file content as illustrated hereafter.

Importing array values from CSV format is possible from any single constant selected in SCADA Suite user interface, but it is impossible via the multi-selection of constants.

- When specifying matrix values in CSV format, use the list separator and the decimal symbol as specified in the regional settings of the machine to remain compatible with spreadsheet application format.
- Do not enter any header information or any constant name in the CSV file.

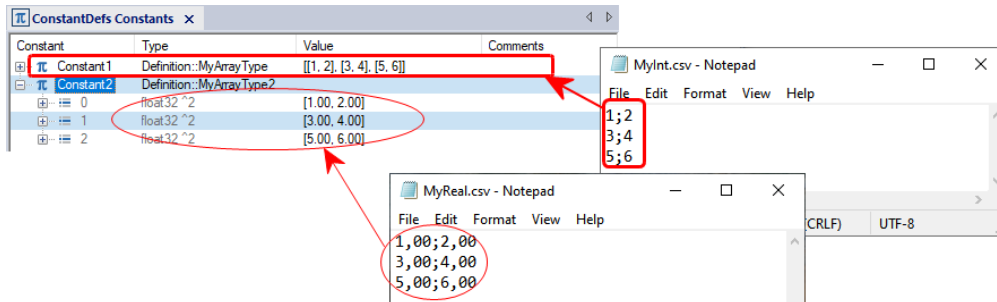


Figure 3.44: Samples of matrix values imported on array constants from CSV file

When the values of array or structured constants are inconsistent with their type or when array/structure constants need to be reorganized, it is possible to edit their values by moving, adding, or suppressing elements directly in **Constants View**.

To restructure the values of array/structure constants

- 1 In **Constants View**, expand the array constant(s) to locate where definitions are highlighted as missing or extra ones.

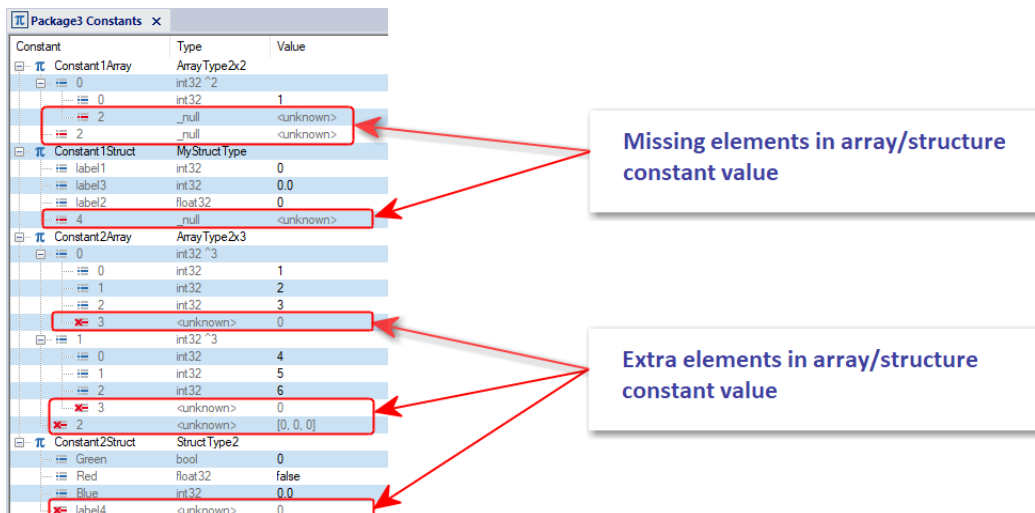


Figure 3.45: Array/structure constants highlighting missing/extra values

- For resolving missing value elements in array/structure constants: select the parent of the missing element(s) and click **New Definition Element**  in **Create** toolbar (or right-click the element and select *Insert > Definition Element*) to add the missing value with expected type. Enter the value of the added array/structure element.
- For resolving extra values in array/structure constants: delete the extra value from the constant.

It is possible to change the order of values moving them up or down by drag and drop directly in **Constants View** or using *Edit > Move Up/Down* menu.

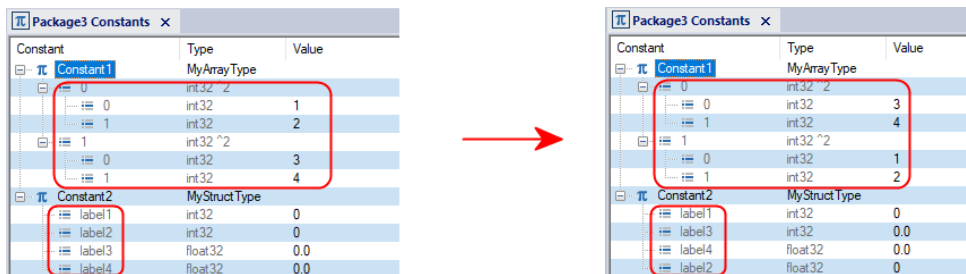


Figure 3.46: Moving order of values by drag and drop in array/structure constants

RELATED TOPICS

- [“Setting Array Types”](#) on page 209
- [“Setting Structure Types”](#) on page 212

Setting Imported Constants with External Code

Imported constants allow to declare constants with values defined externally at compilation time. Constant values are defined in C/Ada files.

To set imported constants

- 1 In **Scade View**, right-click the constant and select *Properties*.
- 2 Select **Declaration** on the left and check **Imported**.

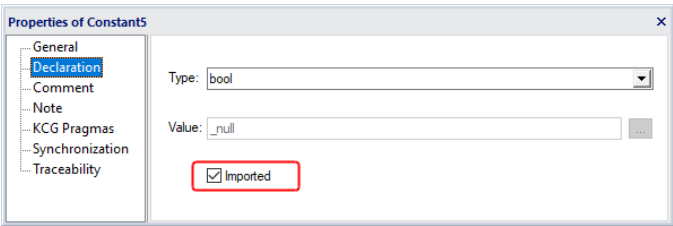


Figure 3.47: Properties window for declaring an imported constant

- 3 Double click the constant in **Scade View** to open the corresponding **Constants View**. The Value cell of the constant is uneditable and displays <Imported> as illustrated in [Figure 3.48](#).

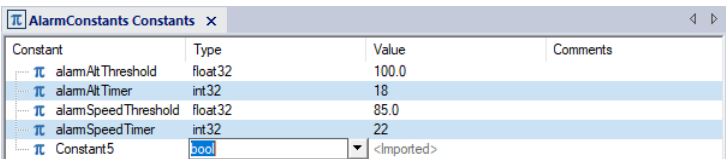


Figure 3.48: Constant View with imported constants

- 4 Select a type from the drop-down list in the Type column.
- 5 Define the values of the imported constant in a C or Ada file as illustrated below. The constants must be named as declared in the **Constants View**.

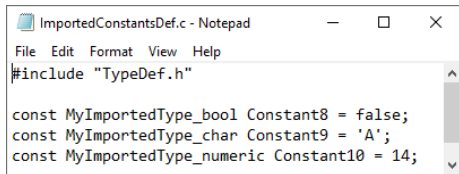


Figure 3.49: C-file defining values of constants configured as imported

- 6 Display the **File View** and right-click the project folder storing external files.
- 7 Select *Insert Files*, browse for file location and click **Open** to select the C or Ada file containing imported constant definitions.
- 8 Right-click the file you inserted in the **File View** and select *Properties*.
- 9 Select **Simulation** on the left and set required properties on the right.

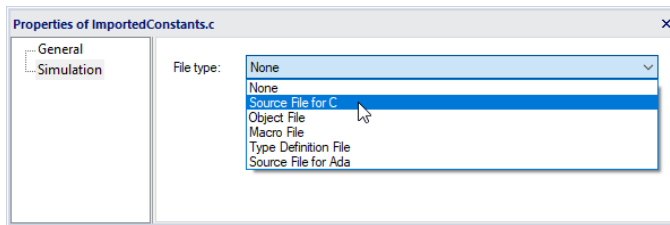


Figure 3.50: Setting C/Ada source file

- 10 Specify **Source File for C** or **Source File for Ada** in the **File type** list.

Note

If you need to declare the imported constant as macro, you should define an H-file and declare it as Macro File in the File type drop-down list. About other file types, see [“To edit the Simulation properties of a file”](#) on page 432

- 11 Close the window or click anywhere in workspace to validate.

RELATED TOPICS

- [“Declaring and Editing Imported Operators”](#) on page 194
- [“Setting Imported Types”](#) on page 214

Setting Imported Constants with Imported Types

You can declare imported constants with imported types whose values are defined externally at compilation time. Constant values are defined in C/Ada files which must call the file(s) where imported types are defined.

Constants cannot be imported if defined by imported types that are constrained.

To set constants with imported type

- 1 Display the **Scade View** and expand existing folders to locate the constant to set as imported.
- 2 Right-click the constant and select *Properties*.
- 3 Select **Declaration** on the left and check **Imported**.

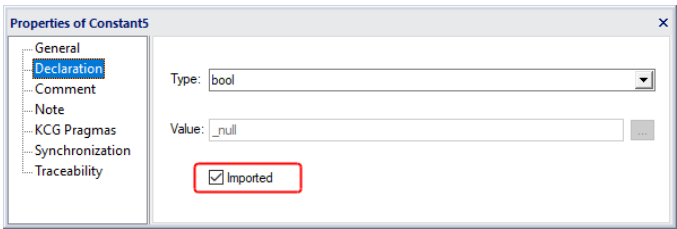


Figure 3.51: Properties window for declaring an imported constant

- 4 Double click the constant in the **Scade View** to open the corresponding **Constants View**. The Value cell of the constant is uneditable and displays `<Imported>` as illustrated below.

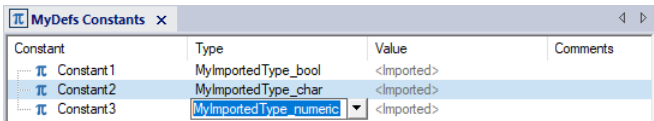
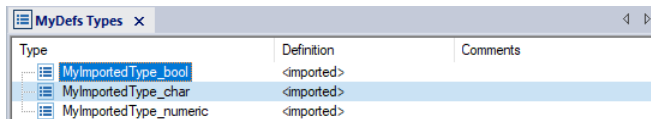


Figure 3.52: Constant View with imported constants

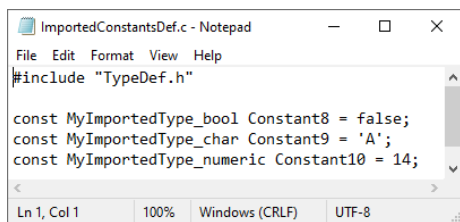
- 5 Select an imported type from the drop-down list in the Type column. Imported types are declared as illustrated below.



Type	Definition	Comments
MyImportedType_bool	<imported>	
MyImportedType_char	<imported>	
MyImportedType_numeric	<imported>	

Figure 3.53: Imported types

- 6 Define the values of the imported constant in a C or Ada file as illustrated below. The constants must be named as declared in the **Constants View**.



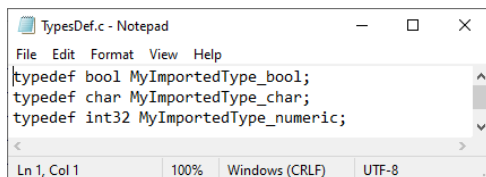
```
File Edit Format View Help
#include "TypeDef.h"

const MyImportedType_bool Constant8 = false;
const MyImportedType_char Constant9 = 'A';
const MyImportedType_numeric Constant10 = 14;
```

Figure 3.54: C-file defining values of imported constant configured with imported types

Warning

The above file to be used as source file of the constant must call the definition file corresponding to the selected imported type illustrated in [Figure 3.55](#). This H-file contains the definition of imported types and must be included in the project to define types as explained in [“Setting Imported Types”](#) on page 214.



```
File Edit Format View Help
typedef bool MyImportedType_bool;
typedef char MyImportedType_char;
typedef int32 MyImportedType_numeric;
```

Figure 3.55: File containing definitions of imported types

- 7 Display the **File View** and right-click on the project folder where you store external files.
- 8 Select *Insert Files*, browse for file location in the project directory and click **Open** to select the C or Ada file containing imported constant definitions.
- 9 Right-click the file you inserted in the **File View** and select *Properties*.
- 10 Select **Simulation** on the left and set required properties on the right.

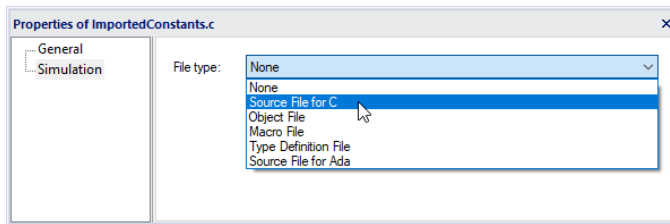


Figure 3.56: Setting C/Ada source file

- 11 Specify **Source File for C** or **Source File for Ada** in the **File type** drop-down list.

Note

About other file types, see [“To edit the Simulation properties of a file”](#) on page 432

- 12 Close the window or click anywhere in workspace to validate.

RELATED TOPICS

- [“Declaring and Editing Imported Operators”](#) on page 194
- [“Setting Imported Types”](#) on page 214

Defining Sensors

The sensors you need to use throughout the model must be declared explicitly based on types declared in the model or using anonymous types defined textually (see [“Defining Data Types”](#) on page 207). Sensors you define in SCADE Suite are organized in **Sensors Views** located in packages or at project root level.

Warning

Sensors can only be consumed. They can never be produced in data flows.

Declaring New Sensors

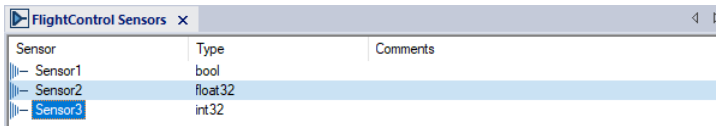
You declare new sensors in models within the **Scade View**. The sensors are defined according to the types configured in the model and assigned to the sensor definition in **Sensors Views**.

To declare and define new sensors

- 1 Display the **Scade View** and expand existing folders.
- 2 Right-click the package folder (or any folder under this package or model) where to declare the sensor, and select *Insert > Sensor*. A sensor is inserted under the **Sensors** folder with a default incremental name and a default type. Alternatively, click **New Sensor**  in **Create** toolbar to create a sensor in the **Sensors View** of the current package.
- 3 Edit the name of the new sensor and press Tab-key to switch to type definition. Start typing text and press **Ctrl+Space** keys to use auto-completion and select type in drop-down list. Press Tab-key again to return to sensor name definition if needed.

To edit sensor definitions

- 1 Double click any sensor in **Scade View** to open the corresponding **Sensors View**.



Sensor	Type	Comments
Sensor1	bool	
Sensor2	float32	
Sensor3	int32	

Figure 3.57: Editing declared sensors from Sensors View

- 2 Edit the name of the new sensor in the Sensor column.
- 3 Edit the definition of the type in Type column:
 - Select the cell and press F2 to edit type definition by keyboard.
 - Start typing text and press Ctr l+Space keys to use auto-completion.
 - Select the type in drop-down list or navigate to opened packages.
 - Validate by pressing Tab key once or Enter key twice:
 - For predefined types: enter or select the keyword.
 - For user-defined types: enter or select the type name.
- 4 Enter a comment in the Comments column if needed.

Note

Once sensors are declared, you can add references to them in the Diagram Views of the model. For user instructions, see [“Creating References to Constants or Sensors”](#) on page 242.

Setting Design Instances in Diagram Views

When designing models in **Diagram Views**, you choose and set calls or references to design elements. Calls to operators and references to inputs, outputs, variables, constants, probes are declared in the model via [“Model Editing Views”](#). If needed, you can display various types of information about design elements set in **Diagram Views** (see [“Enabling Display Options in Diagram Views”](#) on page 143).

Tips

- Create several items in a row in the active Diagram View by pressing SHIFT and clicking the item in toolbars or the Shortcuts Window. Cancel this mode with a right-click anywhere in the View.
 - Select multiple inputs/outputs from an operator interface in the Scade View by holding the **Ct r l**-key down to drag and drop them into Diagram Views.
 - Select multiple elements in Diagram Views by holding the **Ct r l**-key and moving the mouse to intersect with the selection rectangle of elements.
-

To design a model in **Diagram Views**, you can execute the following tasks:

- [“Creating Operator Calls”](#)
- [“Creating Calls to Predefined Operators”](#)
- [“Creating Predefined Operator Symmetric Calls”](#)
- [“Adding/Removing Predefined Operator Inputs”](#)
- [“Composing/Decomposing Structured Types”](#)
- [“Substituting Operator and Variable Calls”](#)
- [“Creating References to Inputs/Outputs”](#)
- [“Creating References to Local Data”](#)
- [“Creating Contracts: Assume/Guarantee”](#)
- [“Creating Terminators”](#)
- [“Creating Lasts of Variables and Signals”](#)
- [“Setting Default Values on Variables”](#)
- [“Connecting Data Flow Elements”](#)
- [“Connecting Multiple Data Flow Elements”](#)
- [“Defining Value of Hidden Inputs on Operators”](#)
- [“Encapsulating Design in Operators or States”](#)

Creating Operator Calls

You create operator calls in the Diagram Views. You can perform this task using the Scade View or toolbars.

Note

To iterate operator calls, see [“Designing with Array Iterators”](#) on page 378.

To create operator calls from the Shortcuts Window

- 1 Open the shortcut folder of the operator to create from **Shortcuts Window**.
- 2 Click the operator icon and click again in the **Diagram View** to create a call to the selected operator where the mouse cursor points.

Note

You can select any of the predefined operators from toolbars added in the Shortcuts Window or user-defined operators set as favorites in the Shortcuts Window. See [“Managing Favorites and Using Shortcuts”](#) on page 131.

To create operator calls from Scade View

- Select an operator in **Scade View**. then drag and drop it in the **Diagram View** on an empty position.

Note

If you click or drop over an existing operator call in the Diagram Views, you substitute the existing one with the new one. See [“Substituting Operator and Variable Calls”](#) on page 238.

If you drag and drop operator calls between Diagram Views, SCAD Suite moves the selected operator to its new location, but unconnected, and removes it from the previous one including all connecting wires. To keep the call on its original location and copy it in another Diagram View, hold the **Ctrl**-key while dragging and dropping the selected item.

Creating Calls to Predefined Operators

In addition to the user-defined operator calls defined in projects (see [“Declaring Model Operators”](#) on page 160), you can use a set of predefined operators and constructs delivered with SCADE Suite to design the model behavior. Predefined operators are available from dedicated toolbars (see [Predefined operator toolbars](#)).

Note

Predefined operator calls and predefined constructs are described in Chapter 4 about [“Designing with Predefined Operators and Constructs”](#)

To insert calls to predefined operators

- 1 Click the appropriate operator icon from predefined operator [toolbars](#). You may click operators from toolbars set in Shortcuts Window.
 - 2 Click in the **Diagram View** to add an operator instance on the designated position.
-

Tip

Hold Shift-key down and click on predefined operator icon to create multiple instances of the selected operator. Right-click to leave the creation mode.

If you want to modify the number of inputs set on any predefined operator call in the model, see [“Adding/Removing Predefined Operator Inputs”](#) on page 235.

Creating Predefined Operator Symmetric Calls

You can transform some of the predefined operator calls you created in **Diagram Views** into symmetric instances as required for design purposes. Symmetric calls allow to invert either operator inputs, or top and bottom parameter entries.

To create symmetric calls for predefined operators

- Right-click the operator to be transformed in a **Diagram View** and select *Set symmetric*. The transformation of a predefined operator instance into symmetric instances is also possible from the **Use** tab of operator **Properties** window as explained in [“Editing Operator Call Properties”](#) on page 448.

To unset symmetric calls

- Right-click the operator to be transformed in a **Diagram View** and select *Unset symmetric*.

Adding/Removing Predefined Operator Inputs

Once you created predefined operator calls (see [“Creating Calls to Predefined Operators”](#) on page 234), it is possible to add/remove input pins from the following operators:

- Plus
- Multiplication
- AND
- OR
- XOR
- Bitwise AND
- Bitwise OR
- Sharp (#)
- If... then ... else
- Case
- Init (->)
- When
- Followed by (fby)
- Previous (pre)
- Merge
- Data structure
- Data array
- Scalar to vector
- Concatenation
- Dynamic projection
- Projection
- Assign of structured element (with)

To add inputs on predefined operators

- 1 Select *Operator > Add an Input to a Predefined Operator*.
- 2 Click on a predefined operator in the **Diagram View**. A new input or hidden input is added to the selected operator call at the last position. For some predefined operators, the corresponding outputs are added.

To remove inputs from predefined operators

- 1 Select *Operator > Remove an Input from a Predefined Operator*.
- 2 Click on the input pin to remove from the predefined operator in the **Diagram View**. The selected input is removed from the operator call:
 - If removing inputs implies removing the corresponding outputs, SCADE Suite Editor does it automatically.
 - If the default number of inputs is reached, input removal becomes impossible and an illegal operation message appears in the **Status** bar.
 - If clicking on operator symbol instead, the last pin is removed.

Tips

Press Shift-key down while selecting the appropriate menu to enable the addition or removal of several inputs in a row. Click right to cancel multi-edition mode.

You can use the Properties of predefined operators to add/remove input on diagram symbols. For more details, see [“Editing Use Properties of Operator Calls”](#) on page 448.

Composing/Decomposing Structured Types

You can compose or decompose input and output flows defined by array or structure data types based on Make, Data Array, and Flatten operator calls.

Flow composition/decomposition with respect to data typing is a design shortcut. Instead of creating Data Array, Flatten, or Make operator calls and setting each with relevant data types, you can instantiate them directly from input and output flows typed as arrays or data structure.

You create Make, Data Array, or Flatten operators in diagrams from the Structure/Array toolbar and set them as detailed in [“Setting and Designing Make or Flatten”](#) on page 363 and [“Setting and Designing Arrays by Enumeration”](#) on page 368.

To compose typed flows from output variables

- 1 In **Diagram Views**, right-click an output, a probe, or an output local variable defined with a structured type (array or structure).

Tip

Output local variables can be instantiated as follows:

- Local variables set as output from Out option in Use properties (see [“Editing Local Variable Properties”](#) on page 459).
- Hold C t r l-key down, then drag and drop local variables from Scade View in diagrams to create output local variable ([“Creating References to Local Data”](#) on page 240).

- 2 Select *Compose*:

- A **Make** operator instance is connected to outputs of structure type.
- A **Data Array** operator instance is connected to outputs of array type.

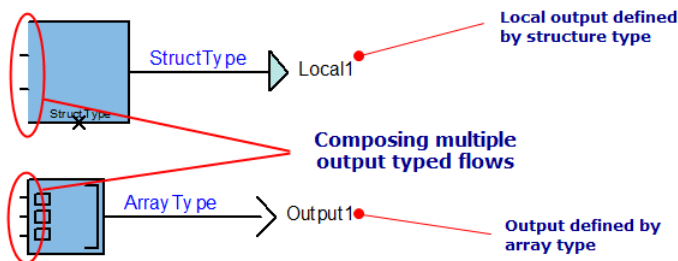


Figure 3.58: Composing types from output variables

To decompose typed flows from input variables

- 1 In **Diagram Views**, right-click an input or an input local variable defined with a structured type (array or structure).

Tip

Input local variables can be instantiated as follows:

- Local variables set as input from In option in Use properties, see [“Editing Local Variable Properties”](#) on page 459.
- Drag and drop local variables from Scade View directly in diagrams to create input local variable, more details in [“Creating References to Local Data”](#) on page 240

- 2 Select *Decompose*. A **Flatten** operator instance is connected to the selected input.

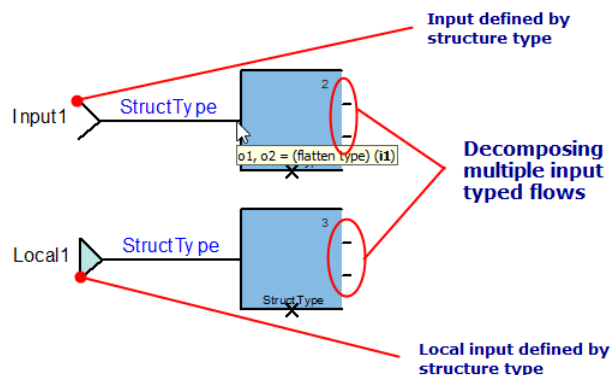


Figure 3.59: Decomposing types from input variables

Substituting Operator and Variable Calls

You can substitute any operator or variable (*i.e.*, inputs/outputs, hidden inputs, local variables, probes, constants, textual expressions, signals, sensors, and terminators) call specified in Diagram Views with another operator or variable defined in the model.

To substitute operator and variable calls from the Scade View

- 1 Select an operator or variable in **Scade View**.
- 2 Drag in the **Diagram View** and drop on the call to substitute with.

To substitute operator and variable calls from toolbars

- 1 Open the shortcut folder of the operator to substitute.
- 2 Click the operator icon in the **Shortcuts Window**.
- 3 Click on the call to substitute in the **Diagram View**.

Note

Operator call substitution is possible with higher-order operators defined in Diagram Views. The substituted operator keeps the higher-order construction.

Creating References to Inputs/Outputs

From the **Scade View**, you can create as many references as you need in the model to inputs, hidden inputs, or outputs defined for operator interface. You create references to any constant.

To create references to operator interface items

- 1 Expand the Interface folder of the operator from **Scade View**.
- 2 Select an input, a hidden input, or an output.
- 3 Drag the selected item and drop it in the **Diagram View**.

For defining the operator interface first, see [“Declaring Inputs/Outputs”](#) on page 162 and [“Declaring Hidden Inputs”](#) on page 163.

Creating References to Local Data

You can create as many references as you need in the model to the local variables, probes, or signals declared in the model. Local variables references can be set as input or output data. Probe references are always used as output data. You can create local data references in the following ways:

- Dragging and dropping variables in the operator design when previously declared as local data in the model from **Scade View**.
- Inserting local data directly on the operator design from **Create** toolbar: SCADe automatically sets the variable declaration in **Scade View**.

For details about local data declaration, see [“Declaring Variables”](#) on page 164, [“Declaring Probes”](#) on page 166, or [“Declaring Signals”](#) on page 167.

To create local variable as input references

- Select a local variable from Locals folder in **Scade View** and drag and drop it in the **Diagram View**. The variable symbol inserted in the design displays as follows: . Alternatively, hold the CTRL - key down, click **New Local Variable**  in **Create** toolbar, and click again in **Diagram View** to insert an input reference of the variable.

To create local variable as output references

- Hold CTRL-key down and select a local variable from Locals folder in **Scade View**. Drag the selected variable and drop it in the **Diagram View**. The variable symbol inserted in the design displays as follows: . Alternatively, click **New Local Variable**  in **Create** toolbar, and click again in **Diagram View** to insert an output reference of the variable.

To create references to probes

- Select a probe from Locals folder in **Scade View** and drag the selected probe and drop it in the **Diagram View**. Alternatively, click **New Probe**  in **Create** toolbar, and click again in **Diagram View** to insert an output instance of the variable. It is possible to create probes automatically typed directly from diagrams: right-click an existing connection and select *Insert Probe*. A new probe is created, connected, and typed according to the selected connection.

Signals can be used as a signal emission, or as a signal presence test that produces a Boolean data flow. For more details about signal syntax and principles, consult 3.4 [“Pure Signals”](#) in Chapter 3 / of *Scade Language Primer*.

To create signal references to test presence

- Select a signal from Locals folder in **Scade View** and drag the selected signal and drop it in the **Diagram View**. The signal symbol inserted in the design displays as follows: . Alternatively, hold CTRL - key down, click **New Signal**  in **Create** toolbar, and click again in **Diagram View** to insert a signal presence test.

To create signal emissions

- Hold CTRL-key down and select a signal from Locals folder in **Scade View**. Drag the selected signal and drop it in the **Diagram View**. The signal symbol inserted in the design displays as follows: . Alternatively, click **New Signal**  in **Create** toolbar, and click again in **Diagram View** to insert an emission of the signal. Design a conditional emission by connecting the condition variable to the upper connector of the signal symbol.

Creating References to Constants or Sensors

From **Scade View**, you can create as many references as you need in the model to constants or sensors defined in models.

To create references to constants or sensors

- Expand the Constants (resp. Sensors) folder from the appropriate package in **Scade View**, select a constant (resp. a sensor), then drag and drop it in the **Diagram View**.

Creating Contracts: Assume/Guarantee

You can create as many Assume or Guarantee contracts as needed in **Diagram Views**. They allow to introduce environment information in SCADÉ Suite models and accelerate verification by reducing the input variation domains. The expression that defines them is defined textually. Contracts are distinguished as follows:

- assume observes inputs and the past of the outputs
- guarantee observes inputs and outputs

To create assume/guarantee contracts

- 1 Click **New Assume** ✓ or **New Guarantee** ▼ in **Create** toolbar.
- 2 Click in the **Diagram View** to insert the element where needed.
- 3 Enter the expression in the **Diagram View**. The expression starts with the contract name as follows: Name : expression. For details, see 6.1 [“Scade Contracts: Assume and Guarantee Observers”](#) in *Scade Language Primer*. Edit expressions by clicking twice the element, by selecting the element and pressing F2, or by opening the **Properties** window of the element.

Creating Terminators

You can create as many terminators as you need in any **Diagram View**. They explicitly end a flow in the model and ignore the output from an operator.

Note

Terminators are not defined in Scade View.

To create terminators

- 1 Click **New Terminator**  in **Create** toolbar.
- 2 Click in any **Diagram View** where needed.

Creating Lasts of Variables and Signals

You can create as many lasts as you need in diagrams. They represent a reference to the last value of any variable (inputs/outputs, hidden inputs, local variables, probes) or signal.

To create last of variable and signals directly in diagrams

- 1 Select an input, an output, a hidden input, a local variable, a probe, or a signal declared in **Scade View**.
 - 2 Click **New Last**  in **Create** toolbar.
 - 3 Click again in diagrams where needed. The context of the diagram must be a valid one.
-

Tips

You can create last references from variable references in diagrams:

- Right-click the variable and select *Last Reference*.
 - Open the Properties of the variable reference, display the Use properties, and check Last.
-

You can also assign last initialization values to variables instantiated in the model design. Last initialization values are allowed on inputs, hidden inputs, outputs, local variables, and probes. You cannot set such values on signals.

To set last initialization values on variables

- 1 In the **Scade View**, right-click any variable and select *Properties*.
- 2 Select **Declaration** on the left and set required properties on the right.
- 3 Enter the initialization value in the **Last** field. The last value is initialized at the first instant.

Note

Any variables set with last initialization values are indicated in models as highlighted in the figure below.

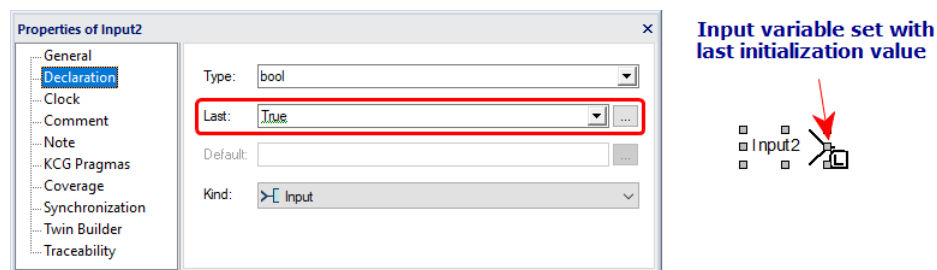


Figure 3.60: Initializing last value on variables

Note

If you need to enter a value as a long expression, you can open the Edit Value window by clicking the browsing button corresponding to the field.

- 4 Click outside the **Properties** window to validate the changes.

Setting Default Values on Variables

You can assign default values to the variables instantiated in the design of models. Default values are allowed on hidden inputs, outputs, local variables, and probes. Inputs and signals cannot have default values.

To set default values on variables

- 1 In the **Scade View**, right-click any variable and select *Properties*.
- 2 Select **Declaration** on the left and set required properties on the right.
- 3 Enter the default value in the **Default** field.

Note

Any variables set with default values are indicated in models as highlighted in the figure below.

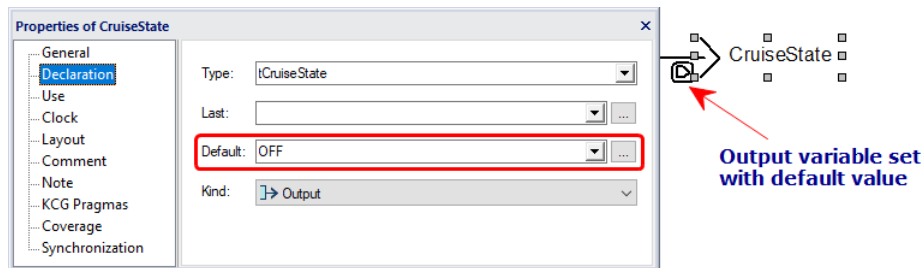


Figure 3.61: Setting default value on variables

Note

If you need to enter a value as a long expression, you can open the Edit Value window by clicking the browsing button corresponding to the field.

- 4 Click outside the **Properties** window to validate the changes.

Connecting Data Flow Elements

Designing models requires creating connections between data flow design elements in **Diagram Views**. You can connect elements one by one or create multiple connections starting from the same source. Automated assistance is available when creating connections with predefined operators.

Note

For pin name or pin rank based connecting assistance, see [“Connecting Multiple Data Flow Elements”](#) on page 248. For connecting control flow elements in State Machines, see [“Drawing and Editing State Transitions”](#) on page 172.

To create design element connections one by one

- Click the connecting point of the source (crosshair cursor) and click over the symbol of the destination operator to connect with the first free input.

Tip

Move towards a pin on operator symbols to connect with closest pin by magnetism.

To create multiple connections consecutively from same source

- Hold **Ctr L**-key, click the connecting point of the source (crosshair cursor) and click over the symbol of each destination operator one after another.

Tip

Right-click to stop multiple connection mode or press **Esc** key.

When creating a connection to predefined operators, if there is no free pin on the symbol and the number of operator inputs can be changed, a new input pin is automatically added.

TYPE PROPAGATION ALONG DATA FLOW CONNECTIONS

In most cases, SCADE Suite propagates types automatically along connecting wires in the whole design. However, while editing the design and modifying connections between elements, you may not recover all typing information. In such cases, it is possible to force the propagation of internal variable (identified as `_L` in diagrams) types by computing types.

Tip

To clean the model from unused or incorrectly typed internal variables before code generation, select *Project > Compute Types* menu to resolve type propagation on all connecting wires (internal variables).

Connecting Multiple Data Flow Elements

You can connect easily multiple data flow input/output variables to operators based on the pin names or pin ranks of the operator to connect.

Tip

Position the mouse cursor over the input/output pins of any operator on Diagram Views to display tooltips that show the formal equation of the operator with the name of the selected operator pin in bold.

You can use either of the following methods:

- [“To draw multiple connections by name”](#)
- [“To draw multiple connections by rank”](#)

Tip

Use also connection by rank to connect pairs of operator.

To draw multiple connections by name

- 1 Select set of input or output elements to connect as illustrated in [Figure 3.62](#).

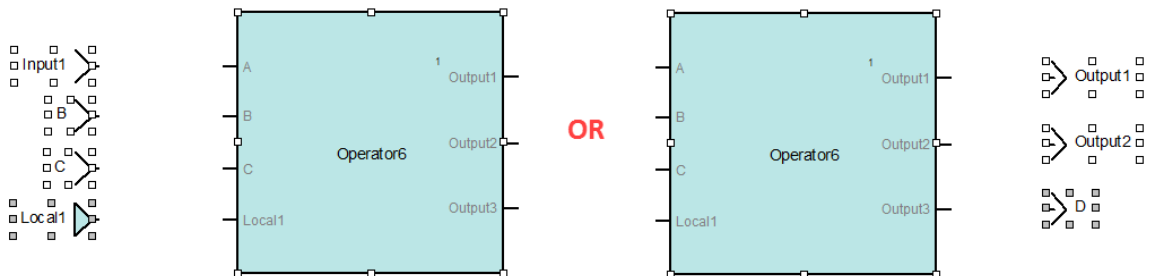


Figure 3.62: Multiselection to connect several inputs/outputs to an operator by name

- 2 Right-click any of the selected elements and select *Connect > By Name* (shortcut: **Ctrl+Shift+M**). Only the element(s) with matching input/output names are connected to the selected element.

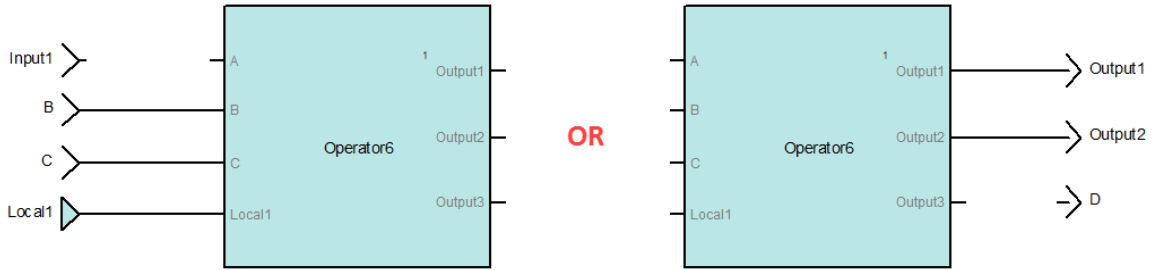


Figure 3.63: Elements connected by name

To draw multiple connections by rank

- 1 Select all the elements to connect as illustrated in [Figure 3.64](#).

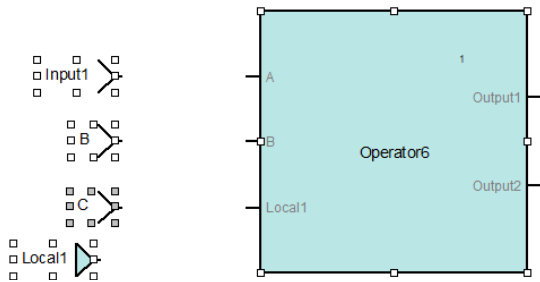


Figure 3.64: Multiselection to connect several inputs to an operator by rank

- 2 Right-click on any of the selected elements and select *Connect > By Rank* (shortcut: **Ctr l+Shif t+R**). The element(s) are connected to the operator pins of the selected element by pin rank order.

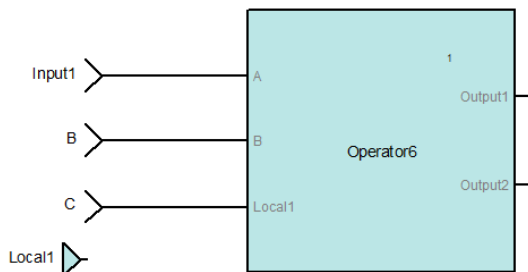


Figure 3.65: Connecting elements by rank

Defining Value of Hidden Inputs on Operators

Whether you define hidden inputs in user operator interfaces (see [“Declaring Hidden Inputs”](#) on page 163), you need to define the value of such inputs in the design. You can choose to define some values directly on operators symbols in **Diagram Views** or in Properties, or define all hidden input values of a user operator at once from a single view. It is also possible to define hidden inputs by connecting graphically an input variable to hidden input pin on operators. If you connect any input wire to the hidden input pin, any value previously set (if any) is replaced on the corresponding pin.

To define hidden input values individually on Diagram Views

- 1 Display the **Diagram View**.
- 2 Select the operator call and click above the hidden input mark on the symbol to activate an editing area (display of text cursor).
- 3 Enter the value of the hidden input. Alternatively:
 - If the value of the hidden input is a constant declared in the model, you can drag and drop the constant directly onto the hidden input area.
 - Define hidden inputs from operator properties as described in [“Editing Parameters of Operator Calls”](#) on page 451.

While defining hidden inputs on operators, you can enable editing help to select from the list of names available in the current project (see [“To complete text automatically”](#) on page 262). Additionally, you get syntactic assistance while typing expressions.

Note

- If the expression text is syntactically incorrect, it is underlined in red.
 - If the expression text has references to unknown object, it is underlined in green.
-

To define hidden input values of user operators from parameter table view

- 1 In **Diagram View**, right-click any user operator defined with multiple hidden inputs or a user operator with multiple hidden inputs and declared under a Mapw[i], a Mapfoldw[i], or a Boolean Activate and select *Open Hidden Values View*.
- 2 Specify the values of all hidden inputs from the displayed view. Expand the structured hidden inputs (if any) to specify the values. A hidden parameter table item is added in **Scade View**.

Hidden	Type	Value
Hidden1	int32	1
Hidden2	int32	3
Hidden3	int32 ^3	
0	int32	_null
1	int32	_null
2	int32	_null
Hidden4	{label1 : int32, label2 : bool}	
label1	int32	_null
label2	bool	_null

Figure 3.66: Specifying hidden values of user operators from parameter table view

- 3 Go back to the diagram where the user operator is represented. A rectangle on the edge of the user operator represents the view where hidden input values are set.

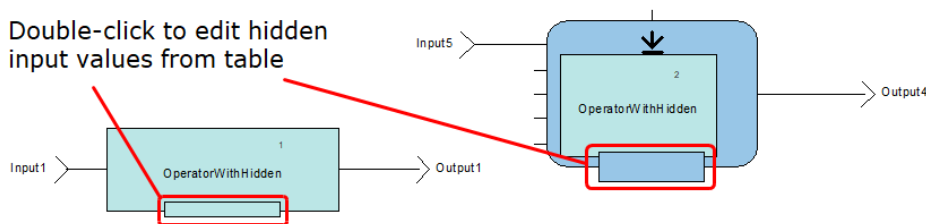
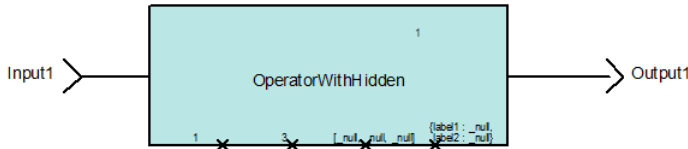


Figure 3.67: User operator with symbol for accessing hidden parameter table view

- 4 From the user operator representation, the following action is possible:
 - Double click the rectangle to display the view and edit values.

- To remove the rectangle and display values on the operator symbol, right-click the operator displaying the rectangle and select *Delete Hidden Values View*.



In case the hidden values rectangle is removed, it is always possible to reopen the table view and edit existing hidden input values.

Encapsulating Design in Operators or States

You can select several design elements and their connections in Diagram Views and group them into a new encapsulation operator (available for equations only), or new encapsulation state and state machine. When encapsulating part of a design, SCADE Suite automatically updates the model with respect to the graphical and syntactical representation of the model.

Note

All design elements to be encapsulated must have the same parent.

To encapsulate design flows into operator

- 1 In the **Diagram View** where to encapsulate, click and draw a selection area with the mouse to select the equations to encapsulate. You must select at least one equation. Other graphical objects, such as state machines, if blocks, when blocks and assertions, are ignored.
- 2 Select *Operator > Pack > in Operator* from the main menu or select *Pack > in Operator* from the contextual menu to create an operator containing the selection.
- 3 Name the operator if needed and validate.

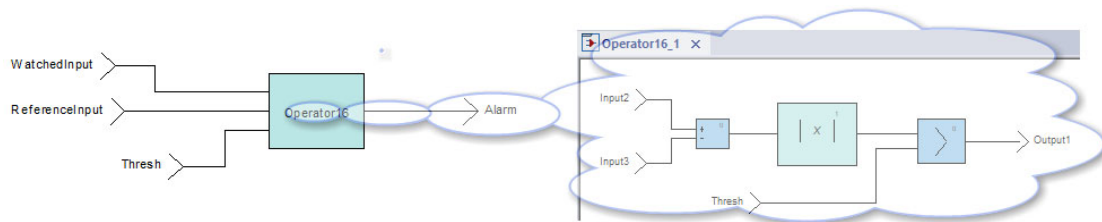


Figure 3.68: Result after packing in new operator

To encapsulate design flows into state and state machines

- 1 In the **Diagram View** where to encapsulate, click and draw a selection area with the mouse to select the elements to encapsulate. If you select at least one equation, one assertion, or one state machine, SCADE Suite creates a new state within an enclosing state machine.
- 2 Select *Operator > Pack > in State* from the main menu or select *Pack > in State* from the contextual menu to create a state (or state machine when required) containing the selection.

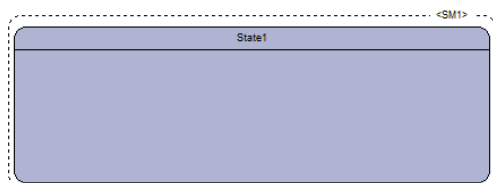


Figure 3.69: Results after packing equations in new state

Note

It is also possible to encapsulate multiple state machines into a new state following the same instructions as shown above for data flow.

Setting and Defining Equation Sets

You can use equation sets to assemble design elements graphically in diagrams for many purposes. Grouping equations enables to globally comment, annotate, or trace to requirements a whole equation set and not only each equation. An equation set is identified by a name and can have its own graphical style.

To create equation sets from diagram selection

- 1 Display the graphical diagram of an operator and select the elements to be included in the equation set.
- 2 Right-click on selection, select *Add to Equation Set > New Equation Set*, and rename it.

To set and define new empty equation sets

- 1 Expand operator description in **Scade View**.
- 2 Right-click a diagram, select *Insert > Equation Set*, and rename it.
- 3 Display the graphical diagram of the selected operator and select the elements to be included in the equation set.
- 4 Right-click on selection, select *Add to Equation Set*, and select the name of the equation set previously created.

To check and manage the list of elements present in equation sets

- 1 Right-click an equation set in **Scade View** and select *Properties*.
- 2 Select **Equation Set** to display the list of all elements in the selected equation set. Double-clicking on an element in the list allows to locate it.
- 3 Select an element or a multiselection of elements in the list and click **Remove** to remove the selection from the definition of the equation set.

To locate equation sets in diagrams

- Double click the equation set in **Scade View** to locate selected elements in diagrams. Press Esc key to remove colored highlight.

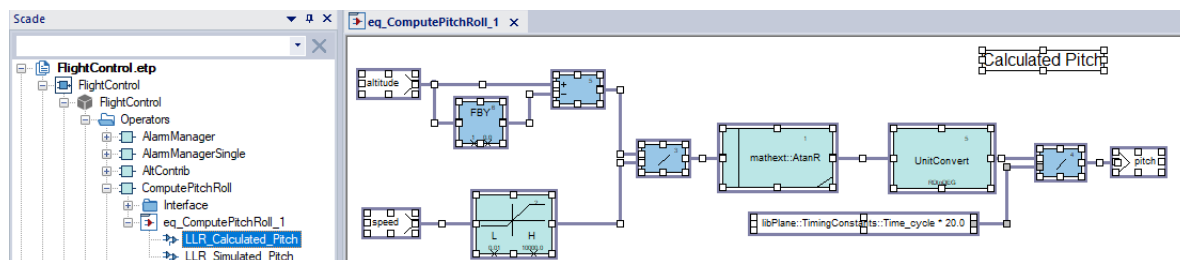


Figure 3.70: Locating equation sets in diagrams

To edit existing equation set definition

- 1 Right-click an element or several ones in an equation set to edit as follows:
 - Select *Remove from Equation Set*, and select the name of the equation set from which selected element(s) are removed.
 - Select *Add to Equation Set*, and select the name of the equation set to which selected element(s) are added.

Delete an equation set item in **Scade View** to remove an equation set definition. By default, the style of equation sets (“Equation Set” style) is transparent. Such default display can be customized using styles to facilitate the graphical identification of equation sets in diagrams. For details, see [“Creating and Modifying Styles”](#) on page 134.

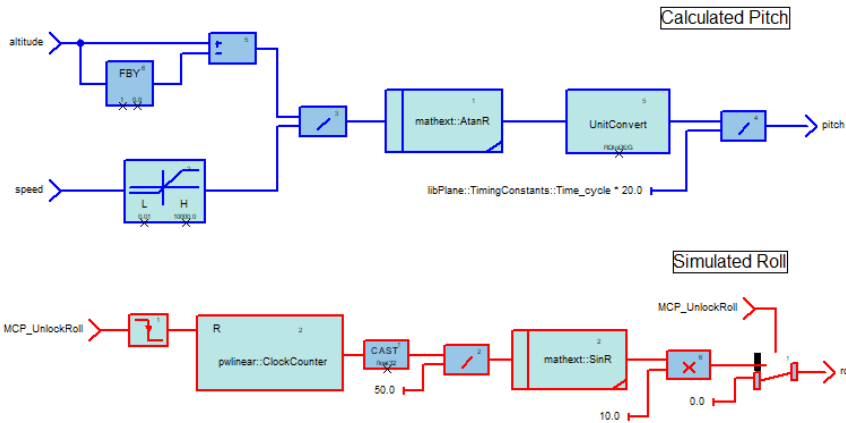


Figure 3.71: Applying different styles on equation sets

RELATED TOPICS

- [“Changing Design Element Layout”](#) on page 482
- [“Adding and Editing Notes”](#) on page 484

Setting Operator Parameterized by Size

When designing models, you may need to reuse operator instances that process array operations by applying different array dimensions. This is possible by creating operator as usual, and, in addition to input and output declarations, adding a list of size parameters. After instantiating such operator in the model, you can specify which dimensions to apply on their instances.

If a parameterized operator is defined in a library, and the order of size parameters is modified while editing the library, then it is not possible to be aware of that change in any model using that library.

To set operator parameterized by size

- 1 Select the operator in **Scade View** and click **New Parameter**  in **Create** toolbar. Repeat for each additional parameter to insert in the operator declaration.

Tip

You can create new parameters on operators contextually in Scade View. Right-click the operator item or the `Interface` folder and select *Insert > Parameter*.

- 2 Name the parameter directly in the tree.
- 3 Display the appropriate graphical diagram and instantiate the operator.
- 4 Right-click the operator instance and select *Properties*. The instance has the following name `OperatorName<<_null>>`.
- 5 Select **Size Parameters** on the left and set required properties on the right. The order of size parameters is defined by their order in **Scade View**.

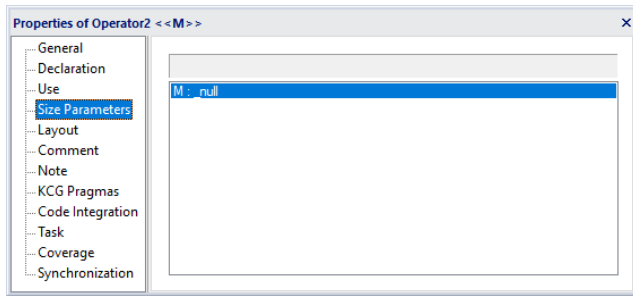


Figure 3.72: Setting value of size parameters

- 6 Specify the value for each parameter. Values must be strictly positive integers. Press F2 to enter in editing mode or press Esc to exit editing mode.
- 7 Connect the inputs and outputs of the operator to the design.

Warning

Parameterized operators cannot be defined as root operators in models.

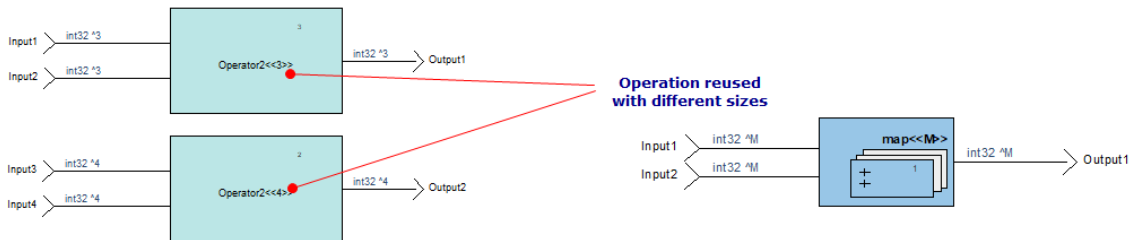


Figure 3.73: Example of parameterized operators in model design

Note

Size parameters can be used for type definition of all locals of the operator.

Designing Model with Textual Definitions

While designing models and declaring various design constructions, you may need to complete the application descriptions and design with the help of textual definitions compliant with Scade language constructs. SCAD Suite Editor provides various accesses to textual definitions:

- Text diagrams created under operator declarations
- Text files inserted in projects as definition references
- Textual expressions embedded in data and control flows

Note

For complete details about the Scade language, consult [“Language Reference Manual Overview”](#) in *Scade Language Reference Manual*.

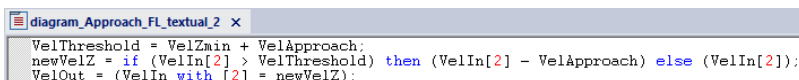
- [“Defining Operators with Textual Diagrams”](#)
- [“Creating Textual Expressions”](#)
- [“Defining Textual States or Actions”](#)
- [“Managing Instance Names Textually”](#)
- [“Creating Instances from Textual Files”](#)

Defining Operators with Textual Diagrams

You can complete the design of operators with text diagrams that contain definitions in Scade textual language. You describe the operator textually in a **Textual View** and declare its interface and locals in the **Scade View**.

Note

Scade language keywords appear in a different color than other words to facilitate reading. You can edit all of the textual diagram content.



```
diagram_Approach_FL_textual_2 X
VelThreshold = VelZmin + VelApproach;
newVelZ = if (VelIn[2] > VelThreshold) then (VelIn[2] - VelApproach) else (VelIn[2]);
VelOut = (VelIn with [2] = newVelZ);
```

Figure 3.74: The colored content of a textual diagram in a Textual View

To design an operator with textual diagrams, the following tasks are possible:

- [“Inserting Textual Diagrams”](#)
- [“Applying Changes in Textual Diagrams”](#)
- [“Editing Textual Diagrams”](#)
- [“Navigating from Text Diagrams to Other Views”](#)

Inserting Textual Diagrams

You can define the design of operators with textual descriptions using textual diagrams in the **Scade View**.

Note

SCADE Suite operators (node or function) and states can be declared with as many text diagrams in combination with graphical diagrams.

To insert textual diagrams

- 1 Display **Scade View** and expand existing folders.
- 2 Right-click an operator name in **Scade View** and select *Insert > Text Diagram*, or click  in **Create** toolbar.
- 3 Double click the icon of the text diagram and write the textual description of the operator directly in the open Textual View.

Note

The textual description must comply with Scade language. For available syntax, consult Chapter 6 about [“User-Defined Operators”](#) and Chapter 7 about [“Operator Bodies”](#) in *Scade Language Reference Manual*.

- 4 Right-click in **Textual View** and select *Apply* contextual menu to apply all changes (see more in [“Applying Changes in Textual Diagrams”](#) on page 265).
-

Note

Graphical diagrams can be turned into text diagrams by selecting the Convert to textual contextual menu from diagrams in Scade View. When converting graphical diagrams into textual diagrams, all graphical information is lost and internal variables are renamed. Textual diagrams cannot be turned back into graphical ones.

Editing Textual Diagrams

You can perform various editing tasks when opening textual diagrams in **Textual Views**.

- **Completion:** type the first letters of keywords and it is completed. Read about how [“To complete text automatically”](#) on page 262.
- **Searching:** facilitates location of character strings. Read about how [“To find text strings in textual diagrams”](#) on page 262.
- **Replacing:** facilitates location and replacement of character strings. Read about how [“To replace text strings in textual diagrams”](#) on page 263.
- **Marking lines:** marks the lines where given character strings are found. Read about how [“To mark lines in textual diagrams”](#) on page 263.
- **Jumping to line:** facilitates navigation in the view directly to a given line. Read about how [“To jump to lines in textual diagrams”](#) on page 264.
- **Syntax checking:** checks that the modified operator is syntactically correct. Read about how [“To check syntax in textual diagrams”](#) on page 264.
- **Pragma display:** controls the display of pragma information. Read about how [“To show/hide pragmas from textual diagrams”](#) on page 264.
- **Undo/Redo:** while editing, return to the previous state of operator description.
- **Cut/Copy/Paste:** facilitates editing of character strings.

To complete text automatically

- 1 Type the beginning of a text string.
- 2 Right-click and select *Complete Word*, or alternatively press `Ctrl+Space` to launch word completion help:
 - If a single word starts by these letters, the word is completed.
 - If several words are possible, a list shows all possible definitions in the context. Choose in the list with arrow keys and click `Tab`-key to validate.
 - If you can access a package scope, select the package name and the list automatically pops up with a list of possible definitions in the package.

To find text strings in textual diagrams

- 1 Right-click in **Textual View** and select *Find* (`Ctrl+F`) to open the **Find** window.

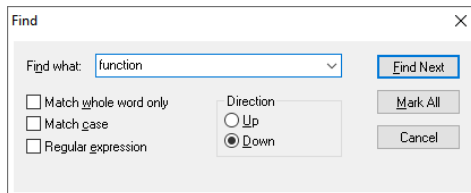


Figure 3.75: Find window from Textual Views

- 2 Enter the string to search in **Find what** or select a string from the drop-down list.
- 3 Specify search options and change **Direction** if needed.
- 4 Click **Find Next** to locate text by occurrences. Press `F3` to keep searching next occurrences forward or `Shift+F3` to keep searching next occurrences backward.

To replace text strings in textual diagrams

- 1 Right-click in **Textual View** and select *Replace* (Ct r L+H) to open the **Replace** window.

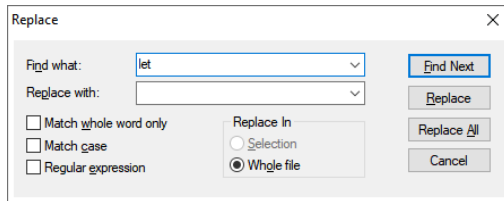


Figure 3.76: Replace window from Textual Views

- 2 Enter the string to search in **Find what** or select the string from the drop-down list.
- 3 Enter the replacing string in **Replace with** or select string from the drop-down list.
- 4 Specify search options and change **Direction** if needed.
- 5 Indicate from **Replace in** options whether to replace searched string in the **Whole file** or only in current text **Selection**.
- 6 Click buttons as needed:
 - **Replace** to replace the current occurrence and search for next one.
 - **Replace All** to replace all searched occurrences.
 - **Find Next** to search for next occurrence without replacing.

To mark lines in textual diagrams

- 1 Right-click in **Textual View** and select *Find* (Ct r L+F) to open the **Find** window.
- 2 Specify the text string to locate by lines in the **Find what** field.

- 3 Click **Mark All** to highlight any line containing the specified text string.
-

Tips

- Mark the line where the cursor is located using the **Ctrl+F2** keys. Unmark it using the **Ctrl+F2** keys. Suppress all marks with the **Ctrl+Shift+F2** keys.
 - To navigate between marked lines: Jump from one string occurrence to another with the **F2** key to move down or the **Shift+F2** keys to move up.
-

To jump to lines in textual diagrams

- 1 Right-click in **Textual View** where the cursor is located and select *Go to line* (**Ctrl+G**) to open the **Go to line** dialog box.
 - 2 Enter the line number and click **OK**. The cursor highlights the line.
-

Note

The line and column number indicating the cursor position are displayed on the right of the Status bar.

To check syntax in textual diagrams

- Right-click in **Textual View** and select *Syntax check*. If SCADE Suite detects errors in the textual description, a pop-up box displays the corresponding error messages. Textual diagrams can be saved and loaded even if syntactically or semantically incorrect.

To show/hide pragmas from textual diagrams

- Right-click in **Textual View** and select *Display Pragmas > Show* or *Display Pragmas > Hide*.
-

Note

In hide mode, the view is read-only with a grey background. To switch back to write-mode, select *Display Pragmas > Show*. For guidelines about pragmas in models, see [“Setting Pragmas on Model Elements”](#) on page 859.

Applying Changes in Textual Diagrams

When updating text diagrams, the content is not consistent with the model anymore. The white background of the view turns to a different color indicating that you must apply changes. Applying changes checks syntax in the textual description.

To apply changes done in textual diagrams

- Right-click in **Textual View** and select *Apply*.

Note

If there are no syntax errors, the text description is integrated in the model. A red underline indicates that changes are syntactically incorrect.

Navigating from Text Diagrams to Other Views

You can navigate directly from operator textual calls in text diagrams to the operator description in the model.

To navigate from a textual diagram

- 1 Open the **Textual View** of any operator with textual definitions by double clicking the appropriate Text Diagram in **Scade View**.
- 2 Double click the text of the operator call to highlight it, right-click the text and select *Navigate*. The **Diagram View** or **Textual View** containing the operator definition opens.

Defining Textual States or Actions

While designing State Machines or If and When blocks, you can integrate the textual representation of states or actions into graphical diagrams and edit textual definitions directly within **Diagram Views**. States and actions can be displayed and edited in graphical or textual definition mode.

- [“Displaying States/Actions in Graphical/Textual Modes”](#)
- [“Editing Textual States/Actions”](#)

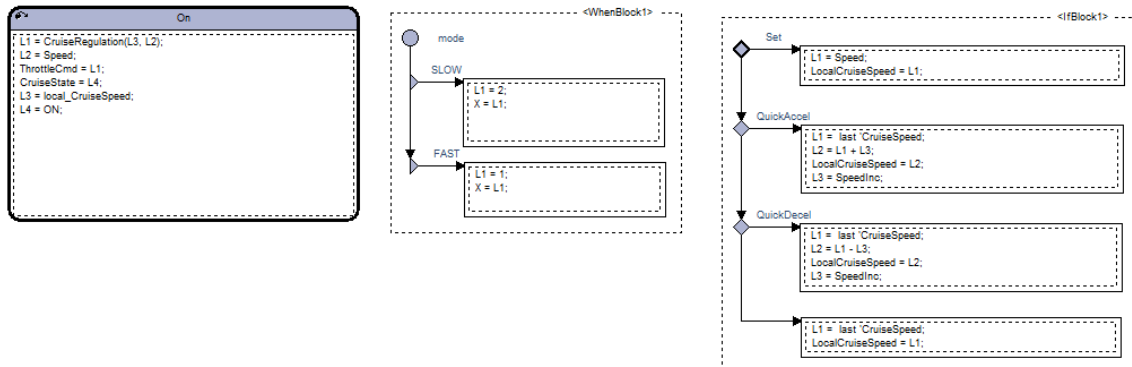


Figure 3.77: State and actions displaying in textual definition mode

Displaying States/Actions in Graphical/Textual Modes

You can change the definition mode for states and actions by switching their display in **Diagram Views** between graphical or textual mode. It is possible to translate graphical definitions into textual definitions.

Depending on their content and whether displayed in hidden/embedded mode (see [“Setting States/Actions in Embedded/Hidden Mode”](#) on page 199), states and actions can be used in graphical and/or textual mode as summarized below.

Table 3.1: States or actions in graphical/textual display mode by context

Context	Possible Changes
State/action with embedded content	• Converting to textual: inserts empty and editable text area or translates graphical content textually (internal variables as locals). • Converting to graphical: possible if no textual content.
State/action with hidden content	• Converting to textual: inserts empty and editable text area or translates graphical content textually (internal variables as locals). Automatically returns to display of parent operator.

To switch states/actions display between graphical and textual

- Right-click in the selected state (or action) and select *Convert to Textual* or *Convert to Graphical*.

Tip

Switch the display between Graphical and Textual from the Declaration properties of the selected state/action.

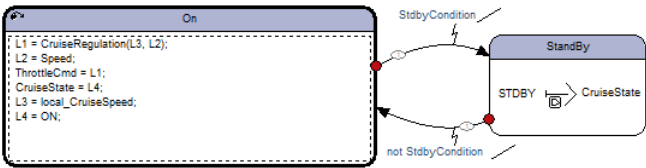


Figure 3.78: State design converted in textual representation

Editing Textual States/Actions

You can edit textual definitions directly within **Diagram Views** or through property pages of states/actions.

- Resizing of states/actions modifies the size of the textual definition area.
 - Text is wrapped automatically after typing (as in transition labels) according to space set for the embedded display of the textual states/actions.
 - Text styles can be edited from *Edit > Styles*. Text is displayed according to **Normal** style applied to the **Action** category. For details, see [“Creating and Modifying Styles”](#) on page 134.
 - Word completion help is available while typing text: press **Ctrl+Space**.
-

Tips

- If a single word starts by these letters, the word is completed.
 - If several words are possible, a list is displayed with all possible definitions in the scope context. Select the appropriate item with arrow keys and click the **Tab** key to validate the choice.
 - If you can access a package scope, select the package name and the list automatically pops up with a list of possible definitions in the package. Select the appropriate item as explained above and validate.
-

To edit textual states/actions

- 1 Click in text field of textual state/action to enable textual content editing. If the size of the state/action is too small for the content, a partial preview is enabled and highlights the entire content.

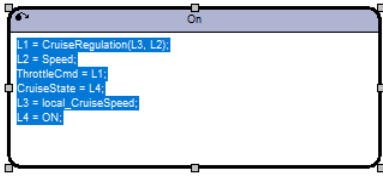


Figure 3.79: Partial preview for textual state/action editing

Tip

To ease content visualization or editing, display the Body properties of the selected state/action. For details, see [“To edit the textual definition body of a state”](#) on page 473, [“To edit the textual definition body of When Branch actions”](#) on page 469, and [“To edit the textual definition body of If Node actions”](#) on page 465.

2 Type the textual definition.

- Press Enter to insert new lines or Alt+Enter to apply modifications.

Note

- If the text is syntactically incorrect, it is underlined in red.
 - If the text contains any reference to unknown object, it is underlined in green.
-

To adjust symbol size to text field content

- Right-click the selected state (or action) and select *Optimize* to resize the symbol and display textual content without ellipsis (...).
-

Note

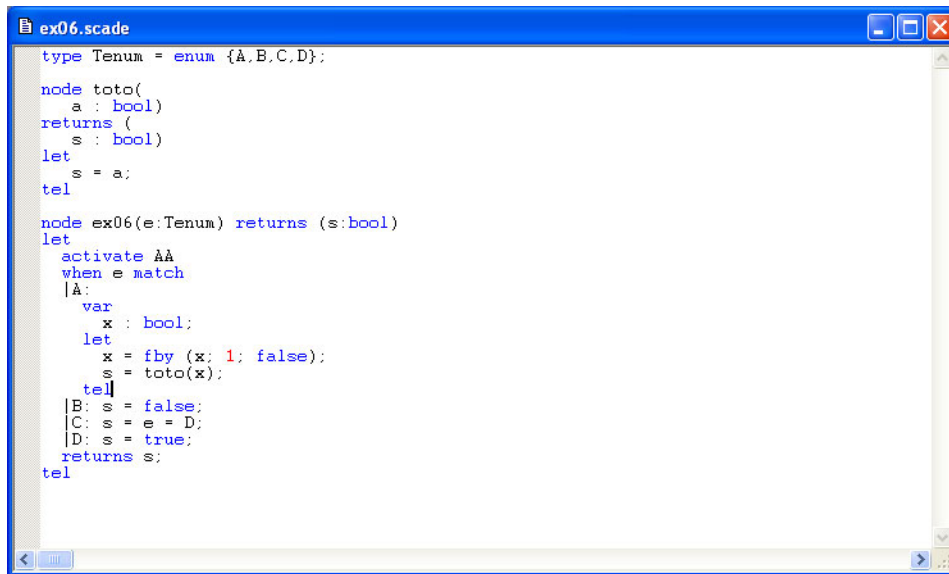
- If space around state/action is insufficient, resizing is impossible: make room around state or action symbol in Diagram View.
 - If there is no textual content, the symbol is resized to default size.
-

Creating Instances from Textual Files

You may provide textual definitions directly in SCADE Suite projects with SCADE files (.scade) inserted through **File View**.

Note

Such files contain textual definitions in Scade language to declare constants, types, sensors, packages and operators at model level. They can be created and edited in any text editor. They can also be edited from a text editor in SCADE Suite.



```
type Tenum = enum {A,B,C,D};

node toto(
  a : bool)
returns (
  s : bool)
let
  s = a;
tel

node ex06(e:Tenum) returns (s:bool)
let
  activate AA
  when e match
  |A:
    var
      x : bool;
    let
      x = fby (x; 1; false);
      s = toto(x);
    tel
  |B: s = false;
  |C: s = e = D;
  |D: s = true;
  returns s;
tel
```

Figure 3.80: SCADE file inserted in project and opened in text editor

You can use textual definitions in the graphical design as any other predefined or user-defined graphical constructs provided the textual definitions are referenced in the SCADE Suite project.

To reference textual files and create instances from textual definitions

- 1 Define the operator using textual Scade language in any text file format.

- 2 Save the definitions in a `.scade` file.
 - 3 Display **File View** and expand existing folders.
 - 4 Right-click on the **Model Files** folder and select *Insert Files*.
-

Note

You may create own folder to reference one's own definition files as explained in [“Creating New Folders in Projects”](#) on page 114.

- 5 Browse for the SCADÉ file where you saved the definitions and click **Open** to insert the selected file.
-

Warning

Opening SCADÉ files (.scade) in SCADÉ Textual Editor does not provide access to the editing features described in [“Editing Textual Diagrams”](#) on page 261.

- 6 Switch to the **Scade View** and expand the tree to display the definitions you inserted (at model level or within defined packages). All SCADÉ objects defined in the file are automatically loaded.
-

Note

In case you insert a file with syntactical errors, you obtain a warning message, but the definitions cannot be loaded in Scade View until you edit the syntax error(s).

- 7 Drag and drop the objects to complete the design in **Diagram Views**.
-

Tip

To edit definitions from SCADÉ files, double click the file in File View or any instance of the definition in Diagram Views or in Scade View.

Creating Textual Expressions

You can create as many textual expressions and connect them to other objects as you need in **Diagram Views**. Textual expressions are not defined in the **Scade View** and their value is set directly in **Diagram Views**.

To create textual expressions

- 1 Click **New Textual Expression**  in **Create** toolbar.
- 2 Click in any **Diagram View** where needed.
- 3 Enter the expression text directly from the **Diagram View**. The expression can be any expression written with respect to textual Scade language, even simple expressions like literals or constant references.

Edit values by clicking twice on the element, by selecting the element and pressing F2, or by opening the **Properties** window of the expression.

Expression can be a char array value specified as string or character by character as detailed in [“To define char array values”](#) on page 210.

While typing textual expressions, you can enable the editing help to select from the list of names available in the current project (see how [“To complete text automatically”](#) on page 262). You get syntactic assistance while typing:

- If the text you type is not syntactically valid, it is underlined in red.
- If the text you type contains any reference to unknown object, it is underlined in green.

Managing Instance Names Textually

You can associate instance names manually with any textual expression contained in models. SCADE Suite is able to manage instance names automatically for you in *textual files*, *text diagrams*, or *textual expressions* within graphical diagrams. If you do not associate any instance name with a textual expression, SCADE Suite Editor can compute them and associate them with expressions declared in textual or graphical views.

Note

Read about instance names in [“Editing Instance Names”](#) on page 487.

To associate instance names manually in textual expressions

- 1 Type any textual expression with respect to Scade language syntax.
- 2 Assign an instance name to any operator by inserting `#identifier` in front of the operator textual definition (`identifier` can be any number).

```
O = N ( 1 + 2 , S ( 3 + 4 ) );  
O = #1N ( 1 #1+ 2 , #1S ( 3 #2 + 4 ) );
```

The user-defined instance names associated with textual expression are displayed everywhere they can be edited in SCADE Suite (textual and graphical diagrams, or Properties). When automatically computed, instance names are not displayed. Numbers as instance names are reserved.

Checking Model Semantics

SCADE Suite Checker is a tool which checks the semantics of models. While designing a model, you may introduce semantic errors. You must correct them before you can ever use a model for simulation or code generation.

- [“Fundamentals about Checker”](#)
- [“Launching Semantic Checks”](#)

Fundamentals about Checker

The semantic checks from the Checker are completed along several check phases: the completion check, the instance name check, and the semantic check.

COMPLETION CHECK

This phase checks whether the model is complete or not: the right part of any equation must not be empty; an expression must not call empty parameters; and constants, sensors, and variables must be typed. The completion check returns errors in the “Completion” category.

Warning

When Completion errors are returned, no semantic check is performed. Correct the completion errors first and run again the tool to resume additional checks.

INSTANCE NAME CHECK

This phase checks in operators if an operator is called several times using the same instance name or several times without instance name. The instance name check returns warnings in the “Instance Name” category.

SEMANTIC CHECK

This phase checks the semantics in the model based on all checks performed by SCADE Suite Code Generator selected in the active configuration. The semantic check returns errors and warnings from Code Generators. For details about checks and messages, consult Chapter 12 about [“Code Generation Messages”](#) in *SCADE Suite Technical Manual*. To launch SCADE Suite Checker in batch, see [“Semantic Check Command Line”](#) on page 918.

Note

In the context of SCADE Code Generator for AUTOSAR (ACG), the semantic checks are performed using SCADE Suite KCG. For details, see Chapter 2 about [“Designing Models for AUTOSAR”](#) in *SCADE Automotive Package Guidelines*.

Launching Semantic Checks

You can launch SCADE Suite Checker by selecting one of the following items:

- Single user-defined operator to check only the selected operator
 - Single package to check all root operators in the package
 - Model folder to check all root operators in the model folder
 - Project (etp file) to check the root operator(s) set in the active configuration
-

Warning

When launching a check on projects, the Checker analyzes all declarations whether used or not in the model. A check on model folders, user-defined operators, or packages does not detect possible semantic issues about constants, types, or sensors if they are not used, directly or indirectly, by an element contained in the object on which you launch a check.

Result of check for model CruiseControl

5 error(s) detected - 1 warning(s) detected

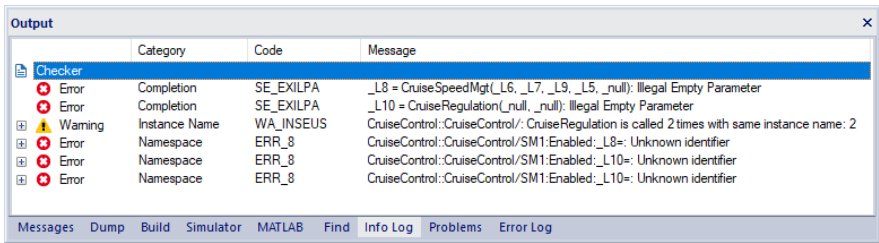
Category	Code	Message
Completion Error	SE_EXILPA	Illegal Empty Parameter at _L8 = CruiseSpeedMgt(_L6, _L7, _L9, _L5, _null)
Completion Error	SE_EXILPA	Illegal Empty Parameter at _L10 = CruiseRegulation(_null, _null)
Instance Name Warning	WA_INSEUS	CruiseRegulation is called 2 times with same instance name: 2 at CruiseControl::CruiseControl/ (_L10 = CruiseRegulation(_null, _null))
Namespace Error	ERR_8	Unknown identifier at CruiseControl::CruiseControl/SM1:Enabled: _L8= the flow identifier _null is undefined at this point
Namespace Error	ERR_8	Unknown identifier at CruiseControl::CruiseControl/SM1:Enabled: _L10= the flow identifier _null is undefined at this point
Namespace Error	ERR_8	Unknown identifier at CruiseControl::CruiseControl/SM1:Enabled: _L10= the flow identifier _null is undefined at this point

End of document.

Figure 3.82: Checker report

The located element is highlighted by a colored frame in the diagram. Press Escape to remove the frame display.

In addition to the report, you can access to a summary of the check results from the **Info Log** output tab as illustrated below. Remember that Completion errors must be corrected first, before detecting possible semantic issues within models.



The screenshot shows a software window titled "Output" with a close button (X). It contains a table with columns "Category", "Code", and "Message". To the left of the table is a "Checker" tree view with icons for Error (red X), Warning (yellow triangle), and Information (blue i). The table lists the following results:

	Category	Code	Message
Checker			
Error	Completion	SE_EXILPA	_L8 = CruiseSpeedMgt(_L6, _L7, _L9, _L5, _null) : Illegal Empty Parameter
Error	Completion	SE_EXILPA	_L10 = CruiseRegulation(_null, _null) : Illegal Empty Parameter
Warning	Instance Name	WA_INSEUS	CruiseControl::CruiseControl: CruiseRegulation is called 2 times with same instance name: 2
Error	Namespace	ERR_8	CruiseControl::CruiseControl/SM1:Enabled:_L8=: Unknown identifier
Error	Namespace	ERR_8	CruiseControl::CruiseControl/SM1:Enabled:_L10=: Unknown identifier
Error	Namespace	ERR_8	CruiseControl::CruiseControl/SM1:Enabled:_L10=: Unknown identifier

At the bottom of the window, there is a tab bar with the following tabs: Messages, Dump, Build, Simulator, MATLAB, Find, Info Log (selected), Problems, and Error Log.

Figure 3.83: Checker results on Info Log tab

Tip

To navigate to the source of detected issues in the design, right-click any line in the Info Log tab and select *Locate*.

CHECKER ERROR AND WARNING MESSAGES

SCADE Suite Checker can raise the following errors (SE) or warnings (WA).

ID	Message
Completion check errors	
SE_EXILER	Source Code Error, refers to expression text
SE_EXILPA	Illegal Empty Parameter, refers to expression call and expression id
SE_TRILER	Source Code Error, refers to transition
SE_TYILDE	Illegal Empty Type in Definition, refers to expression type and typed object
Instance check errors and warnings	
WA_INSEUS	Instance Name Warning - Operator is called N times with same instance name, where N is number of times

Modifying Design Elements

The models you design in SCAD Suite can be modified in various ways depending on the modification purpose. Several features are available in:

- [“Moving Design Elements in Models”](#)
- [“Deleting Declarations from Scade View”](#)
- [“Modifying Definitions from File View”](#)

Moving Design Elements in Models

You can move or cut/paste design elements in the design at any time.

To move design elements in the graphical design

- Select any element in **Diagram Views** and drag the selected element to a new position. The shape of the cursor changes to a multi-directional cross.

To move design elements in Scade View

- Select any design element in **Scade View** and drag the selected element to a new position in **Scade View**. During drag and drop, hold CTRL-key to copy the selected element(s) instead of moving them.

Note

- Use drag and drop to change the order of operator inputs/outputs/parameters.
 - To perform model refactoring, it is possible to use drag and drop between two models in the same workspace. Such operation may create pending references that can be resolved as detailed in Chapter 9 about [“Managing Model Refactoring”](#).
-

To cut/paste elements in or between the graphical design

- 1 Select the element(s) to paste.
- 2 Right-click and select *Cut* if you want to paste the selected element(s).

- 3 Open the **Diagram View** where to paste the element(s) or select a location in the same **Diagram View**.
- 4 Right-click and select *Paste* to copy the selected element(s).

To cut/paste design elements in Scade View

- 1 Select design elements defined in **Scade View** and press C t r L+X.
- 2 Expand tree items and select the location where to paste the selected elements(s) and press C t r L+V.

CUTTING/COPYING/PASTING GENERAL RULES

While cutting, copying, and pasting elements, consider the following cases:

- All notes, comments, and assignments associated with the selected element are pasted.
- Cutting always removes selected elements and its references.
- Cutting a reference to any SCADE object deletes all its connecting wires whether selected or not.
- Cutting a state in State Machines deletes all its transitions whether selected or not.
- Copying a single connecting wire or transition can only be done by copying at the same time the objects connected on each side.
- Cutting and pasting an input between Diagram Views creates the same input declaration in the operator interface corresponding to the destination view.

Note

Cut/copy and paste is also possible between different instances of SCADE or in Word documents.

Modifying Definitions from File View

Modifying definitions in models can be done selectively without losing graphical instances in Diagram Views. From **File View**, you remove only the declaration of a definition (*i.e.*, XSCADE or library file) and keep all instances in **Diagram Views**. Instead, if you want to delete declarations and instances from models, see [“Deleting Declarations from Scade View”](#) on page 282.

To modify definitions and keep graphical instances

- 1 Display the **File View** of the SCADE Suite project.
- 2 Select the XSCADE file (or the ETP library file) of the operator(s) to remove without suppressing its references in **Diagram Views**.
- 3 Click **Delete**  in **Standard** toolbar or press **De** **l** **e** **t** **e**. The file is removed from the project (the file is not deleted from project directories), but the reference(s) remain displayed in **Diagram Views** with all connections.

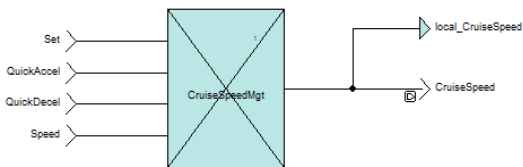


Figure 3.84: Diagram View with operator instance whose definition is missing

- 4 Replace the definition(s) with a new definition(s) of the same name(s) or insert a library that contains the new definition(s) under the same name(s).

Warning

To update undefined library operators, you replace the missing library with a library that contains the library operators with the same name. Remember to check for model inconsistencies in case the definition of library operators has changed in the meantime. SCADE checks automatically for inconsistencies at project launch. You can also resolve inconsistencies from the *Project > Resolve Inconsistencies* menu. For user instructions, see [“Applying User Fixes to Inconsistencies”](#) on page 545.

Deleting Declarations from Scade View

You can delete the declaration of an operator, type, constant, sensor, or variable and remove all corresponding instances at once in **Diagram Views**. Instead, if you want to change definitions without removing instances, see [“Modifying Definitions from File View”](#) on page 281.

To delete declarations from models

- 1 Display the **Scade View** of the SCADE Suite project.
- 2 Select the design element to delete entirely from the model.
- 3 Click **Delete**  in **Standard** toolbar or press **De** **l** **e** **t** **e**. All instances used in the **Diagram Views** of the model are deleted with all their connections. Whenever you delete the last declaration contained in a folder, the corresponding folder is deleted from **Scade View**.

Tip

To remove declarations and all graphical instances directly from Diagram Views, right-click the element and select *Delete from Model*.

Optimizing Design Layout

While designing data and control flow in the **Diagram Views** or editing symbol components in **Symbol Editor** views, you can optimize the layout of graphical elements and improve the reading of the graphical representation. You have the possibility to optimize connection wires and align, distribute, resize, rotate, nudge, or order the elements.

- [“Selecting Design Elements”](#)
- [“Aligning Elements”](#)
- [“Aligning Design Components”](#)
- [“Distributing Elements and Connection Wires”](#)
- [“Resizing Elements”](#)
- [“Resizing Elements Manually”](#)
- [“Rotating Elements”](#)
- [“Nudging Elements”](#)
- [“Optimizing Connection Wires”](#)
- [“Editing Connection Wires and State Transitions”](#)
- [“Straightening Connection Wires”](#)
- [“Snapping Design Elements”](#)

Selecting Design Elements

The selection of design elements in active views can be handled in several ways. Proceed accordingly as needed:

- **Selection of view content:** Select *Edit > Select All* or press **Ct r L+A**.
- **Area selection:** Click in view and drag the cursor to delimit an area. Only elements contained entirely in the specified area are selected.
- **Selection by adding:** Hold **Ct r L**-key and pick multiple elements.
- **Selection by intersection:** Hold **Ct r L**-key, click in graphical view and drag to delimit an area and select any element overlapping the specified area.
- **Selection of small symbols:** Press **V**-key with mouse over the symbol, click and move the element. Labels, handlers, and ports are ignored.

Selecting elements is possible only in Selection mode: click **Select Mode**  in **Zoom/Pan** toolbar to return back to Selection mode.

Aligning Elements

You can select graphical elements and align them horizontally or vertically.

To align design elements

- 1 Select the elements to align while holding the Shift-key down. If you select wires, see [“Straightening Connection Wires”](#) on page 293. Alternatively, right-click selected elements to display alignment possibilities.
- 2 Select *Layout > Align* and choose between:
 - **Align Top** : aligns horizontally relative to top border of last selection.
 - **Align Middle** : aligns horizontally relative to middle of last selection.
 - **Align Bottom** : aligns horizontally relative to bottom border of last selection.
 - **Align Left** : aligns vertically relative to left border of last selection.
 - **Align Center** : aligns vertically relative to center of last selection.
 - **Align Right** : aligns vertically relative to right border of last selection.

Aligning several design elements which have different parents may lead to change the parent of any moved symbol.

All of the above commands are also available from the [Layout toolbar](#).

Aligning Design Components

You can select any set of design components and align it with respect to an anchor element. It is possible to align the bounding box defining a set of selected components, or a referent element in the set of selected components. The relative position of all components in the selected set is kept after alignment selection. All commands dedicated to component sets are available from the [Layout toolbar](#).

To align a set of components using bounding box

- 1 Select *Layout > Align > Align Like a Group*  to switch to component alignment mode.
- 2 Use multiselection to define the set of components to align in the diagram.
- 3 Select *Layout > Align > Define Anchor*  and click on the element to define as anchor (blue anchor symbol displayed).

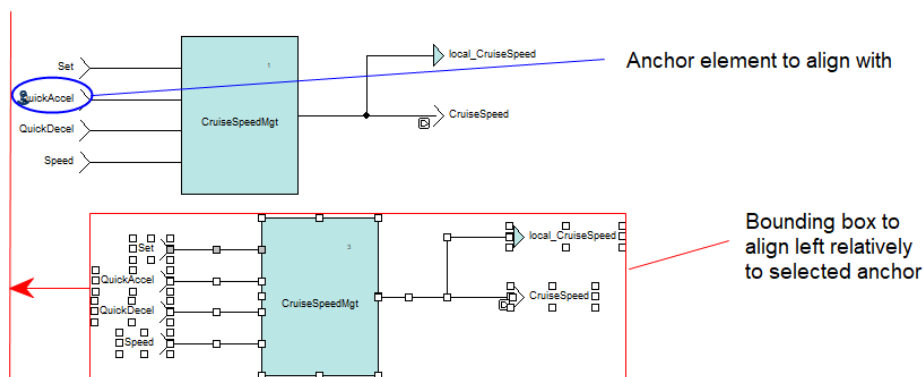


Figure 3.85: Selecting set of components and anchor before left aligning bounding box

- 4 Select *Layout > Align > Consider Bounding Box*  to enable bounding box alignment.
- 5 Align the bounding box relatively to the anchor element borders as needed using the alignment commands described in [“Aligning Elements”](#) on page 284. Right-click the selected components to choose alignment options.

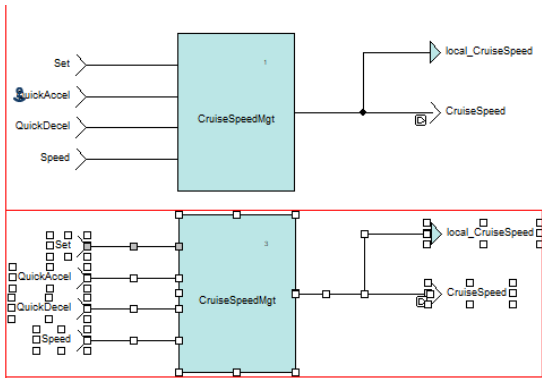


Figure 3.86: Result of aligning set of components using bounding box

To align a set of components using referent element

- 1 Select *Layout > Align > Align Like a Group*  to switch to component alignment mode.
- 2 Use multiselection to define the set of components to be aligned in the diagram.
- 3 Define the referent element (hold **Ctr L**+selection click) from the set of components.

The referent element is highlighted by grey selection handles and can be changed before applying alignment. Connection wires cannot be used as referent element.

- 4 Select *Layout > Align > Define Anchor*  and click on the element to define as anchor (blue anchor symbol displayed).

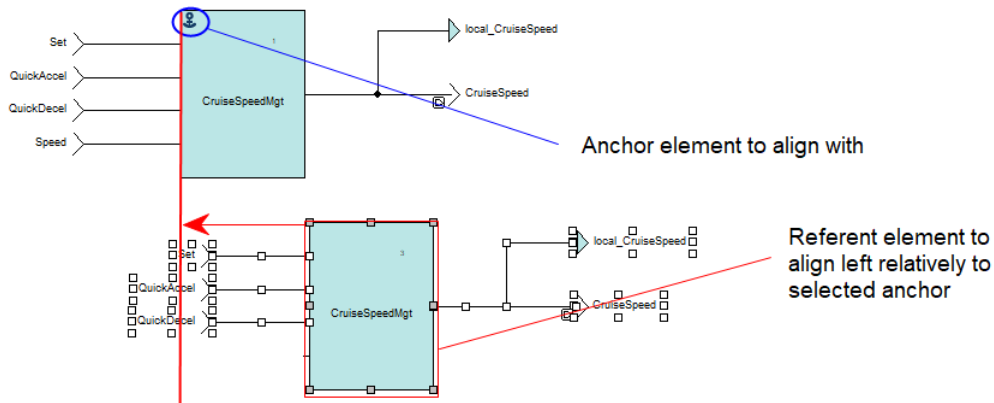


Figure 3.87: Selecting set of components, referent element, and anchor before aligning

- 5 Select *Layout > Align > Consider Bounding Box*  to disable bounding box alignment.
- 6 Align the referent element relatively to anchor element borders as needed using the alignment commands described in [“Aligning Elements”](#) on page 284. Alternatively, right-click the selected set of components to choose from alignment options.

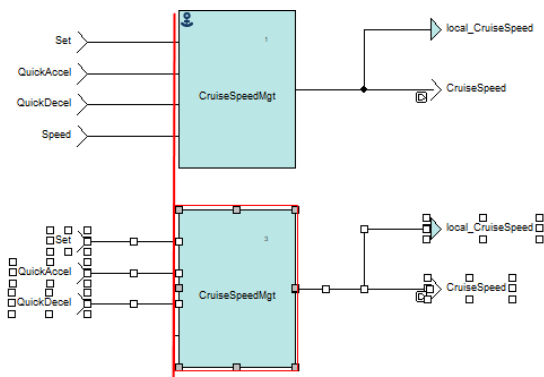


Figure 3.88: Result of left-aligning set of components using referent element

Distributing Elements and Connection Wires

You can select graphical elements and distribute them horizontally or vertically. A note with a relative position cannot be aligned. If a note cannot be moved, you cannot align it. This option keeps the relative position of elements.

To distribute design elements and connection wires

- 1 Select the elements to distribute while holding the **Shift**-key down or select multiple connecting wires (select at least three elements or two wires).
- 2 Select *Layout > Distribute* and choose between:
 - **Space Across** to dispatch elements horizontally relative to the top of the last selected element. Use this option to arrange horizontal wires.
 - **Space Down** to dispatch elements vertically relative to the middle of the last selected element. Use this option to arrange vertical wires.

Distributing several State Machines and other design elements which have different parents may lead to change the parent of any moved symbol.

All of the above commands are also available from the [Layout toolbar](#). In case of multiselection of wires, you also adjust the distance proportionally between wires according to the relative position of the first and last wire in the multiselection.

Resizing Elements

You can optimize the size of several graphical elements and scale them relatively to each other (see also [“Resizing Elements Manually”](#) on page 290). When resizing elements, truncated text is underlined by the ellipsis (...) mark (see [“Underlining Truncated Text in Views”](#) on page 149).

To resize design elements

- 1 Select the elements to resize while holding the Shift-key down.
- 2 Select *Layout > Size* and choose between the resizing options:
 - **Same Height**  to size all elements to the height of last selected element.
 - **Same Width**  to size all elements to the width of last selected element.
 - **Same Size**  to size all elements to the size of last selected element.
 - **Default size** to resize an element as defined from **Symbol Editor**. You must select a single element. See [“Editing Design Elements with Symbol Editor”](#) on page 294.

Applying the same size on several State Machines and other design elements which have different parents may lead to change the parent of any moved symbol. Default size on State Machines and conditional blocks resets its creation size when possible regarding its design content.

All of the above commands are also available from the [Layout toolbar](#). You can also optimize the size of operator calls, state machines, and states. The size can be adjusted with respect to content or to the number of inputs/outputs. Optimizing the size of state machines is done according to any state and transition content. If state display is in hidden mode, states are optimized according to their default creation size.

To optimize design elements

- 1 Select the element whose layout needs optimizing: select an operator, a textual expression, or a state (for connection wires, see [“Optimizing Connection Wires”](#) on page 291).
- 2 Select *Layout > Size > Optimize*. Alternatively, right-click on selected element and select **Optimize**.

Resizing Elements Manually

You can adjust the size of design elements manually in **Diagram Views**. It is possible to scale them horizontally, vertically, or in both directions. To resize elements relatively to each other, see [“Resizing Elements”](#) on page 288. While resizing elements, truncated text is underlined by the ellipsis (...) mark. For more details, see [“Underlining Truncated Text in Views”](#) on page 149.

To resize data flow and control flow elements manually

- 1 Select an element in a **Diagram View** to display its scale handlers.
- 2 Grasp the appropriate one (cursor as bi-directional arrow) and move the cursor horizontally, vertically, or both horizontally and vertically. For proportional scaling:
 - Hold **Ct r l**-key down and grasp either corner handler to resize in all directions or side handler to resize horizontally or vertically.
 - Hold **Sh i f t**-key down and grasp corner handler to resize with equal horizontal and vertical scale factors.

Rotating Elements

You can select graphical elements and rotate clockwise or counter-clockwise. It is not possible to rotate state machines or the content of textual areas.

To rotate design elements

- 1 Select the element(s) to rotate while holding **Sh i f t**-key down.
- 2 Select **Layout > Rotate** and choose between:
 - **Rotate Right**  to rotate elements 90 degrees clockwise.
 - **Rotate Left**  to rotate elements 90 degrees counter-clockwise.

All of the above commands are also available from the [Rotate toolbar](#).

Nudging Elements

You can select graphical elements and move them slightly. This is the same as using keyboard arrows to move elements. A note with a relative position cannot be nudged.

To nudge design elements

- 1 Select the element(s) to nudge while holding **Shift**-key down.
- 2 Select *Layout > Nudge* and choose between:
 - **Nudge Left**  to move all selected objects left.
 - **Nudge Right**  to move all selected objects right.
 - **Nudge Up**  to move all selected objects up.
 - **Nudge Down**  to move all selected objects down.

All of the above commands are also available from the [Nudge toolbar](#). If using the **Nudge** toolbar, hold the **Shift**-key down while clicking nudge buttons to move the selected element(s) by more units.

Optimizing Connection Wires

You can optimize the display of connection wires between elements in **Diagram Views**.

To optimize display of connection wires

- 1 Select the wire(s) whose layout to optimize. Alternatively, hold **Ctrl**-key down and move the mouse to define a selection area and select several connecting wires.
- 2 Select *Layout > Connection > Optimize* or press **Ctrl+Space** keys. Alternatively, right-click on a selected wire and select *Optimize*.

Editing Connection Wires and State Transitions

You can edit segments/tangents used to display connection wires between data flow elements or state transitions in state machines by adding/removing points. You cannot remove the default points necessary to draw them. It is possible to change the color of wires and transitions as described in [“Creating and Modifying Styles”](#) on page 134.

To add points on connection wires and state transitions

- 1 Select position on wire or transition by clicking anywhere on line except segment/tangent points.
- 2 Select *Layout > Connection > Add/Remove Point* to add point on position. Alternatively, right-click on wire or transition line where to add a point and select *Add/Remove Point*; or hold **Ct r l**-key down and click on empty place to add point.

To remove points added on connection wires and state transitions

- 1 Select a segment/tangent point on wire or transition.
- 2 Select *Layout > Connection > Add/Remove Point* to remove selected point. Alternatively, right-click on wire or transition point to remove and select *Add/Remove Point*; or hold **Ct r l**-key down and click the point to be deleted.

Straightening Connection Wires

You can select several wires to straighten them on **Diagram Views** provided there is enough space between connected operators.

To straighten a multiselection of wires

- 1 Hold **Ctr l**-key down and move the cursor to define a selection area that intersects with the wires to straighten.
- 2 Right-click on any of the selected wires:
 - Select *Align Left* or *Align Right* to straighten horizontal wires.
 - Select *Align Top* or *Align Bottom* to straighten vertical wires.

All of the above commands are also available from the [Layout toolbar](#).

Snapping Design Elements

While designing models, you may need to adjust and optimize the position of elements by snapping to the design grid.

To snap design elements to grid

- 1 Select **View > Snap to Grid** if you want the top left corner of the element graphical symbol to be aligned with the grid.

To display the grid, right-click in **Diagram Views** and select *Show Grid*. To modify the grid display, see [“Setting Grid Properties to Assist Design Positioning”](#) on page 152.

- 2 Select the element(s) and move where needed to adjust position to the grid.

Editing Design Elements with Symbol Editor

Some design elements you use to build models can be redesigned graphically. In Symbol Editor, you are able to modify the display of design elements or draw own symbols using various graphical components.

- [“Editing Layout of Graphical Symbols”](#)
- [“Inserting Images in Symbols”](#)
- [“Shaping Graphical Symbols”](#)
- [“Positioning Name on Variable Symbols”](#)
- [“Drawing Graphical Symbols”](#)
- [“Shifting Symbol Foreground or Background”](#)
- [“Editing and Inserting Text in Symbols”](#)

For refining the display and size of SCADE Suite elements in models, see also [“Optimizing Design Layout”](#) on page 283. The styles applicable to design elements can be used in Symbol Editor: for user instructions about predefined and user-defined styles, see [“Creating and Modifying Styles”](#) on page 134. Editing graphical symbols with Symbol Editor is handled by SSL symbol files: for details, consult Chapter 3 about [“Specific SCADE Suite Files”](#) in *SCADE Suite Technical Manual*.

EDITABLE SYMBOLS AND CONDITIONS

The symbols of user-defined operators can be edited graphically. The symbols of predefined operators cannot be edited. The symbols of inputs, hidden inputs, outputs, probes, or local variables can be edited under the following conditions:

- You can edit the symbols of inputs, outputs, hidden inputs, probes and local variables with Symbol Editor provided these element are configured with user-defined types (see [“Defining Data Types”](#) on page 207).

- If inputs, outputs, hidden inputs, probes and local variables are configured with predefined types, you have to enable the **Create in specific symbol directory** option from the *Tools > Options* and set a directory for graphical symbols (see [“Selecting General Options”](#) on page 77).

Warning

When editing the symbol of an input/output, hidden input, probe, or local variable, the symbols of all design elements set with the same type and the same category of symbol are updated when you apply changes.

In case some design elements are contained in a project library, their respective symbols cannot be edited graphically unless you open the library as a project.

Editing Layout of Graphical Symbols

You can edit the layout of the graphical components used for representing the symbols of design elements provided they are editable symbols.

To edit the layout of graphical symbols

- 1 Right-click an element in a diagram and select *Edit Symbol* to open the **Symbol Editor** window. Alternatively, right-click an element in **Scade View** and select *Edit Symbol*.

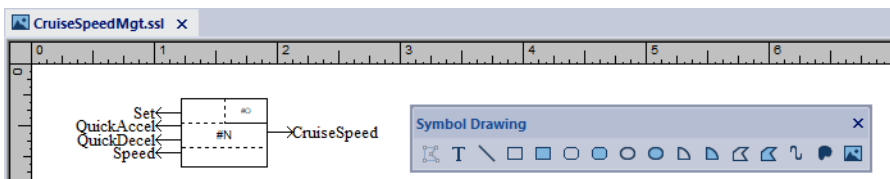


Figure 3.89: Display of Symbol Editor

- 2 Edit the layout of the selected element as follows:

- **To move graphical components**, click a component and drag where needed. If **Snap to Grid** is enabled (read how [“To move design elements in the graphical design”](#) on page 279), the top left corner of the component bounding box aligns with the grid.
- **To adjust the position of graphical components**, select a component and use the **Nudge** toolbar or the keyboard arrow keys. See [“Nudging Elements”](#) on page 291.
- **To rotate graphical components**, select a component and use the **Rotate** toolbar. See [“Rotating Elements”](#) on page 290.
- **To align graphical components**, select components and use the **Layout** toolbar. See [“Aligning Elements”](#) on page 284.
- **To distribute graphical components**, select components and use the **Layout** toolbar. See [“Distributing Elements and Connection Wires”](#) on page 288.
- **To shift graphical components in foreground/background**, right-click a component and select *Order*. See [“Shifting Symbol Foreground or Background”](#) on page 301.
- **To resize graphical components**, select component(s) and use the **Layout** toolbar. See [“Resizing Elements”](#) on page 288.

When changing any graphical layout in **Symbol Editor**, the colour of the background changes indicating layout inconsistency with the model. You need to apply changes.

- 3 Right-click in **Symbol Editor** window and select *Apply* to validate changes.

Shaping Graphical Symbols

You can modify the shape of editable symbols by editing vertices. The vertices define the shape of graphical components. Upon selection of any components, **Symbol Editor** displays control handles on every vertex. It is possible to manipulate vertices to change the shape of any component defined graphically by several handles.

Inserting or removing vertices applies only to components like multi-segment lines, polygons, polycurves or closed curves.

To shape graphical symbols with vertices

- 1 Select a graphical component in **Symbol Editor**.
- 2 Click **Edit Vertices**  in **Symbol Drawing** toolbar. Proceed as follows:
 - **To move a vertex**, position the cursor over a control handle. The cursor changes showing the selected vertex can be moved. Click the control handle and drag it to a new position. Release the mouse button to leave the vertex on its new position.
 - **To insert a vertex**, move the cursor between two control handles and hold down the C t r l-key. The cursor changes showing a vertex can be inserted. Click left to place the vertex on the line segment.
 - **To remove a vertex**, move the cursor over a vertex and press the C t r l-key. The cursor changes for a cross showing the vertex can be deleted. Click left to delete the vertex from the component.
- 3 Right-click in **Symbol Editor** window and select *Apply* to validate changes.

Drawing Graphical Symbols

You can create new symbols to represent design elements using a variety of basic shapes available from the **Symbol Drawing** toolbar.

To draw new graphical symbols

- 1 Create an operator and define its interface in **Scade View**.
- 2 Right-click the operator in **Scade View** and select *Edit Symbol* to open **Symbol Editor**.
- 3 Select *View > Toolbars > Symbol Drawing* to display the **Symbol Drawing** toolbar and insert the components you need for drawing the symbol:
 - To **draw lines**, click **Line**  in **Symbol Drawing** toolbar, click in **Symbol Editor** window to position a line start and drag to extend the line.

Tip

To draw horizontal/vertical lines, show the grid contextually and enable Snap to Grid. The line component automatically aligns between grid intersections.

- To **draw rectangles, rounded rectangles, ellipses, or arcs**, click the corresponding button in **Symbol Drawing** toolbar, click in **Symbol Editor** window to position the shape and drag to extend its dimensions.
 - To **draw polygons, polycurves or closed curves**, click the corresponding button in **Symbol Drawing** toolbar, click in **Symbol Editor** window to place vertices and double click to finish drawing. See also [“Shaping Graphical Symbols”](#) on page 297. Interrupt the drawing of polygons, polycurves or closed curves with a right-click.
 - To edit the shapes and components used in the drawing, see [“Editing Layout of Graphical Symbols”](#) on page 295.
- 4 Right-click in **Symbol Editor** window and select *Apply* to validate the new symbol.

Editing and Inserting Text in Symbols

You can edit existing text boxes or insert new ones while drawing symbols.

To edit text and font in text boxes

- 1 In **Symbol Editor** window, double click the text box to edit the text.
- 2 Right-click the text box and select *Font*.

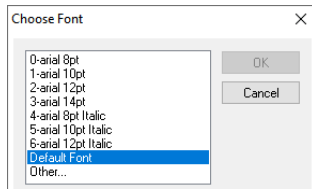


Figure 3.90: Choosing font applied to symbol text box

- 3 In **Choose Font** dialog, select a font or *Other* to customize it, and click **OK**. Fonts are saved in .ssl files. Fonts on text of variables cannot be changed.

To insert text box on symbols

- 1 Right-click an operator or variable in **Scade View** and select *Edit Symbol*.
- 2 Click **Text Box T** in **Symbol Drawing** toolbar, click and hold in **Symbol Editor** window to define a text area moving the mouse.
 - #N replaced by the name of the operator or variable, #NL enables to align text to the left, and #NR enables to align text to the right
 - #O replaced by the instance name of the operator
 - #S replaced by the signature of the operator
 - #NP replaced by the parameters list for a parameterized operator
 - #H_<number> replaced by the name of hidden input with rank number
 - #R replaced by the image of the content of the operator
- 3 Right-click and select *Apply* to validate changes.

Inserting Images in Symbols

While editing or drawing symbols, you can insert image files on one's own symbols. Images can be inserted in JPEG, EMF, BMP or ICO formats.

To insert image files on symbols

- 1 Right-click the symbol and select *Edit Symbol*.
- 2 Click **Image**  in **Symbol Drawing** toolbar to browse for files.
- 3 Select the file and press **Open**. The cursor turns into image insertion mode: click anywhere in the drawing area to insert the image.
- 4 Grab, resize, and move the image as needed on the symbol.

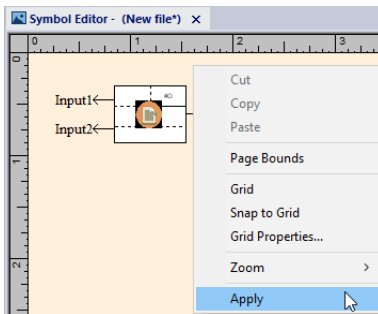


Figure 3.91: Applying image insertion

- 5 Right-click and select *Apply* to validate changes.

Positioning Name on Variable Symbols

While editing or drawing the symbol of a variable in Symbol Editor, you can modify the position of the text corresponding to the name of the element.

Warning

Changing variable name position on symbols has an impact on all variables of the same type. This may alter inadvertently the graphical display of the model.

To position name of variable symbols

- 1 Right-click an editable variable in **Diagram View** and select *Edit Symbol* to open **Symbol Editor**.
- 2 Select *View > Toolbars > Symbol Drawing* to display the toolbar.
- 3 Click **Text Box T** in **Symbol Drawing** toolbar to add a text box, and double click inside to type #N (replaced by name of operator), #NL (name of operator aligned left), or #NR (name of operator aligned right).
- 4 Select the text box you added and move it where needed next to the symbol.
- 5 Right-click and select *Apply* to validate changes.

With variables of predefined type, you must enable the creation of graphics in specific symbol directory (see [“Selecting General Options”](#) on page 77).

Shifting Symbol Foreground or Background

You can order elements displayed in Symbol Editor only by shifting their position between the foreground and the background.

To shift design elements in foreground or background

- 1 Select the element(s) to order while holding the Shift-key down.
- 2 Select *Layout > Order* and choose between the options:
 - **Bring to Front** to move the selected element(s) to the foreground.
 - **Send to Back** to move the selected element(s) to the background.
 - **Send Backward** to shift the selected element(s) one level backward.
 - **Bring Forward** to shift the selected element(s) one level forward.
- 3 Right-click and select *Apply* to validate changes.

Developing Control and Display Software

SCADE Suite allows developing software with graphical components. To do so, it is necessary to connect SCADE Suite and SCADE Display designs for co-simulation, standalone execution, code generation, and co-reporting.

- [“Joint Development Workflow”](#)
- [“Connecting Operator and Graphical Panel Interfaces”](#)
- [“Inserting Graphical Panels in Project”](#)

The tight integration between SCADE Suite and SCADE Display facilitates advanced co-development flows for safety-critical embedded display applications. This integration enables to interface SCADE Suite and SCADE Display for co-design and co-simulation, as well as for co-generation, co-reporting, and standalone executable generation.

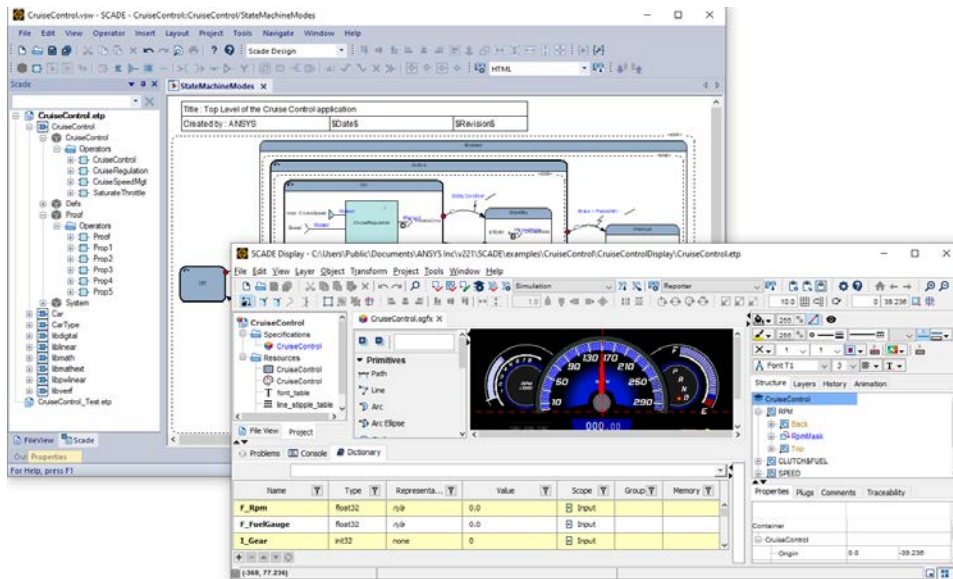


Figure 3.92: Logic (SCADE Suite) and graphic (SCADE Display) components of an application

There is no unique way of using SCADE Suite and SCADE Display together:

- start designing the SCADE Display graphical specification first, and then develop the logics part in SCADE Suite

- start with the SCADE Suite model first, and design the graphics part later
- develop both logics and graphics parts of the application jointly

A dedicated **Inputs/Outputs Connection** View allows managing all connections between SCADE Suite and SCADE Display models at operator level in SCADE Suite.

The view is separated in two resizeable parts:

- On the left, two views list all interfaces (inputs, outputs, sensors, and probes) of the selected operator and all interfaces (inputs and outputs) of the graphical panels inserted in the project. Connected items are in bold.
- On the right, the **Connections** view lists all existing connections between SCADE Suite operator and graphical interfaces, with status and direction.

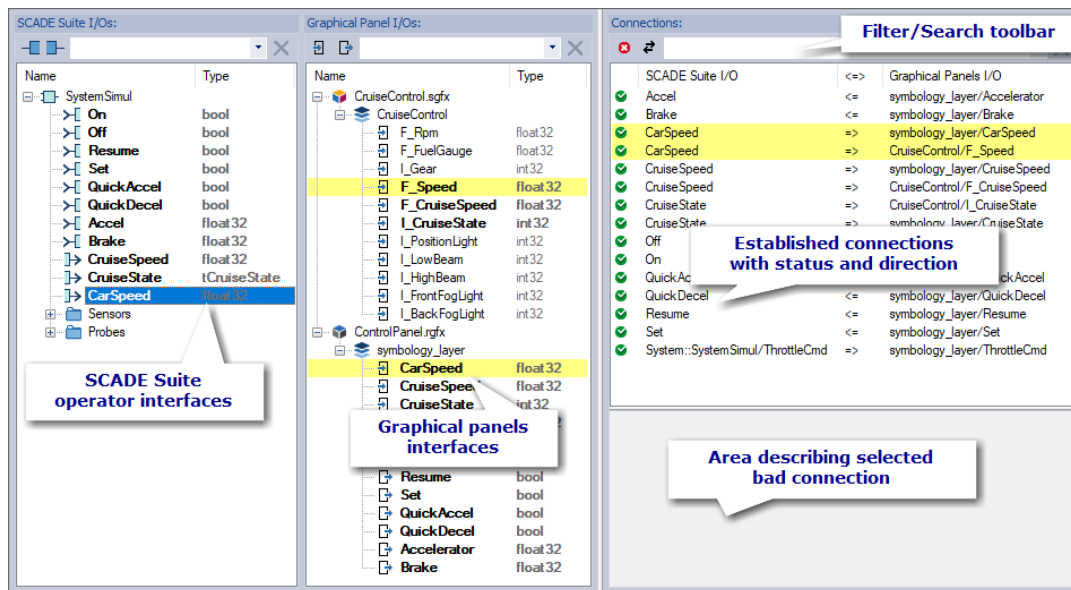


Figure 3.93: Inputs/Outputs Connection View

Joint Development Workflow

The workflow for developing control and display software using SCADE Suite and SCADE Display is described in the following figure:

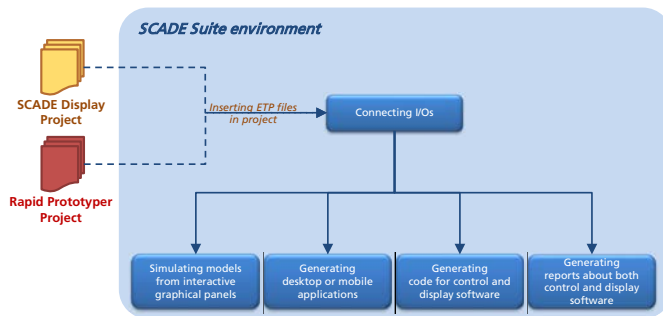


Figure 3.94: General workflow for developing control and display software

This workflow consists of the following activities:

- 1 Inserting projects with graphical panels in SCADE Suite project: [“Inserting Graphical Panels in Project”](#) on page 305.
- 2 Connecting SCADE Suite I/Os to graphical panels I/Os: [“Connecting Operator and Graphical Panel Interfaces”](#) on page 306. For connecting UA DF models, see [“Connecting SCADE Suite UA Model to Messages”](#) on page 105 in *SCADE UA Page Creator User Manual*.
- 3 Co-simulating models interactively from graphical panels: [“Simulating Models with Graphical Panels”](#) on page 625.
- 4 Generating desktop and mobile applications for demos, test, or validation purposes: Chapter 15 about [“Generating Desktop and Mobile Applications”](#).
- 5 Generating code embedding logics and graphics for target integration: [“Generating and Integrating SCADE Display Code”](#) on page 885.
- 6 Reporting about joint design models: [“Reporting Suite and Display/ARINC 661 Joint Design”](#) on page 45 in *SCADE LifeCycle User Manual*.

Inserting Graphical Panels in Project

You must insert a project file (.etp) containing graphical panels in the active project to be able to set I/Os connections and launch simulation sessions with these graphical panels.

To insert graphical panel project file in current SCADE Suite project

- 1 Select **Insert > Files** and browse for projects with graphical specifications:
 - SCADE Display, SCADE Server Creator, or SCADE UA Page Creator projects (.etp files containing .sgfx files)
 - Rapid Prototyper projects (.etp files containing .rgfx files)
- 2 Click **Open** to insert file. The inserted file is displayed in the **File View**.

For more information on file insertion in the current project, see [“Inserting Existing Project or Files”](#) on page 112.

You can also create Rapid Prototyper project files directly from the current SCADE Suite project. The project is automatically inserted in the project.

To create a Rapid Prototyper graphical panel from active project

- 1 Select **Insert > Graphical Panel > Rapid Prototyper Panel** from Menu bar.
- 2 Specify name and file location of the new Rapid Prototyper project (.etp).
- 3 Click **Save** to insert the file in the current project and open it in Rapid Prototyper.

For information on designing Rapid Prototyper panels, refer to the *Rapid Prototyper User Documentation* distributed with the product.

Connecting Operator and Graphical Panel Interfaces

You can set I/O connections for each graphical panel referenced in the active project.

- [“Connecting I/Os between SCADE Suite and Graphical Panel\(s\)”](#)
- [“Importing connections from another operator”](#)
- [“Importing I/Os from graphical panel to SCADE Suite operator”](#)
- [“Exporting I/Os from SCADE Suite operator to graphical panel”](#)
- [“Locating I/Os in design models”](#)
- [“Deleting connections from Connections list”](#)
- [“Defining matching rules to connect I/Os by name \(Advanced\)”](#)

TYPE AND KIND COMPATIBILITY BETWEEN SCADE SUITE AND GRAPHICAL PANELS

Type compatibility and acceptable ranges between SCADE Suite I/Os and Graphical Panels I/Os are described below.

Table 3.2: Type mapping between SCADE Suite and graphical panels

SCADE Suite I/O Basic Type	Graphical Panel I/O Type	Comment
bool	bool	
char	char	
int8	int8	
int16	int16	
int32	int32	
int64	int64	
uint8	uint8	
uint16	uint16, wchar	
uint32	uint32	
uint64	uint64	

Table 3.2: Type mapping between SCADE Suite and graphical panels

SCADE Suite I/O Basic Type	Graphical Panel I/O Type	Comment
float32	float32	
float64	float64	
enum	int32	Only for Rapid Prototyper compatibility, direction =>
$X \wedge N$	$X \wedge N$	Where X is a scalar (not an enum) or array type, and N a dimension
structure of Inputs or Outputs	group: sequence of Inputs or Outputs	Each group element type must match with the corresponding structure element type

Kind compatibility and acceptable ranges are described below:

Table 3.3: Kind correspondence between SCADE Suite and graphical panels

SCADE Suite I/O Kind	Direction	Graphical Panel I/O Kind
Input	=>	Input
Input	<=	Output
Hidden	=>	Input
Hidden	<=	Output
Output	=>	Input
Probe	=>	Input
Sensor	=>	Input
Sensor	<=	Output

Connecting I/Os between SCADE Suite and Graphical Panel(s)

You can connect I/Os from the model root operator to one or several graphical panels declared in the project. The **Inputs/Outputs Connection View** allows managing all the connections at operator level. it also provides

To connect SCADE Suite I/Os to graphical panels I/Os

- 1 Right-click the root operator of the model in **Scade View** and select *Connect to Graphical Panels* from the contextual menu.

The **Inputs/Outputs Connection View** that opens lists all interfaces (inputs, outputs, sensors, and probes) of the selected operator, and all interfaces (inputs and outputs) of the graphical panels inserted in the project:

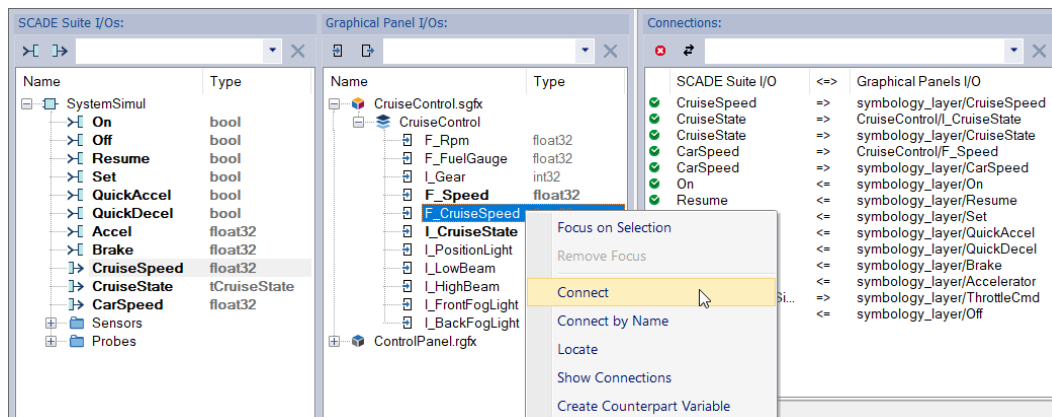


Figure 3.95: Connecting I/Os between SCADE Suite root operator and graphical panels

- 2 Select an item in **SCADE Suite I/Os** view and one or several items with a compatible kind and type (see [“Type and Kind Compatibility between SCADE Suite and Graphical Panels”](#) on page 306) in **Graphical Panels I/Os** view. Use **Ctrl** or **Shift** keys for multi-selection.
- 3 Right-click any selected item and select *Connect*. Alternatively, drag and drop an item from one view onto another item in the other view.

You cannot connect several panel outputs to a single input. Each authorized connection is added to the **Connections** list, with a status and the direction of the connection. Note that any change in a connected input/output/sensor may invalidate the connections using that item.

Several display, search, and filtering options are available as explained below.

To highlight items contained in a connection

- Select a connected item in **SCADE Suite I/Os** or **Graphical Panel I/Os** views (connected items are displayed in bold): the connections in which it appears and the item(s) to which it is connected are highlighted with a yellow background. Use **Ctrl** or **Shift** keys for multi-selection.
- Select a connection in the **Connections** view: the items its contains are highlighted with a yellow background in **SCADE Suite I/Os** and **Graphical Panel I/Os** views. Use **Ctrl** or **Shift** keys for multi-selection.

To search an item in a list

- Type the name of an item in the search field to show in the corresponding list only items that match the search. The subtree of elements of each matching item is also displayed. Search is done on the fly.

To filter displayed items in a list

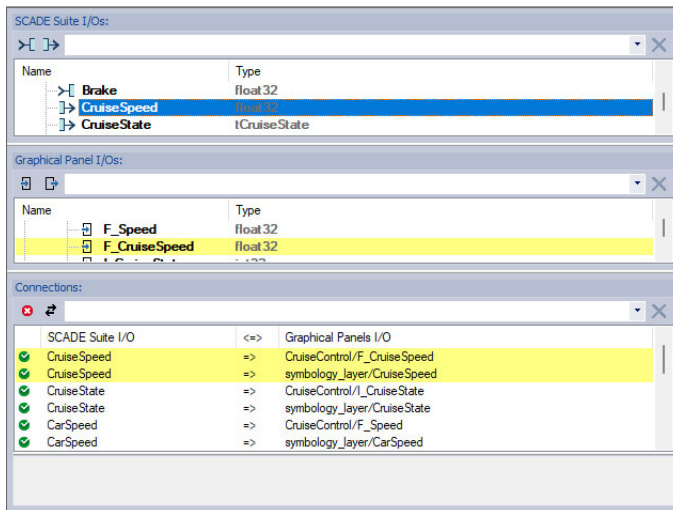
- At the top of **SCADE Suite IOs** view:
 - Click **Inputs**  to show only inputs, hidden inputs, and sensors.
 - Click **Outputs**  to show only outputs and probes.
 - Right-click an item and select *Show Connections* to show only connections containing that item in the **Connections** view.
- At the top of **Graphical Panel IOs** view:
 - Click  to show only input variables.

- Click  to show only output variables.
- Right-click an item and select *Show Connections* to show only connections containing that item in the **Connections** view.
- At the top of the **Connections** view:
 - Click  to show only broken or incomplete connections.
 - Click  to show all connections set between **SCADE Suite I/Os** or **Graphical Panel I/Os**.

To sort items in Connections view

- In the **Connections** view, click column titles to sort column content.

In case the **SCADE Suite I/Os**, **Graphical Panel I/Os**, or **Connections** views contain very long information, it is possible to widen their display.



To change the display of graphical panel connection information

- Right-click anywhere in views and select *Display Views as Rows* or *Display Views as Columns* depending on the current state of the display.

Importing connections from another operator

If connections exist in an another operator in current project, you can import those connections provided both operator share the same interfaces.

To import connections from another operator in the project

- Select *Insert > Graphical Panels > Import Connections* and select an operator containing connections in the **Connections Import** dialog.

Importing I/Os from graphical panel to SCADE Suite operator

When importing I/Os from graphical panels, imported I/Os are created in the operator with reversed direction (inputs become outputs and vice versa). If an I/O with the same name is already defined in the model, it is not imported.

To import I/Os from graphical panel to SCADE Suite operator

- Right-click an I/O in **Graphical Panels I/Os** and select *Create Counterpart Variable*.

Exporting I/Os from SCADE Suite operator to graphical panel

When exporting I/Os from operators, exported I/Os are created in the graphical panel with reversed direction (inputs become outputs and vice versa). They are added to the variables dictionary of the graphical panel if no variable has already the same name. If possible, a connection is created between the SCADE Suite I/O and the new graphical panel variable.

To export I/Os from SCADE Suite operator to graphical panel

- 1 Select the destination layer in **Graphical Panels I/Os**.
- 2 Select one or several I/Os (using Ctrl/Shift keys) in **SCADE Suite I/Os**.
- 3 Right-click the I/O selection and select *Create Counterpart Plug*.

Locating I/Os in design models

You can locate I/Os listed in **Inputs/Outputs Connection View** and highlight them in SCADE Suite models or Graphical Panel models.

To locate I/Os in design models

- 1 Right-click an item in **SCADE Suite I/Os** and select  *Show in Browser* to locate it, in **Scade View**, under the model element where it is declared.
- 2 Right-click an item in **Graphical Panels I/Os** and select *Locate* to locate it in the variable dictionary of the corresponding graphical panel. If closed, SCADE Display and SCADE Rapid Prototyper open themselves.

Deleting connections from Connections list

You can delete connections between an operator and graphical models.

To delete connections

- Right-click connections in **Connections** view and select *Delete* to remove selected connections only. Use **Ctrl** or **Shift** keys for multi-selection.
- Right-click any connection and select *Delete All* to remove all existing connections in current operator.

Defining matching rules to connect I/Os by name (Advanced)

This content is for advanced users. I/O connection by name does substitutions based on regular expression pattern matching. It relies on the definition of translation rules on I/O names before performing comparison and automatic I/O connections when names match.

Consider Name Mapping Rules below:

Name Rules Definition	
SCADE Suite I/O Name Translation Pattern: LoadDspArray[[0-9]+]\FCP_(.+) Replacement: \2_\1	
Graphical Panels I/O Name Translation Pattern: ^(F B I)_(.+)([0-9]+) Replacement: \2_\3	
Test I/O Name: LoadDspArray[1].FCP_varName Translation: varName_1 I/O Name: B_varName_1 Translation: varName_1	
OK Cancel	

Figure 3.96: Example of name mapping rules definition

In **SCADE Suite I/O Name Translation** area:

- \1 refers to the first group in parenthesis (**[0-9]+**), meaning *whichever combination of one or more digits* (e.g., 1, 2, ..., 9)
- \2 refers to the second group in parenthesis (**.+**), meaning *whichever combination of one or more characters* (e.g., **varName**)

The **Replacement** field underneath indicates a concatenation of \2, then underscore (_), then \1. Therefore, **LoadDspArray[1].FCP_varName** is translated into **varName_1** as displayed in the **Test** area.

In **Graphical Panels I/O Name Translation** area:

- \1 refers to the first group in parenthesis (**(F|B|I)**), meaning *either F, B, or I character*
- \2 refers to the second group in parenthesis (**.+**), meaning *whichever combination of one or more characters* (e.g., **varName**)
- \3 refers to the third group in parenthesis (**[[0-9]+]**), meaning *whichever combination of one or more digits* (e.g., 1, 2, ..., 9)

The **Replacement** field underneath indicates a concatenation of \2, then underscore (_), then \3. Therefore, **B_varName** is translated into **varName_1**. This Name Mapping Rule shows that SCADE Suite variable **LoadDspArray[1].FCP_varName** is mapped automatically with **B_varName_1** (provided they have the same type) as displayed in the **Test** area.

In **SCADE Suite/Graphical Panels I/O Connections** window, select all variables in the SCADE Suite array and all variables in the graphical panel (incl. those already connected), and click **Connect by Name**. All single items (except sub-structures requiring a more complex name mapping rule) of an array connect with the corresponding graphical panels I/Os.

For information about substitutions based on regular expression pattern matching, read about the tcl command "regsub" at:

- http://www.tcl.tk/man/tcl8.4/TclCmd/re_syntax.htm#3
- <http://www.regular-expressions.info>

You can define only one Name Mapping Rule per PC at a time (it is stored in Windows registry). For complex applications, several mapping patterns may be necessary. Therefore, it is recommended to store (in a simple text file) all Name Mapping Rules that are needed, to be able to reuse them later.

To define matching rules on names

- 1 Select *Project > Graphical Panels > Define Mapping Name Rules* to open the **Name Rules Definition** window.
- 2 Specify the pattern to be replaced and the replacement pattern.
- 3 Test the name rule in **I/O Name**.

To connect SCADE Suite and graphical panel inputs/outputs by name

- 1 Specify the items to connect by name by selecting either:
 - I/Os in **SCADE Suite I/Os** and **Graphical Panels I/Os** to process only selected items.
 - no item in **SCADE Suite I/Os** to process all operator interfaces.
 - no item in **Graphical Panels I/Os** to process all panel interfaces.
- 2 Click **Connect by Name**. The **Connections** view displays the new connection(s) according to the patterns defined in **Name Rules Definition**.

4 / Designing with Predefined Operators and Constructs

This chapter covers the design of SCADE Suite models with the help predefined operators and constructs you can use to define data flows and control flows.

- [“Designing with Arithmetic Operators”](#)
- [“Designing with Bitwise Arithmetic Operators”](#)
- [“Designing with Choice Operators”](#)
- [“Designing with Comparison Operators”](#)
- [“Designing with Logical Operators”](#)
- [“Designing with Temporal Operators”](#)
- [“Designing with Structured Values”](#)
- [“Designing with Array Values”](#)
- [“Designing with Array Iterators”](#)
- [“Designing with Operator Activation”](#)
- [“Designing with Restart Instantiation”](#)

Designing with Arithmetic Operators

When designing models, you may need to create arithmetic operations on data flows. In SCADE Suite, it is possible to create arithmetic operations and manipulate mathematical data like any other flow.

Note

A set of arithmetic constructs can be created with predefined primitives from the Mathematical toolbar. The syntax and semantics of such primitives are detailed in [“Arithmetic Operators”](#) on page 102 of the *Scade Language Reference Manual*.

You can create the following arithmetic constructions:

- [“Setting and Designing Additions”](#)
- [“Setting and Designing Subtractions”](#)
- [“Setting and Designing Multiplications”](#)
- [“Setting and Designing Polymorphic Divisions”](#)
- [“Setting and Designing Modulos”](#)
- [“Setting and Designing Unary Minus”](#)
- [“Setting and Designing Numeric Cast”](#)

Setting and Designing Additions

You can create **Plus** predefined operators to sum numeric flow values.

To define Plus operators

- 1 Click **New Plus**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

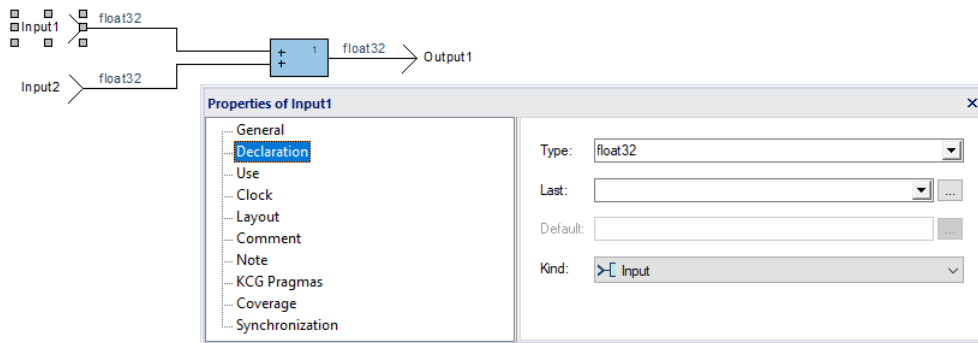


Figure 4.1: Setting numeric type on Plus inputs

- 4 Set the output type with the same type as inputs.

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Subtractions

You can create **Minus** predefined operators to subtract numeric flow values.

To define Minus operators

- 1 Click **New Minus**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

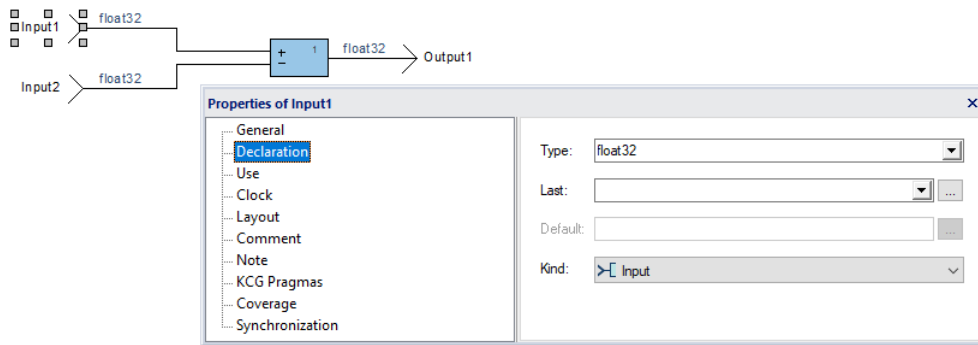


Figure 4.2: Setting numeric type on Minus inputs

- 4 Set the output type with the same type as inputs.

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Multiplications

You can create **Multiplication** predefined operators to multiply numeric flow values.

To define Multiplication operators

- 1 Click **New Multiplication**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

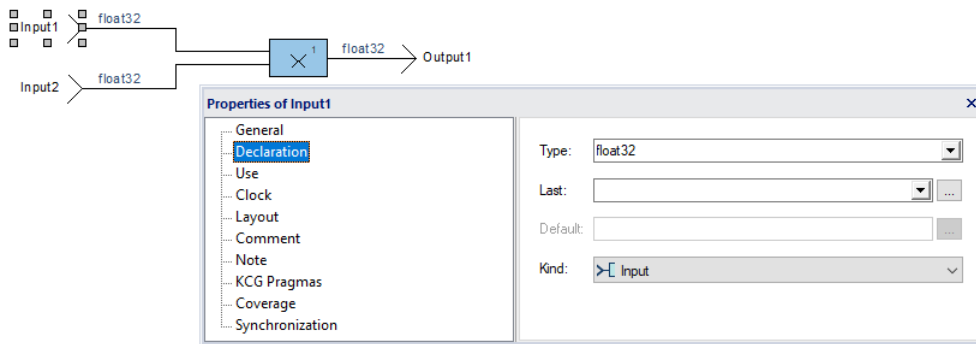


Figure 4.3: Setting numeric type on Multiplication inputs

- 4 Set the output type with the same type as inputs.

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Polymorphic Divisions

You can create **Polymorphic Division** predefined operators to divide numeric flow values.

To define Polymorphic Division operators

- 1 Click **New Polymorphic Division**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set input flows with a numeric type in **Declaration** properties.

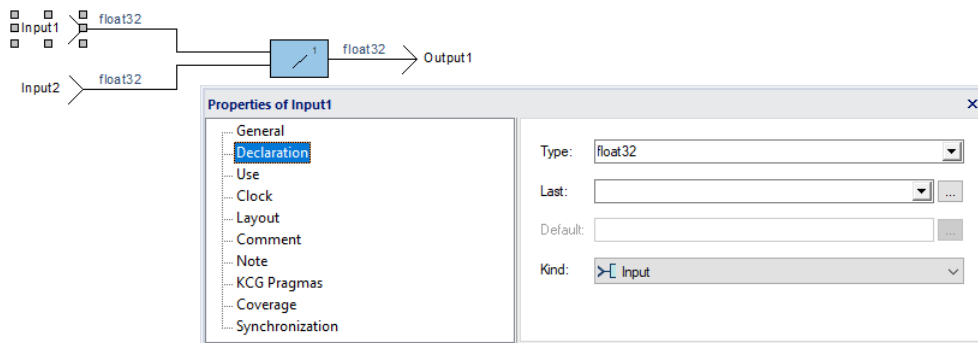


Figure 4.4: Setting type on Polymorphic Division inputs

- 4 Set the output type with the same type as inputs.

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Modulos

You can create **Modulo** predefined operators to define the remainder of a division between integer values.

To define Modulo operators

- 1 Click **New Modulo**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as integer (for instance `int32`) in **Declaration** properties.

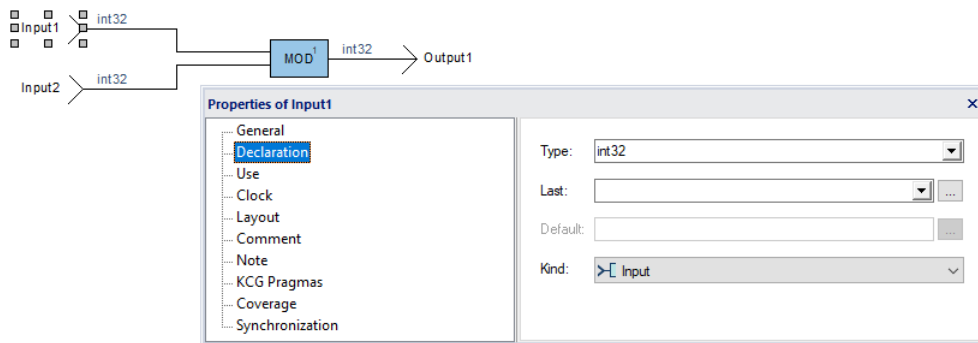


Figure 4.5: Setting integer type on Modulo inputs

- 4 Set the output type with the same type as inputs.

Setting and Designing Unary Minus

You can create **Unary Minus** predefined operators to invert the sign of numeric flow values.

To define Minus operators

- 1 Click **New Unary Minus**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input and output flows to the operator.
- 3 Right-click the input (and output), select *Properties*, and set the type of both input/output flows as numeric in **Declaration** properties.

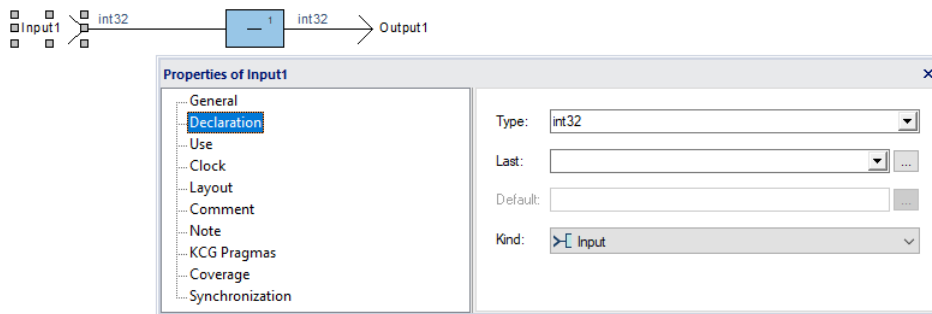


Figure 4.6: Setting numeric type on Unary Minus inputs

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Numeric Cast

You can create **Numeric Cast** predefined operators to convert a numerical expression of a numeric type into another numeric type.

To define Numeric Cast operators

- 1 Click **New Numeric Cast**  in **Mathematical** toolbar and click in diagram.
- 2 Connect the input and output flows to the operator.
- 3 Display the **Properties** docking window and specify types:
 - Select the cast operator and set **Type** as numeric in **Parameters** Properties. It is possible to set the value directly on the hidden parameter at bottom of the operator symbol.
 - Select the input and set **Type** as numeric in **Declaration** properties.
 - Select the output and set **Type** as the cast operator.

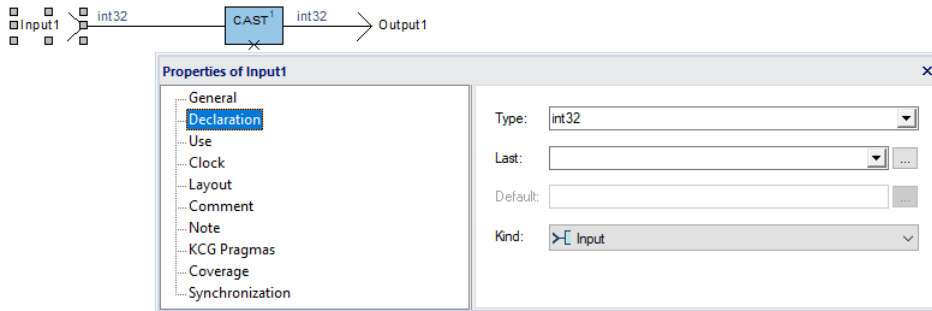


Figure 4.7: Setting numeric types for Numeric Cast operator design

Designing with Bitwise Arithmetic Operators

When designing models, you may need to define bitwise arithmetic operations on numeric flows. In SCADE Suite, it is possible to create bitwise arithmetic operations and manipulate numeric data like any other flow. All operations require unsigned integer inputs/outputs.

Note

A set of bitwise arithmetic constructs can be created with predefined primitives from the Bitwise toolbar. The syntax and semantics of such primitives are detailed in [“Arithmetic Operators”](#) on page 102 of the *Scade Language Reference Manual*.

You can create the following bitwise arithmetic constructions:

- [“Setting and Designing Bitwise AND”](#)
- [“Setting and Designing Bitwise OR”](#)
- [“Setting and Designing Bitwise XOR”](#)
- [“Setting and Designing Bitwise NOT”](#)
- [“Setting and Designing Logical Left Shift”](#)
- [“Setting and Designing Logical Right Shift”](#)

Setting and Designing Bitwise AND

You can create **Bitwise AND** predefined operators to perform logical conjunction on numeric flows.

To define bitwise AND operators

- 1 Click **New Bitwise And**  in **Bitwise** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

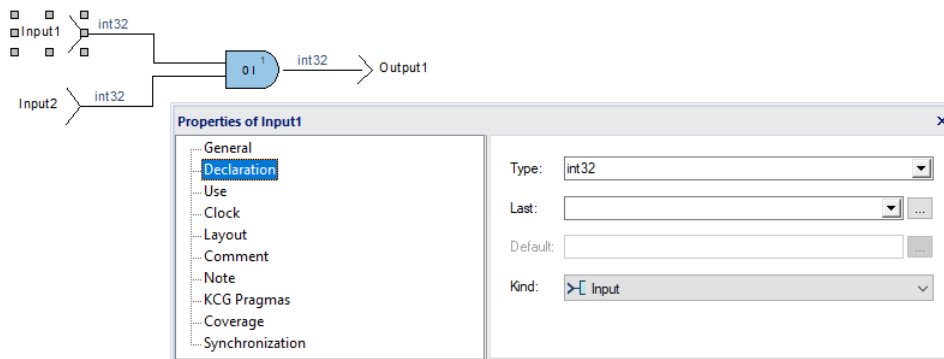


Figure 4.8: Setting type on Bitwise AND inputs/output

Setting and Designing Bitwise OR

You can create **Bitwise OR** predefined operators to perform logical disjunction on numeric flows.

To define bitwise OR operators

- 1 Click **New Bitwise Or**  in **Bitwise** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

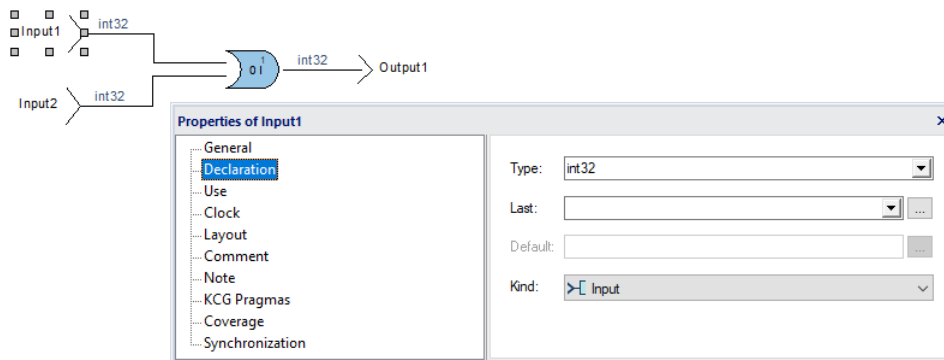


Figure 4.9: Setting type on Bitwise OR inputs/output

Setting and Designing Bitwise XOR

You can create **Bitwise XOR** predefined operators to perform logical exclusive disjunction on numeric flows.

To define bitwise XOR operators

- 1 Click **New Bitwise Exclusive Or**  in **Bitwise** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

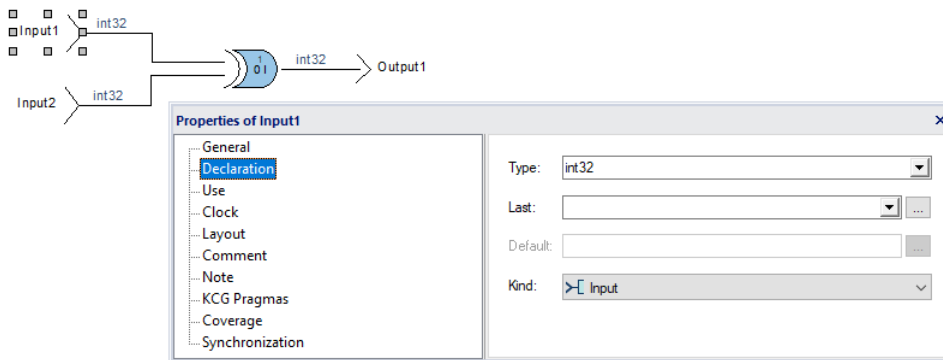


Figure 4.10: Setting type on Bitwise XOR inputs/output

Setting and Designing Bitwise NOT

You can create **Bitwise Not** predefined operators to perform logical negation on numeric flows.

To define bitwise NOT operators

- 1 Click **New Bitwise Not**  in **Bitwise** toolbar and click in diagram.
- 2 Connect both input and output flows to the operator.
- 3 Select both input/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

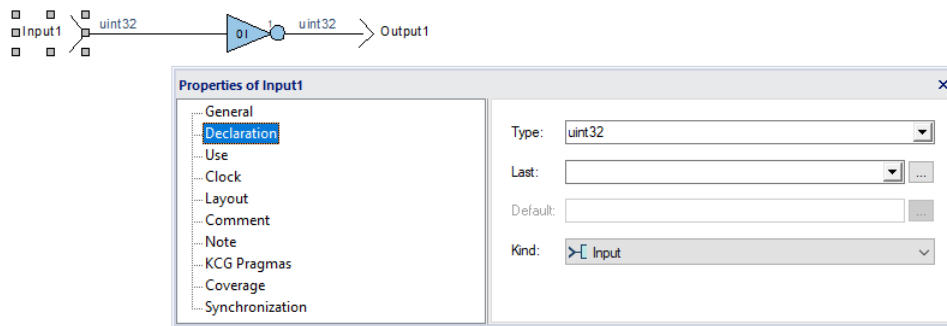


Figure 4.11: Setting type on Bitwise NOT input/output

Setting and Designing Logical Left Shift

You can create **Logical Left Shift** predefined operators to perform logical left shift on numeric flows.

To define bitwise logical left shift operators

- 1 Click **New Logical Left Shift**  in **Bitwise** toolbar and click in diagram.
- 2 Right-click the operator, select *Properties*, and specify an integer value for the shift in **Parameters** properties. It is possible to set the value directly on the hidden parameter at bottom of the operator symbol or to connect a flow containing the appropriate value.
- 3 Connect both input and output flows to the operator.
- 4 Select both input/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

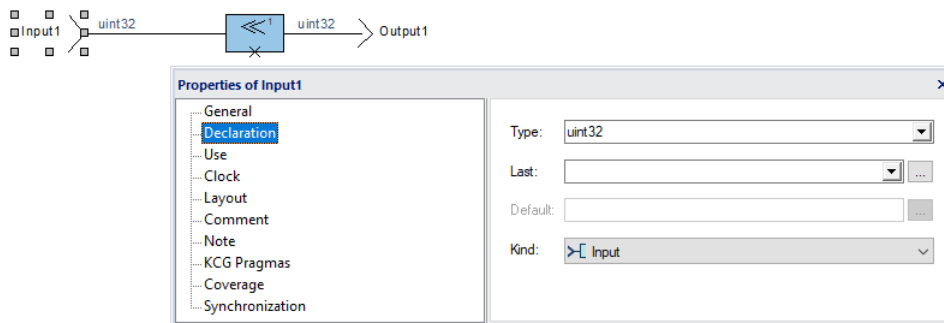


Figure 4.12: Setting type on Logical Left Shift input/output

Setting and Designing Logical Right Shift

You can create **Logical Right Shift** predefined operators to perform logical right shift on numeric flows.

To define bitwise logical right shift operators

- 1 Click **New Logical Right Shift**  in **Bitwise** toolbar and click in diagram.
- 2 Right-click the operator, select *Properties*, and specify an integer value for the shift in **Parameters** properties. It is possible to set the value directly on the hidden parameter at bottom of the operator symbol or to connect a flow containing the appropriate value.
- 3 Connect both input and output flows to the operator.
- 4 Select both input/output, right-click and select *Properties*, and set the type of input and output flows with the same unsigned integer type in **Declaration** properties.

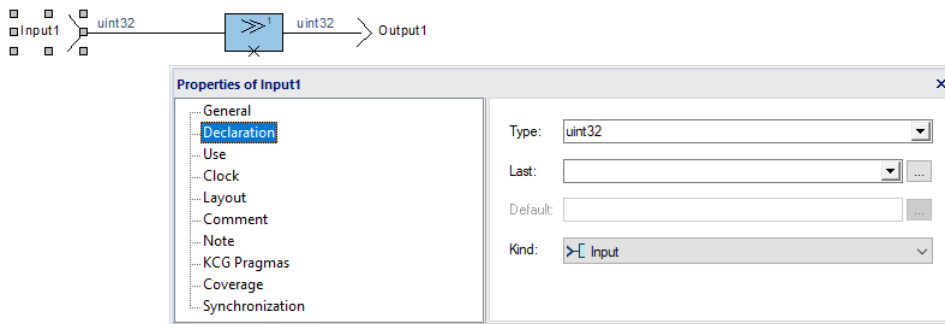


Figure 4.13: Setting type on Logical Right Shift input/output

Designing with Choice Operators

When designing models, you may need to define conditional operations on data flows to make a decision. In SCADE Suite, it is possible to create flow switches and manipulate decision branches like any other flow.

Note

A set of constructs can be created with predefined primitives from the Choice toolbar. The syntax and semantics of such primitives are detailed in [“Flows Switches”](#) on page 106 of the *Scade Language Reference Manual*.

You can create the following conditional constructions:

- [“Setting and Designing If Then Else”](#)
- [“Setting and Designing Case”](#)

Setting and Designing If Then Else

You can create **If..Then..Else** predefined operators to make a decision between pairs of decision input flows (then and else branches) based on a Boolean conditional flow (if branch).

To define If..Then..Else decision branches

- 1 Click **New If..Then..Else**  in **Choice** toolbar and click in diagram.

Tip

Modify the number of decision branches with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 2 Connect the inputs/outputs as follows:
 - Connect the conditional flow (if branch) to the top input of the operator.
 - Connect the other input flows by pairs (then and else branches).
 - Connect the output flow (single pair of branches) or each output flow (corresponding to each pair of decision branches) to the operator.

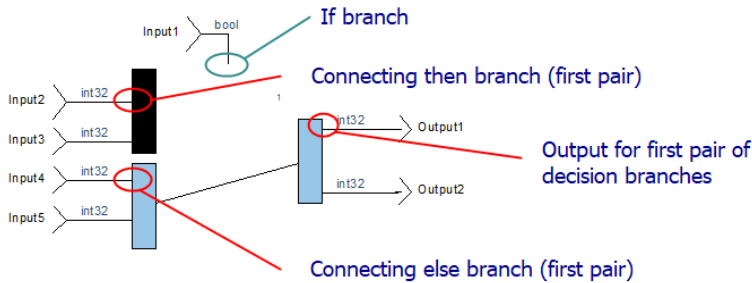


Figure 4.14: Designing pairs of decision branches on If..Then..Else predefined operator

- 3 Right-click the condition input, select *Properties*, and set the type of this input flow as Boolean (bool) in **Declaration** properties.
- 4 Set the type of each pair of branches (then and else branch) with the same type: any type including imported types.
- 5 Set the type of the output(s) with the same type as the corresponding pair of branch (then and else branch) inputs.

Setting and Designing Case

You can create **Case** predefined operators to evaluate a condition with regard to a number of decision patterns (and input flows) covering every possible case of the condition. When the condition gets the value of one of the patterns, the result is that of the corresponding input flow. When no pattern matches the evaluated condition, the result is that of the default input flow.

Note

A Case has a Default input flow to assign a default value to the operator output when condition values are not verified. This default value is optional.

To define Case operators

- 1 Click **New Case**  in **Choice** toolbar and click in diagram.

Tip

Modify the number of decision patterns with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 2 Connect the inputs/outputs as follows:
 - Connect the conditional flow to the top input of the operator.
 - Connect to operator inputs as many decision patterns as condition cases.
 - Connect (when needed) to last operator input a default decision flow.

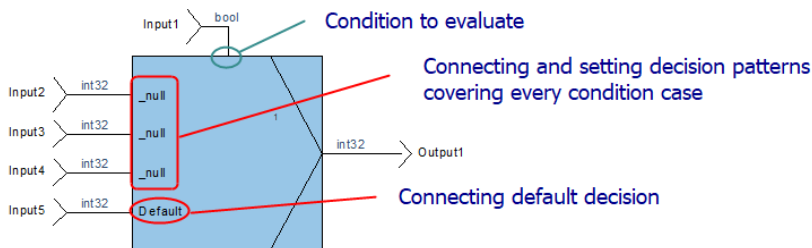


Figure 4.15: Designing condition and decision patterns of Case predefined operator

- 3 Right-click the condition input, select *Properties*, and set the type of this input flow as enumeration, integer (int), or character (char) in **Declaration** properties.

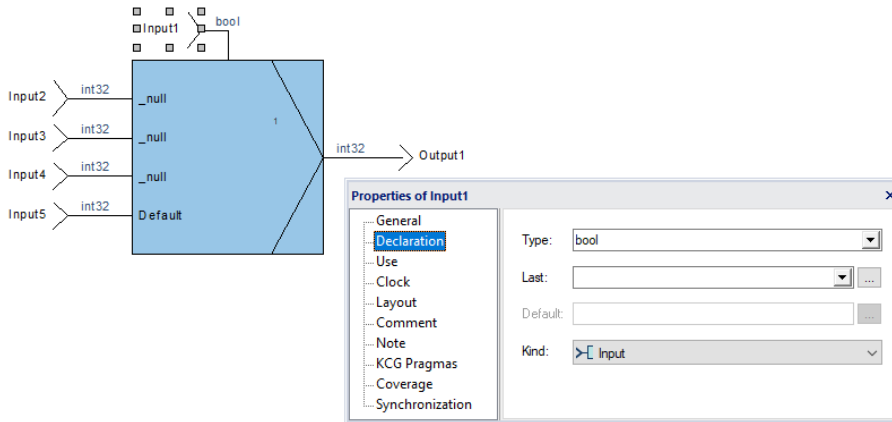


Figure 4.16: Setting type on Case conditional input

- 4 Set the type of each input flow associated with each decision pattern (including default) with the same type: any type including imported and structured types.
- 5 Set the type of the output with the same type as input flows.
- 6 Right-click the Case operator instance in the diagram, select *Properties*, then select **Parameters** and set decision patterns on each line as follows:
 - Specify each pattern with a different value of the same type as the condition input flow. Patterns must cover exhaustively all condition cases.
 - Unselect **Default** to set the operator without default decision.

Note

Default is mandatory when the operator has an integer or character condition. Enumerated conditions can omit the default decision. When removing the default pattern, make sure all enumeration and Boolean values are represented.

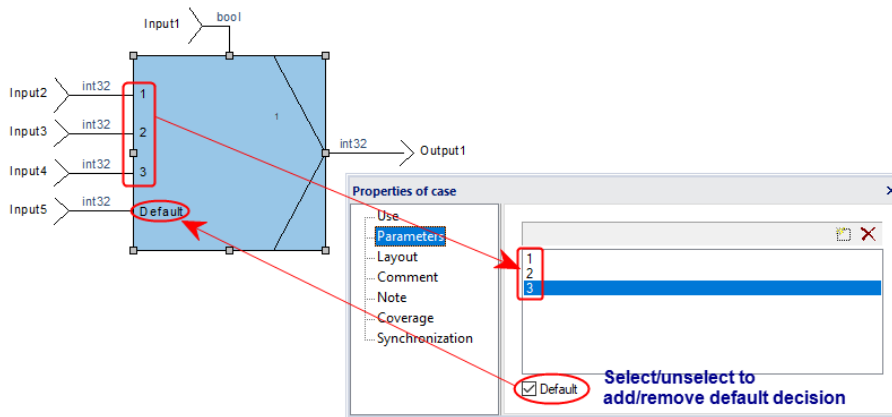


Figure 4.17: Setting decision patterns as Parameters of Case operator

Note

Adding new parameters from the Properties is equivalent to adding an additional input on the operator symbol to set a decision pattern and connect the associated input flow.

Designing with Comparison Operators

When designing models, you may need to define comparison operations on data flows. In SCADE Suite, it is possible to create relational operations and compare data like any other flow.

Note

A set of relational constructs can be created with predefined primitives from the Comparison toolbar. The syntax and semantics of such primitives are detailed in [“Relational Operators”](#) on page 105 of the *Scade Language Reference Manual*.

You can create the following relational constructions:

- [“Setting and Designing Strictly Less Than”](#)
- [“Setting and Designing Less Than or Equal”](#)
- [“Setting and Designing Strictly Greater Than”](#)
- [“Setting and Designing Greater Than or Equal”](#)
- [“Setting and Designing Different”](#)
- [“Setting and Designing Equal”](#)

Setting and Designing Strictly Less Than

You can create **Strictly Less Than** predefined operators to compare numeric flow values for lower value.

To define Strictly Less Than operators

- 1 Click **New Strictly Less Than**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

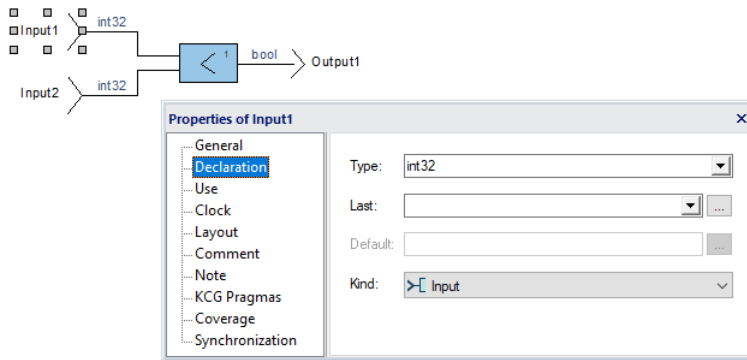


Figure 4.18: Setting numeric type on Strictly Less Than inputs

- 4 Set the output type as Boolean (bool).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Less Than or Equal

You can create **Less Than or Equal** predefined operators to compare numeric flow values for lower or equal value.

To define Less Than or Equal operators

- 1 Click **New Less Than or Equal**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

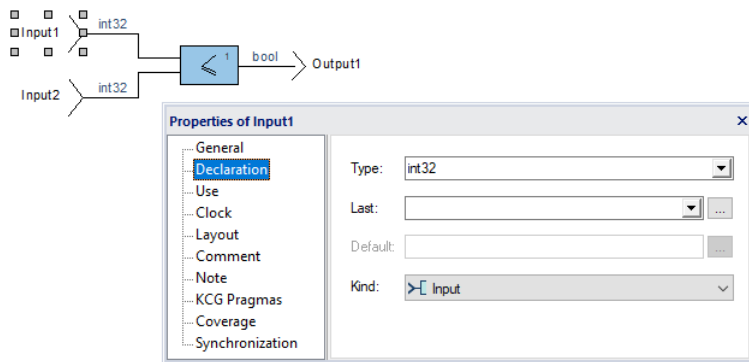


Figure 4.19: Setting numeric type on Less Than or Equal inputs

- 4 Set the output type as Boolean (bool).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Strictly Greater Than

You can create **Strictly Greater Than** predefined operators to compare numeric flow values for greater value.

To define Strictly Greater Than operators

- 1 Click **New Strictly Greater Than**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

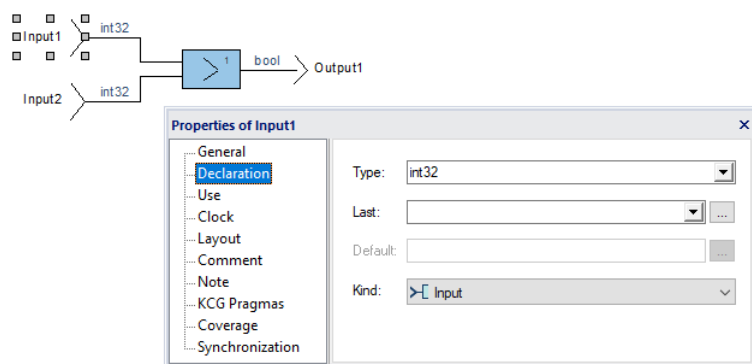


Figure 4.20: Setting numeric type on Strictly Greater Than inputs

- 4 Set the output type as Boolean (bool).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Greater Than or Equal

You can create **Greater Than or Equal** predefined operators to compare numeric flow values for greater or equal value.

To define Greater Than or Equal operators

- 1 Click **New Greater Than or Equal**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows as numeric in **Declaration** properties.

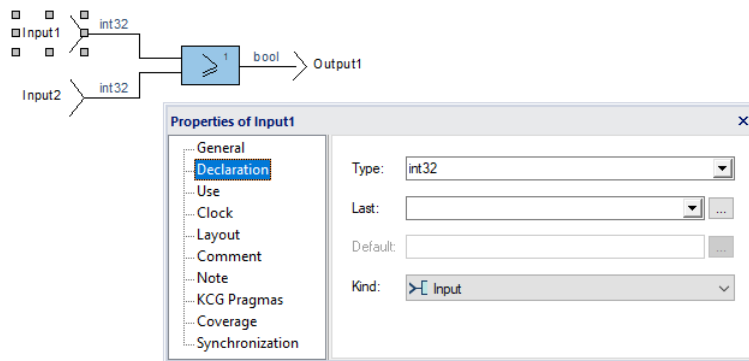


Figure 4.21: Setting numeric type on Greater Than or Equal inputs

- 4 Set the output type as Boolean (bool).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Different

You can create **Different** predefined operators to compare flow values for difference between values.

To define Different operators

- 1 Click **New Different**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows with any predefined type (`intN`, `uintN`, `floatN`, `bool`, `char`) in **Declaration** properties.

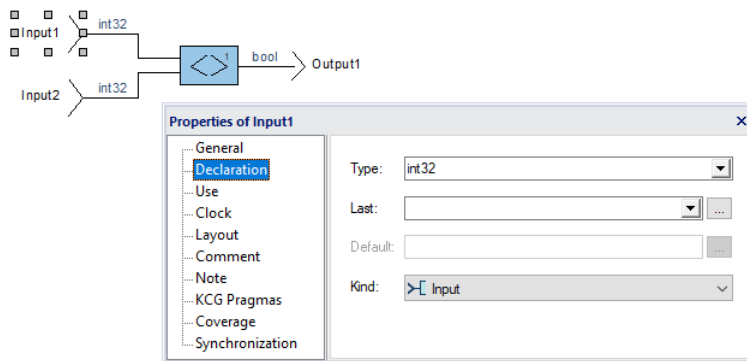


Figure 4.22: Setting predefined types on Different inputs

- 4 Set the output type as Boolean (`bool`).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Setting and Designing Equal

You can create **Equal** predefined operators to compare flow values for difference between values.

To define Equal operators

- 1 Click **New Equal**  in **Comparison** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Right-click an input, select *Properties*, and set the type of input flows with any predefined type (`intN`, `uintN`, `floatN`, `bool`, `char`) in **Declaration** properties.

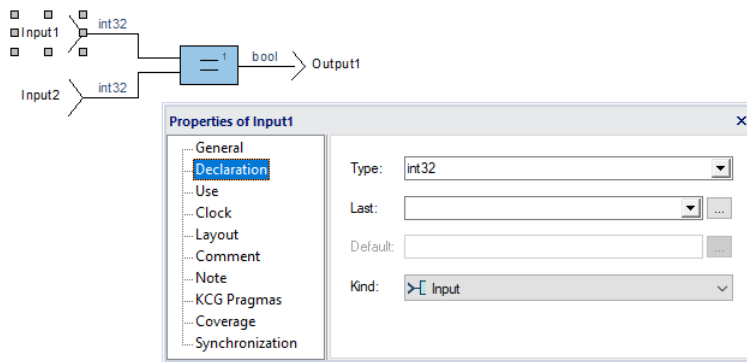


Figure 4.23: Setting predefined types on Equal inputs

- 4 Set the output type as Boolean (`bool`).

Tip

For generic typing, set input and output type as 'T' in Declaration properties and declare this generic type as numeric in the Type Variables properties of the operator declaration containing the predefined operator. For details, see [“Declaring Operators Using Generic Types”](#) on page 216.

Designing with Logical Operators

When designing models, you may need to define logical operations on data flows. In SCADE Suite, it is possible to create Boolean logical operations and manipulate Boolean data like any other flow.

Note

A set of Boolean constructs can be created with predefined primitives from the Logical toolbar. The syntax and semantics of such primitives are detailed in [“Flows Switches”](#) on page 106 of the *Scade Language Reference Manual*.

You can create the following Boolean logical constructions:

- [“Setting and Designing AND”](#)
- [“Setting and Designing OR”](#)
- [“Setting and Designing XOR”](#)
- [“Setting and Designing Sharp”](#)
- [“Setting and Designing NOT”](#)

Setting and Designing AND

You can create **AND** predefined operators to perform logical conjunction on Boolean flows. The output is True when all inputs are True.

To define AND logical operation

- 1 Click **New And**  in **Logical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows as Boolean (bool) in **Declaration** properties.

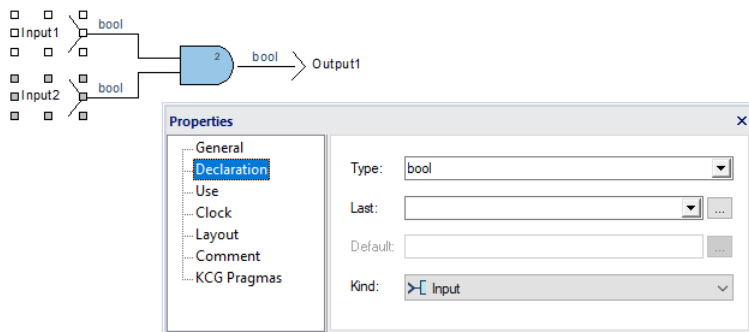


Figure 4.24: Setting Boolean type on AND inputs/output

Setting and Designing OR

You can create **OR** predefined operators to perform logical disjunction on Boolean flows. The output is `True` when at least one input is `True`.

To define OR logical operation

- 1 Click **New Or**  in **Logical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows as Boolean (`bool`) in **Declaration** properties.

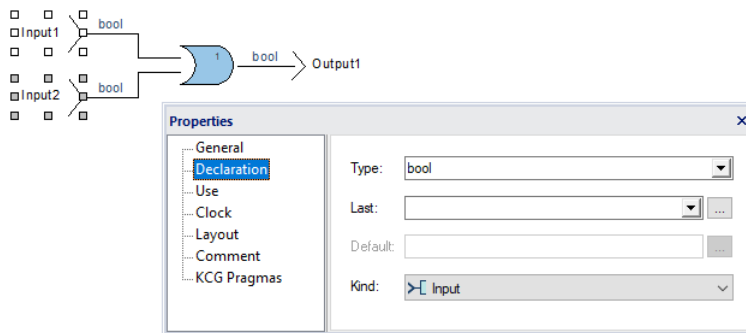


Figure 4.25: Setting Boolean type on OR inputs/output

Setting and Designing XOR

You can create **XOR** predefined operators to perform exclusive disjunction on Boolean flows. The output is **True** when an odd number of inputs is **True**.

To define XOR logical operation

- 1 Click **New Exclusive Or**  in **Logical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.
- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows as Boolean (**bool**) in **Declaration** properties.

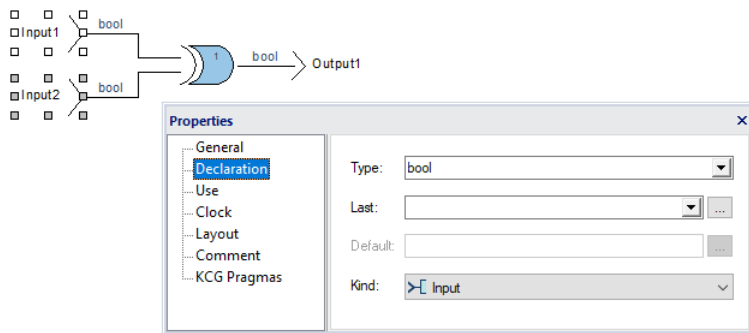


Figure 4.26: Setting Boolean type on XOR inputs/output

Setting and Designing Sharp

You can create **Sharp** predefined operators to perform a special exclusive operation on Boolean flows. The output is `True` when at most one input (zero or one) is `True`.

To define Sharp logical operation

- 1 Click **New Sharp**  in **Logical** toolbar and click in diagram.
- 2 Connect the input flows and the output flow to the operator.

Tip

Modify the number of operator operands with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Select all inputs/output, right-click and select *Properties*, and set the type of input and output flows as Boolean (`bool`) in **Declaration** properties.

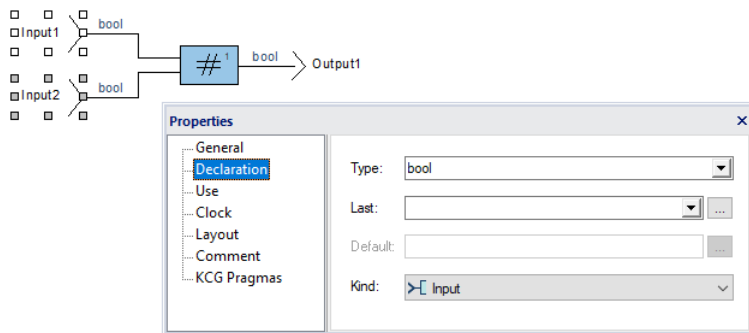


Figure 4.27: Setting Boolean type on Sharp inputs/output

Setting and Designing NOT

You can create **NOT** predefined operators to perform negation on Boolean flows. The output is True when the input is False.

To define NOT logical operation

- 1 Click **New Not** ➤ in **Logical** toolbar and click in diagram.
- 2 Connect the input and output flows to the operator.
- 3 Select both input/output, right-click and select *Properties*, and set the type of input and output flows as Boolean (bool) in **Declaration** properties.

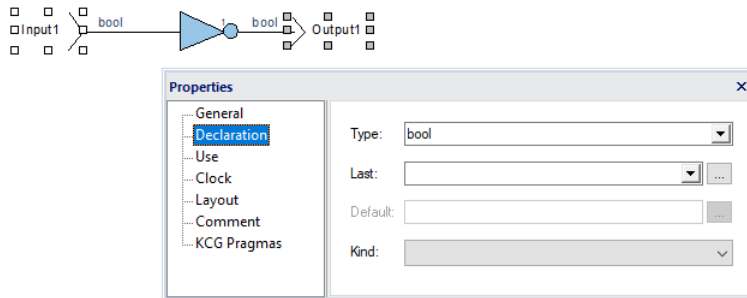


Figure 4.28: Setting Boolean type on NOT input/output

Designing with Temporal Operators

When designing models, you may need to define temporal (or sequential) operations on data flows. In SCADE Suite, it is possible to create sequential operations and manipulate past data like any other flow.

Note

A set of temporal constructs can be created with predefined primitives from the Time toolbar. The syntax and semantics of such primitives are detailed in [“Sequential Operators”](#) on page 94 of the *Scade Language Reference Manual*.

You can create the following temporal constructions:

- [“Setting and Designing Init”](#)
- [“Setting and Designing Previous”](#)
- [“Setting and Designing Followed By”](#)
- [“Setting and Designing Times”](#)
- [“Setting and Designing When”](#)
- [“Setting and Designing Merge of Flows”](#)

Setting and Designing Init

You can create **Init** predefined operators (a.k.a. `Init`) to initialize the value of a flow (or several flows) at first cycle.

Note

In combination with a Previous operator, the Init operator fixes the initial value of flows and allows to build deterministic applications. For design details, see [“Setting and Designing Previous”](#) on page 352.

To define Init operators

- 1 Click **New Init**  in **Time** toolbar and click in diagram.
- 2 Connect the input flow(s) and output flow(s) to the operator.

Tip

Modify the number of initialized flows with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 3 Set the type of each pair of input and output flow with the same type. The type of the corresponding initial value must be the same.
- 4 Set the initial value of each input flow either:
 - by connecting an input flow to the corresponding hidden input.

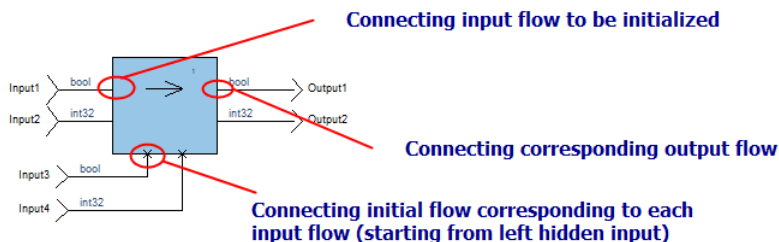


Figure 4.29: Designing pairs of input flows with initialization values

- by specifying parameter values from Init instance properties (see below).

To set initial values as parameters of Init operator instances

- 1 Right-click the operator instance in the **Diagram View** and select **Properties**.
- 2 Specify the initial value(s) of the corresponding input flow(s) in **Parameters** properties. Initial values are listed from top in the order of operator inputs (starting from upper pin on symbol).

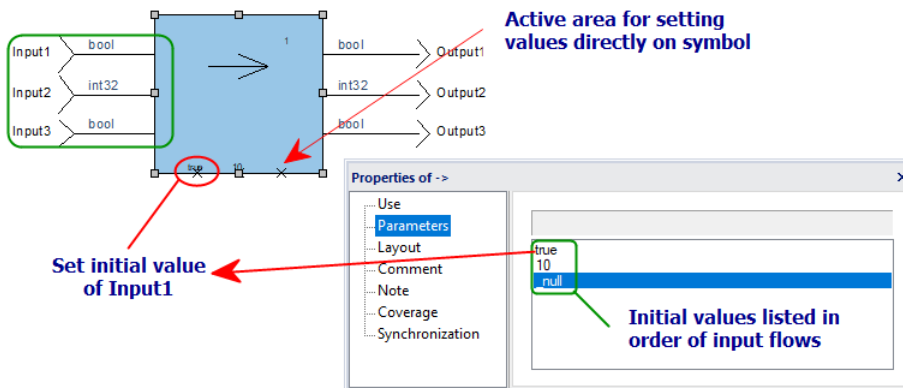


Figure 4.30: Parameters of Init operators

Tip

Click above any hidden input mark on symbol to enter each initial values directly on the editing area corresponding to each hidden input definition.

Setting and Designing Previous

You can create **Previous** predefined operators (a.k.a. Pre) to access to the previous value(s) of flow(s) with undefined value at the first cycle of activation. The value of the first cycle must be defined by means of an **Init** operator.

To define Pre operators

- 1 Click **New Previous**  in **Time** toolbar and click in diagram.

Tip

Modify the number of input/output flows with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 2 Connect the input flow(s) to the operator and connect the output flow(s) to an **Init** operator to initialize each corresponding input flow at first cycle.
- 3 Set the initial value of the previous input flow(s) on the **Init** operator (see [“Setting and Designing Init”](#) on page 350).
- 4 Set the type of the input flow(s). The type of each initial value and that of the corresponding output flow are the same.

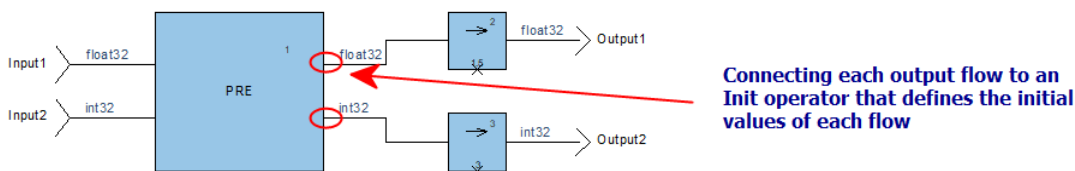


Figure 4.31: Designing previous input flows with initialization values

Setting and Designing Followed By

You can create **Followed By** predefined operators (a.k.a. FBY) to access the previous values of well-initialized flows. The use of this operator is equivalent to combining the **Previous** and **Init** predefined operators.

Note

When designing such operators, you set initial values and a cycle delay during which an input flow is evaluated to its initial value.

To define FBY operators

- 1 Click **New Followed By**  in **Time** toolbar and click in diagram.

Tip

Modify the pairs of input/output flows with additional input pins directly from Use properties as detailed in [“Editing Use Properties of Operator Calls”](#) on page 448.

- 2 Connect the input flow(s) and output flow(s) to the operator.
- 3 Set the type of each pair of input and output flow with the same type. The type of the corresponding initial value must be the same.

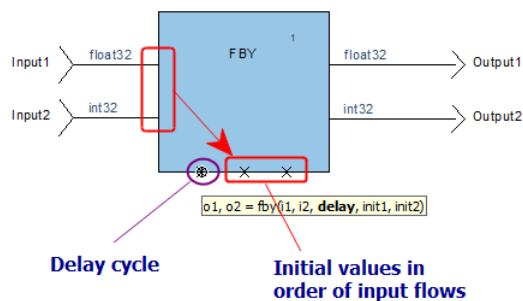


Figure 4.32: Designing FBY operator with multiple input flows and initial values

- 4 On the first hidden input (from left of symbol), specify the cycle delay as a strictly positive integer.

Tip

Click above the first hidden input mark on symbol to enter the delay value directly on the editing area.

- 5 Set the initial value(s) of the input flow(s) on remaining hidden input(s):
- by connecting an input flow to the corresponding hidden input.
 - by specifying parameter values from FBY instance properties.

Note

It is possible to specify complex expressions to the condition that the expression matches a static expression syntax. For example, $\text{Length} * \text{Width}$ (where Length and Width are real constants) is valid as a parameter for a real hidden input.

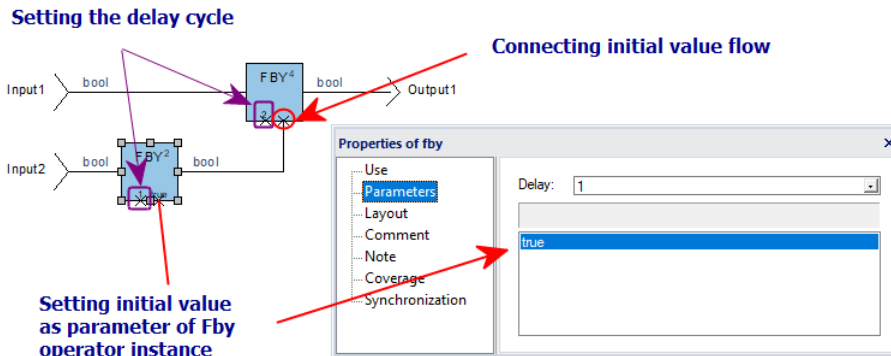


Figure 4.33: Designing Fby input flows with cycle delay and initialization values

Setting and Designing Times

You can create **Times** predefined operators to count down the repeated occurrence of a given condition before triggering an event. A single counter can be defined in Times operators.

To define Times operators

- 1 Click **New Times**  in **Time** toolbar and click in diagram.
- 2 Select the operator instance, click below hidden input mark to enable the editing zone, and specify the counter on the hidden input as an integer value
- 3 Connect the expression of the condition to the operator input. The condition is a Boolean expression.

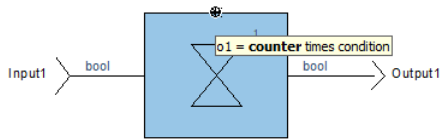


Figure 4.34: Example of Times operator

Tip

You can set the counter via the Parameter properties of the operator instance.

RELATED TOPICS

- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

Setting and Designing When

You can create **When** predefined operators to filter flows on particular cycles or synchronize flows sampled on different rates. You must set the operator using specific clock expression properties and design incoming flows as local variables and outgoing flows with respect to clock expression.

Note

By default, you cannot create When operators. To be able to design such language construct, go to *Tool > Options* menu, in Language options expand the Time list, and check When. You set parameter values which represent a clock expression. It is possible to define three distinct representations that reflect the clock expression.

To define When operators

- 1 Click **New When**  in **Time** toolbar and click in diagram.
- 2 Right-click the operator instance in the **Diagram View** and select *Properties*.
- 3 Select **Use** on the left and set required properties on the right.

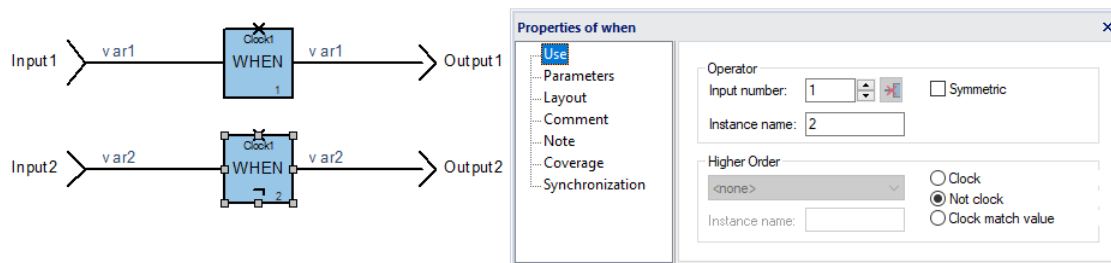


Figure 4.35: Tuning when activation with clock expressions

- 4 Select the clock expression rule associated with the operator:
 - **Clock** (default): to execute the operator when the specified clock is true.
 - **Not clock**: to execute the operator when the specified clock is false.

- **Clock match value:** to execute the operator when the clock matches a specified value.

5 Select **Parameters** on the left and set required properties on the right.

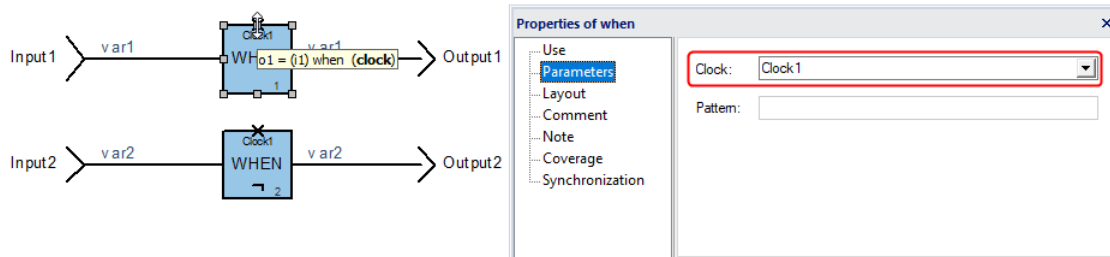


Figure 4.36: Parameters of When operator with default clock expression

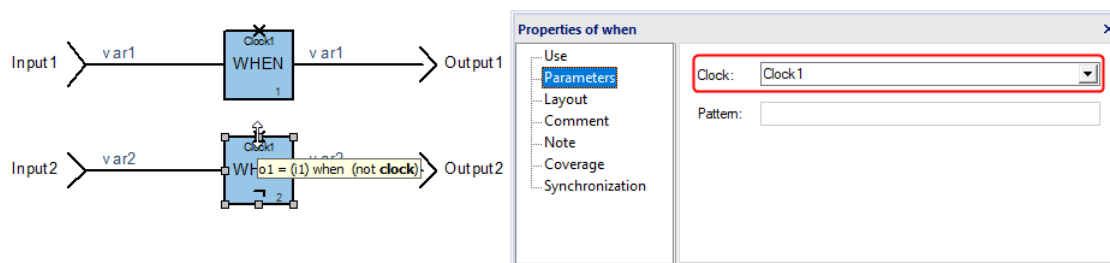


Figure 4.37: Parameters of When operator with not clock expression

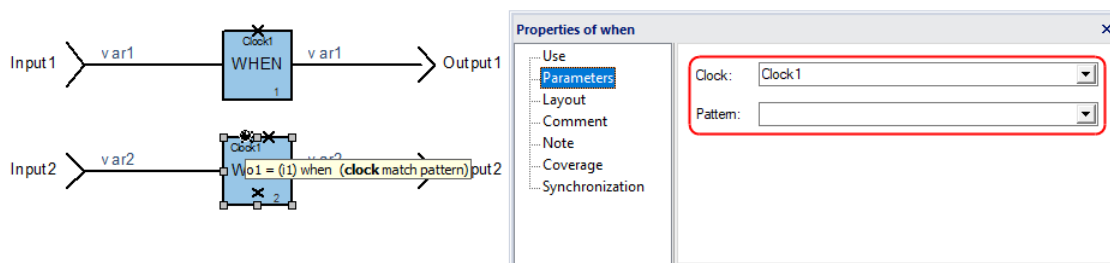


Figure 4.38: Parameters of When operator with clock match value expression

6 In the **Clock** field, specify the clock variable that assigns the value of the incoming flow to the outgoing flow with respect to the clock expression.

- 7 In the **Pattern** field, specify the value with which the clock variable has to match with. The value must be a Boolean or enumerated value.

Note

Required only for When operators set with `clock match value` expression

- 8 Set the incoming flow of the operator using local variables:
- Right-click the wire of an incoming flow and select *Properties*.
 - Select **Define** on the left and set a local variable as follows: check **Named Variable Definition**, type the variable name in the **Local Variable** field or select it from the drop-down list.

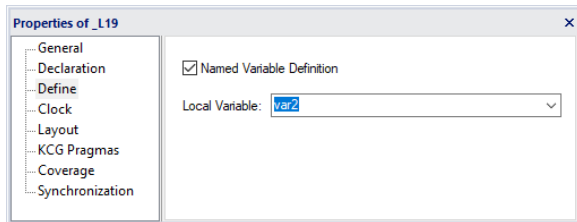


Figure 4.39: Defining local variables on incoming flow for When operator

- 9 Set the outgoing flow of the operator using local variables:
- Right-click the wire of an outgoing flow and select *Properties*.
 - Select **Define** on the left and set a local variable as follows: check **Named Variable Definition**, type the variable name in the **Local Variable** field or select it from the drop-down list.

Note

The local variable selected for outgoing flows must be different from the one defined for incoming flows on [Figure 4.39 for instance](#).

- Select **Clock** on the left and type the clock expression of the incoming flow in the **When** field or select it from the drop-down list.

Setting and Designing Merge of Flows

You can create **Merge** predefined operators to build a faster flow from slower ones. You must set the operator using a local clocked variable and design incoming flows with respect to clock expression. The resulting design is illustrated in [Figure 4.40](#).

Note

By default, you cannot create Merge operators. To be able to design such language construct, go to *Tool > Options* menu, in Language options expand the Time list, and check Merge. You set parameter values which represent a clock expression.

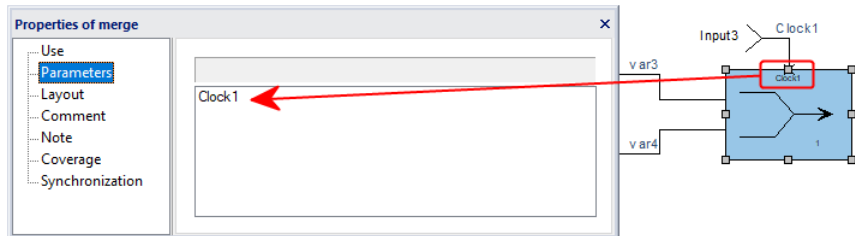


Figure 4.40: Parameters of Merge operators

Note

Merge operators accept the creation of additional inputs with the meaning:
`o=merge(clock;f1;f2;...;fn).`

To define Merge operators

- 1 Click **New Merge of Flows**  in **Time** toolbar and click in diagram.
- 2 Specify the clock defined to merge incoming flows set respectively with complementary clocks.
 - Connect an additional input to the Merge parameter entry on top.
 - Right-click the connecting wire and select *Properties*.

- Select **Define** on the left and set the local variable as follows: check **Named Variable Definition**, type the variable name in the **Local Variable** field or select it from the drop-down list.

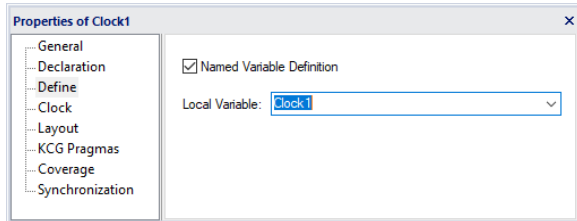


Figure 4.41: Defining local variable as Clock of Merge operator

- Select **Clock** on the left and set the local variable as **Clock**.

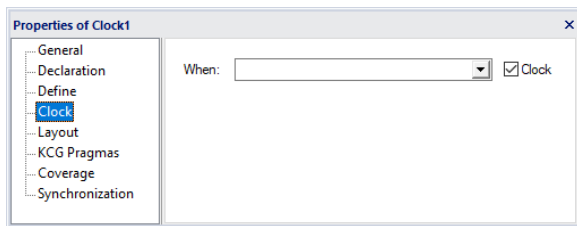


Figure 4.42: Setting local variable as clock

- 3 Set each incoming flow of the operator using local variables and complementary clocks:
 - Right-click the wire of an incoming flow and select *Properties*.
 - Select **Define** on the left and set a local variable as follows: check **Named Variable Definition**, type the variable name in the **Local Variable** field or select it from the drop-down list.

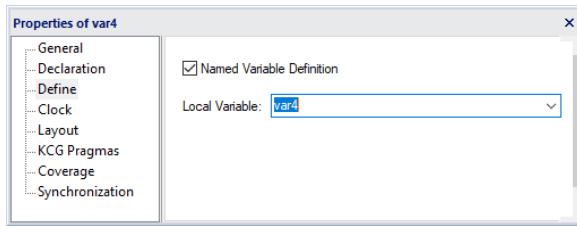


Figure 4.43: Defining local variables on incoming flows for Merge operator

Note

The local variable selected for each incoming flow must be different.

- Select **Clock** on the left and type the clock expression of the incoming flow in the **When** field or select it from the drop-down list.

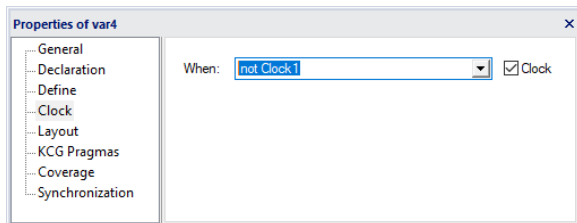


Figure 4.44: Defining clock expression of incoming flows for Merge operator

Designing with Structured Values

When designing models, you may need to define data structure operations on data types. In SCADE Suite, it is possible to create data structures and manipulate data in structure types like any other flow.

Note

A set of data structure constructs can be created with predefined primitives from the Structure/Array toolbar. The syntax and semantics of such primitives are detailed in [“Structure Expressions”](#) on page 115 of the *Scade Language Reference Manual*.

You can create the following data structure constructions:

- [“Setting and Designing Data Structure”](#)
- [“Setting and Designing Make or Flatten”](#)

Setting and Designing Data Structure

You can create **Data Structure** predefined operators to design a structure of data types from several flows.

Note

The syntax and semantics of such primitives are detailed in [“Structure Expressions”](#) on page 115 of the *Scade Language Reference Manual*.

To define a Data Structure

- 1 Click **New Data Structure**  in **Structure/Array** toolbar and click in diagram.

2 Insert structure labels by adding inputs on the operator symbol.

Tip

You can set labels from the Parameter properties of the operator instance.

- Adding labels: click New and edit the label name
- Deleting labels: select the label and click Delete
- Editing labels: select a name and type the new value

3 Connect the input and output of the operator to the design. Each label must be connected with an incoming flow of the corresponding type in the declared structure.

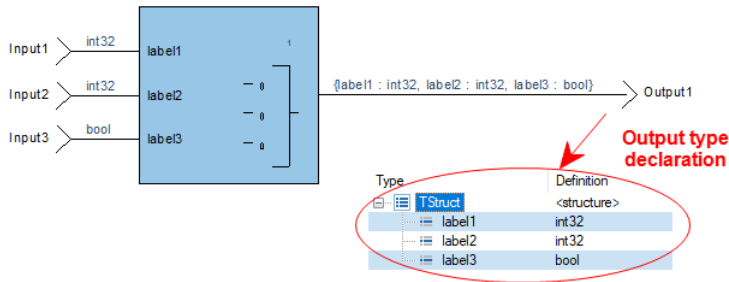


Figure 4.45: Example of Data Structure construction

Setting and Designing Make or Flatten

You can create **Make** or **Flatten** predefined operators to respectively design output flow composition or input flow decomposition with array (only Flatten) and structure data types. **Make** operators are instantiated as composition calls and **Flatten** operators as decomposition calls.

Basic design procedure:

- Declaring structure or array types necessary for composition/ decomposition.
- Creating operator calls in diagrams either by drag and drop of declared types, or contextually from typed outputs/inputs.

- Connecting properly typed input/output flows.
-

Note

If you ever create empty Make or Flatten operator instances from the Structure/Array toolbar, you must set the corresponding type in Parameters properties of the operator instance to automatically design the symbol with the appropriate number of inputs/outputs.

Read the following detailed instructions:

- [“To define Make operator \(composition\) calls”](#)
- [“To define Flatten operator \(decomposition\) calls”](#)

To define Make operator (composition) calls

- 1 Declare the structure type to be used for composition or identify from the **Scade View** a structure type declared in the project.
-

Note

For instructions, see [“Setting Structure Types”](#) on page 212.

- 2 Hold CTRL-key down, select the type, then drag and drop it from **Types View** in a **Diagram View**.
-

Tip

Design composition calls directly from output flows in diagrams: right-click the previously typed (structure) output instance in the diagram and select *Compose*. For details, see [“Composing/Decomposing Structured Types”](#).

- 3 Connect the input flow(s) and output flow(s) to the operator.
-

Note

Each input must have a type matching the type of the labels in the structure. You cannot connect the hidden parameter that represents the associated type.

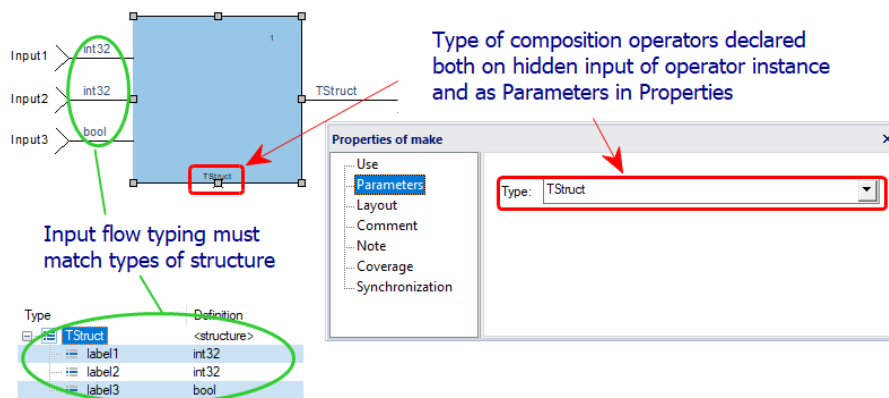


Figure 4.46: Designing Make operator instances

Tip

Edit types either directly on symbol (click in editing zone above hidden input mark) or via the Parameter properties of the operator instance.

To define Flatten operator (decomposition) calls

- 1 Declare the type (structure or array) to be used for decomposition or identify from **Scade View** a structure/array type declared in the project. For instructions, see [“Setting Array Types”](#) on page 209 or [“Setting Structure Types”](#) on page 212.
- 2 Select the type, then drag and drop it from **Types View** in a **Diagram View**.

Tip

Design decomposition calls directly from input flows in diagrams: right-click the previously typed (structure or array) input instance in the diagram and select *Decompose*. For details, see [“Composing/Decomposing Structured Types”](#).

3 Connect the input flow(s) and output flow(s) to the operator.

Note

Each output must have a type matching the type of the array. You cannot connect the hidden parameter that represents the associated type.

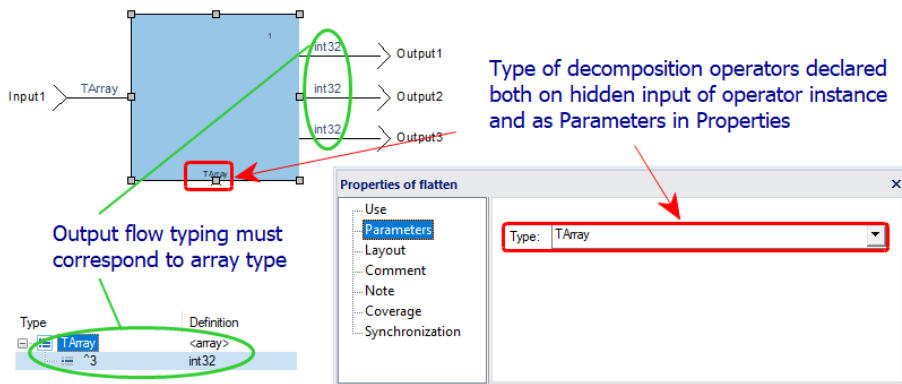


Figure 4.47: Designing Flatten operator instances

Tip

Edit types either directly on symbol (click in editing zone above hidden input mark) or via the Parameter properties of the operator instance.

Designing with Array Values

When designing model, you may need to define array operations on data types. In SCADE Suite, it is possible to create array operations and manipulate array data values like any other flow.

Note

A set of array constructs can be created with predefined primitives from the Structure/Array toolbar. The syntax and semantics of such primitives are detailed in [“Array Expressions”](#) on page 109 of the *Scade Language Reference Manual*.

You can create the following array constructions:

- [“Setting and Designing Arrays by Enumeration”](#)
- [“Setting and Designing Array Concatenation”](#)
- [“Setting and Designing Arrays by Repetition”](#)
- [“Setting and Designing Permutation of Array Values”](#)
- [“Setting and Designing Array Indexation Access”](#)
- [“Setting and Designing Arrays by Copy with Modification”](#)
- [“Setting and Designing Array Static Slice Access”](#)
- [“Setting and Designing Array Transposition”](#)

RELATED TOPICS

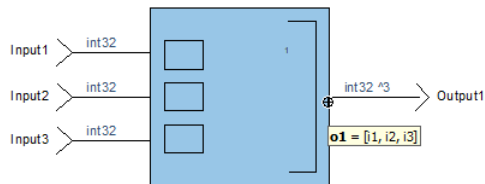
- [“Setting Array Types”](#) on page 209
- [“Setting Array/Structure Type Constants”](#) on page 220

Setting and Designing Arrays by Enumeration

You can create an array by enumerating its values. You use the **Data Array** predefined operator. This operator takes as many variables as inputs and produces a single array. The returned type is an array with sub-type equal to the type of the input (all inputs must have the same type) and with a size equal to the number of inputs.

To define arrays by value enumeration

- 1 Click **New Data Array**  in **Structure/Array** toolbar and click in diagram.
- 2 Connect the input(s) and the output of the operator to the design.



To create Data Array operator calls from array type

- 1 Hold Ctr L-key down.
- 2 Select an array type in **Scade View**.
- 3 Drag and drop the selected type in a **Diagram View**.

RELATED TOPICS

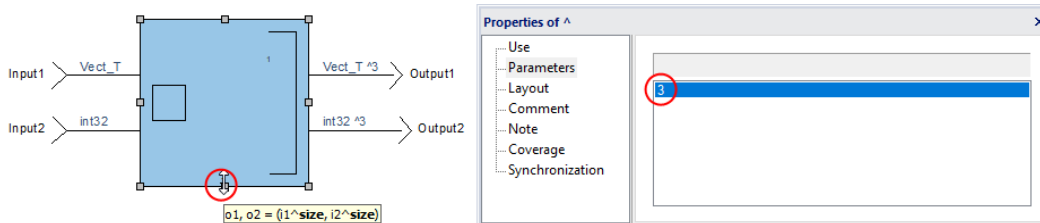
- [“Adding/Removing Predefined Operator Inputs”](#) on page 235

Setting and Designing Arrays by Repetition

You can create an array by value repetition using the **Scalar To Vector** predefined operator. This operator takes as input an expression (or several) representing the value(s) to repeat and as parameter a static expression indicating the number of repetitions. The returned type of each outgoing flow is an array with sub-type equal to the type of the input expression (it can be an array) and with a size equal to the number of repetitions.

To define arrays by value repetition

- 1 Click **New Scalar to Vector**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.
- 3 Select **Parameters** on the left and set required properties on the right.



Tips

- Press F2 to enter in editing mode or press Esc to exit editing mode.
- Specify the array size by typing directly on the symbol: select the symbol, click above the hidden input mark (editing area activated), and type the value.

- 4 Specify the number of repetitions (array size) to apply to all input values.
- 5 Connect the input and output of the operator to the design.

RELATED TOPICS

- [“Adding/Removing Predefined Operator Inputs”](#) on page 235
- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

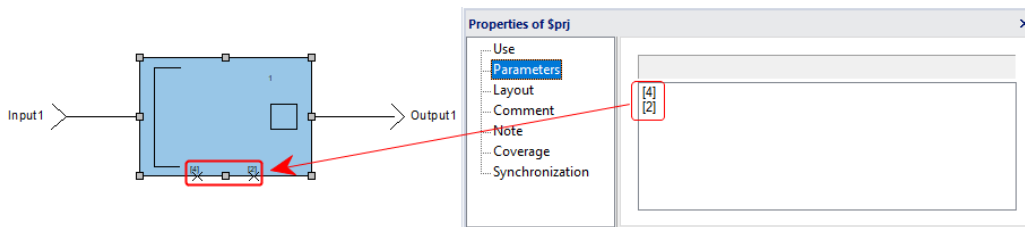
Setting and Designing Array Indexation Access

You can create access to an array by static or dynamic indexation. For these purposes, you use respectively the (static) **Projection** or **Dynamic Projection** predefined operators.

Static projection and *Dynamic projection* operators can take an array or a structure as single input. Both take as parameters any variables that represent the indexes or labels of the projection; dynamic projection has a default value as additional parameter. Both produce a single output. The returned type of dynamic projection operators is the same as the default parameter provided the default pin is connected; otherwise the type is computed from the array type and indexes projection.

To define array or structure access by static indexation

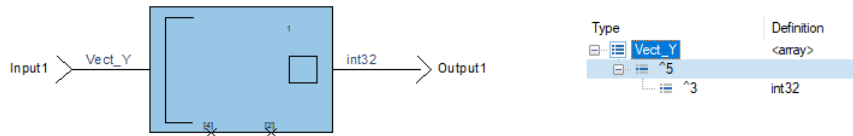
- 1 Click **New Projection**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.
- 3 Select **Parameters** on the left and set required properties on the right.



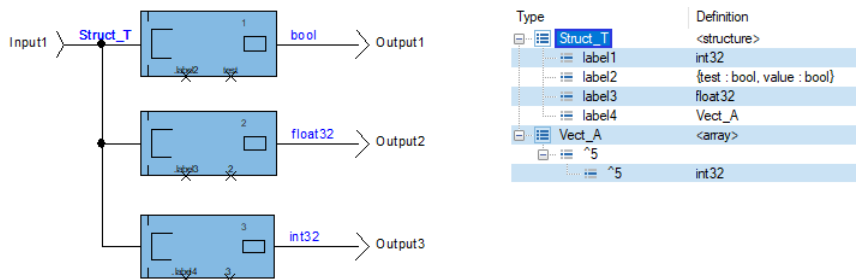
Tips

- Press F2 to enter in editing mode or press Esc to exit editing mode.
- Specify the array size by typing directly on the symbol: select the symbol, click above the hidden input mark (editing area activated), and type the value.

- 4 Add new hidden inputs by increasing **Number of inputs** from **Use** properties. New hidden inputs on projections means that each new input represents a new index addressing another dimension of the array or structure. Array indexes always start from 0.
- 5 Specify the index that identifies a value in the input type:
 - **For array types:** enter a static integer expression. Expressions are specified between brackets like [4] or [min + 2]. The first expression corresponds to the last dimension of the array. For instance in the next example, for a variable of type `int32^3^5`, the first expression [4] refers to dimension of size 5.



- **For structure types:** enter a label. Labels are specified with the ' . ' character in front like .label2.

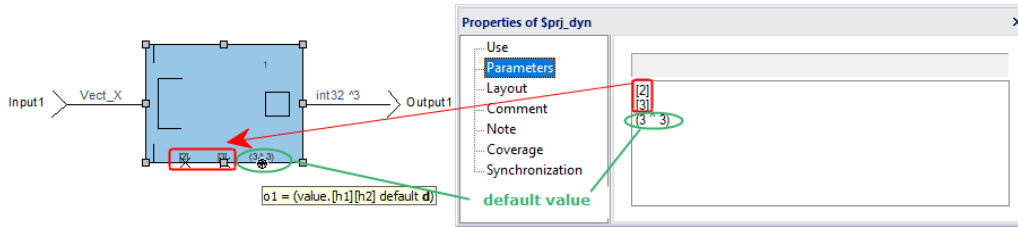


- 6 Connect the input and output of the operator to the design.

To define array access by dynamic indexation

- 1 Click **New Dynamic Projection**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.

3 Select **Parameters** on the left and set required properties on the right.



Tips

- Press F2 to enter in editing mode or press Esc to exit editing mode.
- Specify the array size by typing directly on the symbol: select the symbol, click above the hidden input mark (editing area activated), and type the value.

4 Add new hidden inputs by increasing **Number of inputs** from **Use** properties. New hidden inputs on projections means that each new input represents a new index addressing another dimension of the array or structure. Array indexes always start from 0.

5 Specify the index that identifies a value in the input type:

- **For array types:** enter an integer expression. Expressions are specified between brackets [] (default display). The first expression corresponds to the last dimension of the array. For instance, for a variable of type `int32^2^5`, the first index expression refers to dimension of size 5.
- **For structure types:** enter a label. Labels are specified with the ' . ' character in front like `.label2`.

6 Specify the default value to be used if the dynamic index is outside of array bounds. The last parameter in the list is the default value. Whenever substituting operators, any connection to the default parameter is lost.

7 Connect the input and output of the operator to the design.

RELATED TOPICS

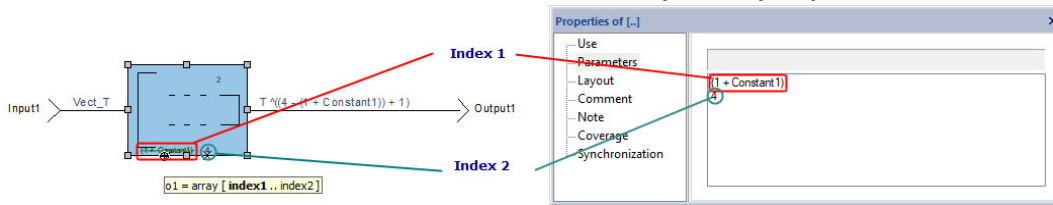
- [“Adding/Removing Predefined Operator Inputs”](#) on page 235
- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

Setting and Designing Array Static Slice Access

You can create access to arrays by static slice allowing to extract an array from a larger one. You use the **Slice** predefined operator. This operator takes as input an array and as parameters two static indexes representing the slice; it produces a single array. The returned type is an array with a size equal to $\text{index2} - \text{index1} + 1$. If no result can be computed, the type is unchanged.

To define arrays by static slice

- 1 Click **New Slice of an Array**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.
- 3 Select **Parameters** on the left and set required properties on the right.



Tip

Press F2 to enter in editing mode or press Esc to exit editing mode.

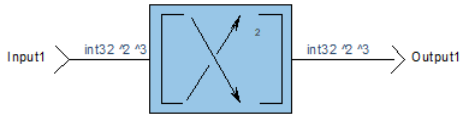
- 4 Specify the slice indexes to apply to the input array. Indexes must be defined as static expressions with a size constraint of: $\text{index1} \leq \text{index2}$. Adding new inputs on slice operator is impossible.
- 5 Connect the input and output of the operator to the design. When substituting operators, and depending on parameter compatibility between substituted operators (static vs. dynamic indexation), any connection to index parameters (of static expressions only) is maintained.

RELATED TOPICS

- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

Setting and Designing Permutation of Array Values

You can create permutation between array values of the last dimension. You use the **Reverse** predefined operator. This operator takes an array as single input and produces a single array. The returned type is the same as the type of the input array.

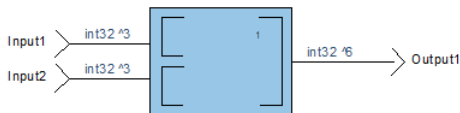


To define array permutation

- Click **New Reverse of an Array**  in **Structure/Array** toolbar, click in diagram, and connect the input/output of the operator to the design. Adding new inputs on reverse operator is impossible.

Setting and Designing Array Concatenation

You can create concatenation of arrays using the **Concatenation** predefined operator. This operator takes several arrays as input and produces a single array. The returned type is an array with sub-type equal to the sub-type of the first input and with a size equal to the sum of the sizes corresponding to the latest dimension of each input.



To define array concatenation

- Click **New Concatenation of Arrays**  in **Structure/Array** toolbar, click in diagram, and connect the inputs and output of the operator to the design.

RELATED TOPICS

- [“Adding/Removing Predefined Operator Inputs”](#) on page 235

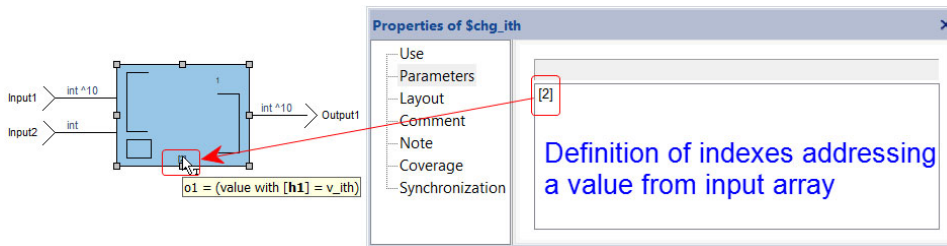
Setting and Designing Arrays by Copy with Modification

You can create an array from an existing array while specifying dynamic indexation for which they differ. You use the **Assign of a Structured Element** predefined operator.

This operator takes as first input an array or structure, as second input the new value, and as parameter a list that represents array indexes or structure labels; it produces an array or structure as single output.

To define array by copy with modification and dynamic indexation

- 1 Click **New Assign of a Structured Element**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.
- 3 Select **Parameters** on the left and set required properties on the right.



Tip

Press F2 to enter in editing mode or press Esc to exit editing mode.

- 4 Specify as parameter, the index/label identifying a value in the input type:
 - **For array types:** enter an integer or a static expression. Expressions are specified between brackets [] (default) like [3] or [min + 2].
 - **For structure types:** enter a label. Labels are specified with the ' . ' character in front like . labe l2.

- 5 Add new hidden inputs by increasing **Number of inputs** from **Use** properties. New hidden inputs on assign operators means that each new hidden input represents a new index addressing another dimension of the array. Array indexes always start from 0.
- 6 Connect the inputs and output of the operator. Whenever substituting operators, and depending on parameter compatibility between substituted operators (static vs. dynamic indexation, structure vs. array), any connection to index parameters (of static expressions only) is kept.

RELATED TOPICS

- [“Adding/Removing Predefined Operator Inputs”](#) on page 235
- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

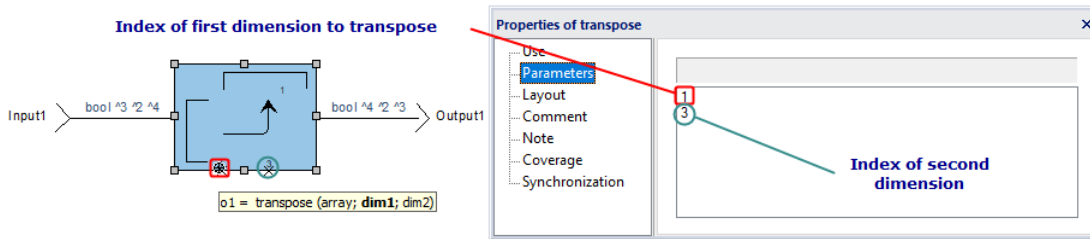
Setting and Designing Array Transposition

You can create the transposition of multidimensional arrays. This allows to swap two arbitrary dimensions of an array of array. You use the **Transpose** predefined operator. This operator takes as unique input an array and as parameters two static indexes that represent the dimensions to transpose; it produces a single array. The returned type is an array with permutation of both dimensions to transpose. It is computed only if the dimensions are valid.

To define transposition of array dimensions

- 1 Click **New Transpose of an Array**  in **Structure/Array** toolbar and click in diagram.
- 2 Right-click the operator in the **Diagram View** and select *Properties*.

- 3 Select **Parameters** on the left and set required properties on the right. Press F2 to enter in editing mode or press Esc to exit editing mode.



- 4 Specify the index of the dimensions to transpose. Dimension indexes must be defined by strictly positive integers to be valid. For instance, an input array is defined with size of $^3^2^4$, the resulting array after transposition of index 1 and 3 has size $^4^2^3$.
- 5 Connect the input and output of the operator to the design. Adding new inputs on transpose operator is impossible. Whenever substituting operators, parameters are lost if they are not valid indexes.

RELATED TOPICS

- [“Defining Value of Hidden Inputs on Operators”](#) on page 250

Designing with Array Iterators

You can iterate a node instantiation over array items following different patterns: map, fold, or a combination of both mapfold. These iterators are particular cases of higher-order patterns since one of their argument is an operator (user-defined or not).

Note

For an introduction to iteration patterns in Scade language, consult [“Iterators”](#) on page 79 in Chapter 5 / of *Scade Language Primer*. The syntax and semantics of such iteration pattern primitives are detailed in [“Operator Application and Higher-Order Patterns”](#) on page 119 of *Scade Language Reference Manual*.

You can use the following iteration pattern definitions:

- [“Definition for Map Iteration Pattern”](#)
- [“Definition for Mapi Iteration Pattern”](#)
- [“Definition for Mapw Iteration Pattern”](#)
- [“Definition for Mapwi Iteration Pattern”](#)
- [“Definition for Fold Iteration Pattern”](#)
- [“Definition for Foldi Iteration Pattern”](#)
- [“Definition for Foldw Iteration Pattern”](#)
- [“Definition for Foldwi Iteration Pattern”](#)
- [“Definition for MapFold Iteration Pattern”](#)
- [“Definition for MapFoldi Iteration Pattern”](#)
- [“Definition for MapFoldw Iteration Pattern”](#)
- [“Definition for MapFoldwi Iteration Pattern”](#)

Apply all these iteration patterns on operator calls within model diagrams:

- [“Creating and Setting Iteration Patterns”](#)

Definition for Map Iteration Pattern

You can create operators that apply a node point-wise to array items. You use the **Map**  predefined iteration pattern with the following characteristics.

Table 4.1: Characteristics of Map iterator

(map op <<n>>)	Inputs	Takes as many inputs as op . All are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	Produces as many outputs as op each with the type of the corresponding op output lifted to an array of size n .
Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	No constraint on input types.
	Outputs	No constraint on output types.
Warning: Respect the order of inputs/outputs as set in the operator's interface in Scade View .		

The following figures show examples of design using a **Map** iteration pattern.

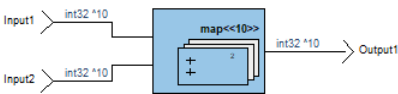


Figure 4.48: Map iterator example: point-wise sum of two arrays

The execution of the example above gives:

For
Input1 = [1,2,3,4,5,6,7,8,9,10]
Input2 = [0,1,2,3,4,5,6,7,8,9]
the result is Output1 = [1,3,5,7,9,11,13,15,17,19]

The next example shows how to use **Map** to multiply all the elements of two arrays by a scalar.

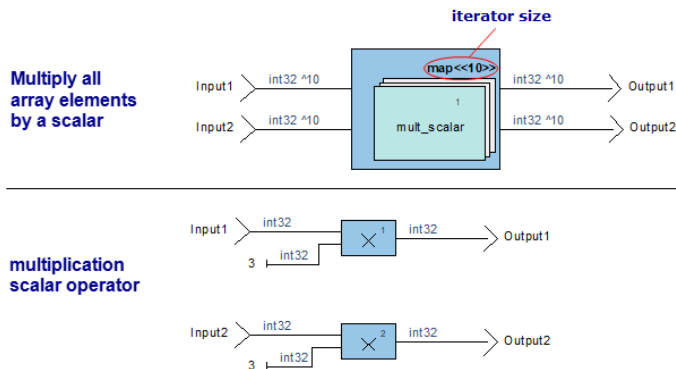


Figure 4.49: Map iterator example: multiplying all the arrays elements by a scalar

The execution of the example above gives:

For
Input1 = [1, 2, 3, 4, 5, 0, 0, 0, 0, 0]
Input2 = [6, 7, 8, 9, 10, 0, 0, 0, 0, 0]
the result is
Output1 = [3, 6, 9, 12, 15, 0, 0, 0, 0, 0]
Output2 = [18, 21, 24, 27, 30, 0, 0, 0, 0, 0]

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Mapi Iteration Pattern

You can create iterators with access to the index and facilitate the mapping between usual loop instructions and the map iteration pattern. Such iterators apply to all items of an array. You can use the index value of the iteration in the applied operator computation. You use the **Mapi** predefined iteration pattern with the following characteristics.

Table 4.2: Characteristics of Mapi iterator

(mapi op <<n>>)	Inputs	Takes as many inputs as op minus one. All are arrays of size n whose elements have the type of the corresponding op input <i>i.e.</i> , in order starting at op input 2.
	Outputs	Produces as many outputs as op each with the type of the corresponding op output lifted to an array of size n .
Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	First input must be of type integer to capture the iteration index.
	Outputs	No constraint on output types.

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

The following figures show examples of design using a **Mapi** iteration pattern.

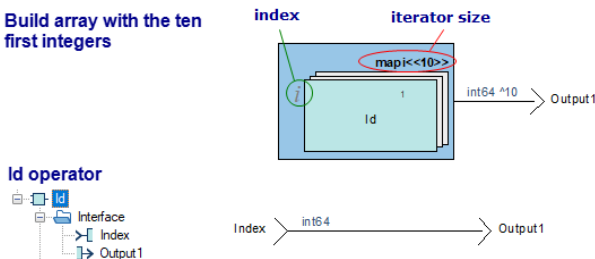
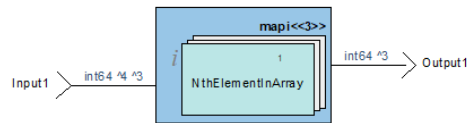


Figure 4.50: Mapi iterator example: building an array enumerating the ten first integers

The next example shows how to use **Mapi** to get elements from an array based on a formula using the index.

Iterator with index access
to array elements



NthElementInArray operator

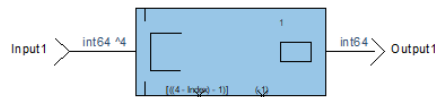
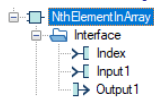


Figure 4.51: Mapi iterator example: index access to array elements

The execution of the example is:

For NthElementInArray operator

Index = 2

Input1 = [1, 2, 3, 4]

Output2 = 2

the result with mapi is

Input 1 = [1, 2, 3, 4] [4, 5, 6, 7] [8, 9, 10, 11]

Output1 = [4, 6, 9]

4 is computed when index = 1

6 is computed when index = 2

9 is computed when index = 3

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Mapw Iteration Pattern

You can create partial mapping iterators (bounded while-loops) that stop processing as soon as a condition is not fulfilled anymore (or the array is iterated completely).

Warning

Partial iteration is to be used carefully only in very specific cases of safety critical embedded applications. For details, consult [“Partial Iterators”](#) on page 84 in *Scade Language Primer*.

You use the **Mapw**  predefined iteration pattern with the following characteristics.

Table 4.3: Characteristics of Mapw iterator

(mapw op <<n>> if c default d1, d2, ...)	Iteration Pattern Inputs	The initial condition c that decides to enter the iteration or not. The default values d1 , d2 , . . . one per output.
	Resulting Operator Inputs	Takes as many inputs as op . All are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	Produces a first output corresponding to the index where iteration stopped (<i>i.e.</i> , exact number of iterations where op was activated). Produces as many outputs as op minus one, each with the type of the corresponding op output (started at output 2) lifted to an array of size n .
Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	No constraint on input types.
	Outputs	First output must be of type <code>bool</code> . In iterator, this output value stops iteration when false and continues iteration when true.

Warning: Respect the order of inputs/outputs as set in the operator's interface in **Scade View**.

The following figure shows how to use **Mapw** to add an element in a list (the list is an array where the end part is composed of non-significant values).

Add an element 3 at the end of a list (array with only a part of significant values)

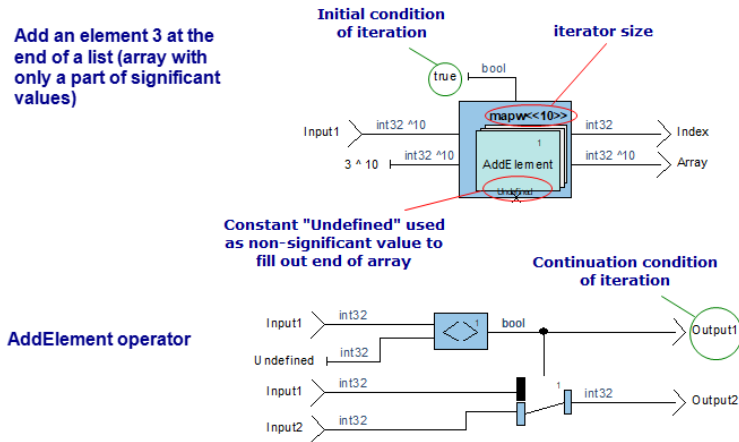


Figure 4.52: Mapw iterator example: adding an element at the end of a list

Important

If the initial iteration condition is false, the default value is given to the corresponding output array.

The execution of the above example gives:

For **Input1** = [0,1,2,-1,-1,-1,-1,-1,-1,-1]
 the result is **Array** = [0,1,2,3,-1,-1,-1,-1,-1,-1]
 and Undefined is set to -1

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Mapwi Iteration Pattern

You can create partial mapping iterators with access to the index of iteration.

Warning

Partial iteration is to be used carefully only in very specific cases of safety critical embedded applications. For details, consult [“Partial Iterators”](#) on page 84 in Chapter 5 / of *Scade Language Primer*.

You use the **Mapwi**  predefined iteration pattern with the following characteristics.

Table 4.4: Characteristics of Mapwi iterator

(mapwi op <<n>> if c default d1, d2, ...)	Iteration Pattern Inputs	The initial condition c that decides to enter the iteration or not. The default values d1 , d2 , . . . one per output.
	Resulting Operator Inputs	Takes as many inputs as op . All are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	Produces a first output corresponding to the index where iteration stopped (<i>i.e.</i> , exact number of iterations where op was activated). Produces as many outputs as op minus one, each with the type of the corresponding op output (starting at output 2) lifted to an array of size n .
Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	First input must be of type integer to capture the iteration index.
	Outputs	First output must be of type bool. In the iterator, this output value stops iteration when false and continues iteration when true.

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

The following figure shows how to use the mapwi iterator to fill an array with Input1’s first integers and keep on filling the end of the array with an undefined value.

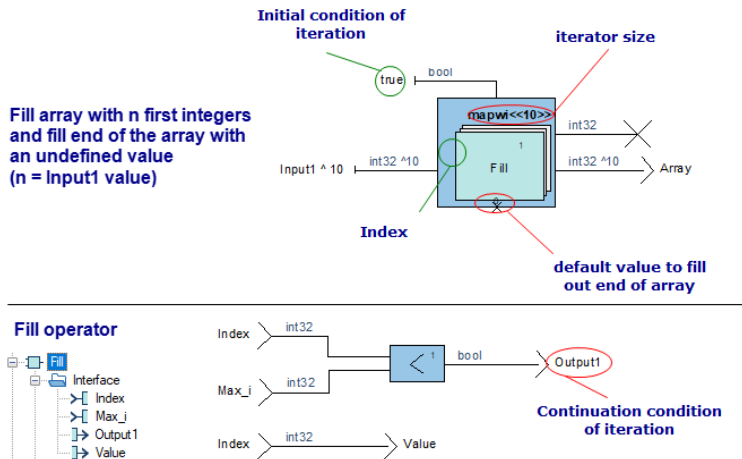


Figure 4.53: Mapwi iterator example: Filling the beginning of an array with selected values

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Fold Iteration Pattern

You can create operators that reduce arrays to an accumulator of any type, by applying a node successively to array elements. You use the **Fold** , predefined iteration pattern with the following characteristics.

Table 4.5: Characteristics of Fold iterator

(fold op <<n>>)	Inputs	Takes as many inputs as op . The first is of the type of the first op input and all the others are arrays of size n whose elements have the type of the corresponding op input.
	Output	Produces a single output with same type as op output.
The output accumulator is computed by applying successively op to the accumulator computed at the previous rank and to each element of the input arrays.		
op	Inputs	Takes at least one input. The first input is used to initialize the fold accumulator.
	Output	Has only one output of the same type as the first input.

Warning: Respect the order of inputs/outputs as set in the operator's interface in **Scade View**.

The following figures show examples of design using a **Fold** iteration pattern.

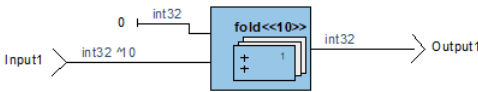


Figure 4.54: Fold iterator example: sum of the elements of an array

The execution of the example above gives:

For **Input1** = [0,1,2,3,4,5,6,7,8,9]
the result is **Output1** = 45

The next example shows how to use **Fold** to multiply all the elements of two arrays by a scalar.

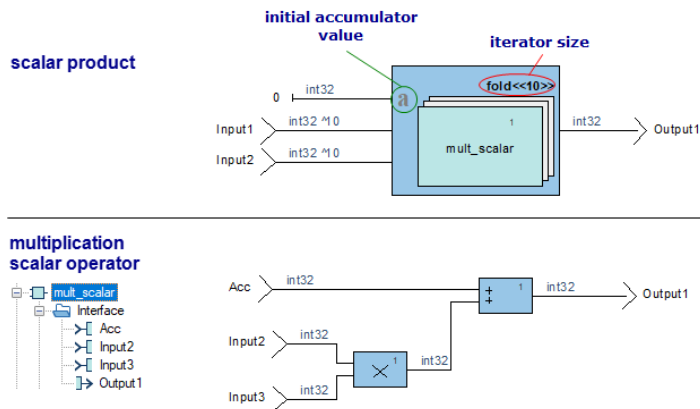


Figure 4.55: Fold iterator example: scalar product

The execution of the example above gives:

For

Input1 = [1,2,3,4,5,0,0,0,0,0]

Input2 = [2,3,4,5,6,0,0,0,0,0]

the result is **Output1** = 70

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Foldi Iteration Pattern

You can create iterators with access to the index and facilitate the mapping between usual loop instructions and the fold iteration pattern. Such iterators apply to all items of an array. You use the **Foldi** , predefined iteration pattern with the following characteristics.

Table 4.6: Characteristics of Foldi iterator

(foldi op <<n>>)	Inputs	Takes as many inputs as op minus one. The first is of the type of the second op input and all the others are arrays of size n whose elements have the type of the corresponding op input starting at rank 3.
	Output	Produces a single output with the same type as the second op output.
The output accumulator is computed by applying successively op to the accumulator computed at the previous rank and to each element of the input arrays.		
op	Inputs	Takes at least two inputs: the first one is of type integer to capture the iteration index; the second one is used to initialize the foldi accumulator.
	Output	Has only one output of the same type as the first input.
Warning: Respect the order of inputs/outputs as set in the operator's interface in Scade View .		

The following figures show examples of design using a **Foldi** iteration pattern.

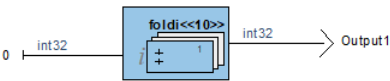


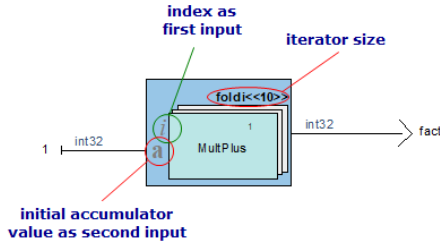
Figure 4.56: Foldi iterator example: sum of the ten first integers

The execution of the example above gives:

For Input1 = [0,1,2,3,4,5,6,7,8,9] the result is Output1 = 45

The next example shows how to use **Foldi** to compute the factorial of a static number.

Factorial of a static number



MultPlus operator

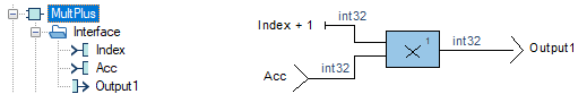
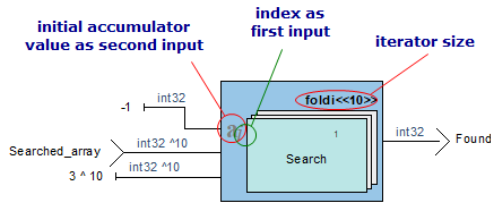


Figure 4.57: Foldi iterator example: factorial of a static number

The next example shows how to use **Foldi** to search an element in an array and return the index.

Search an element and return the index



Search operator

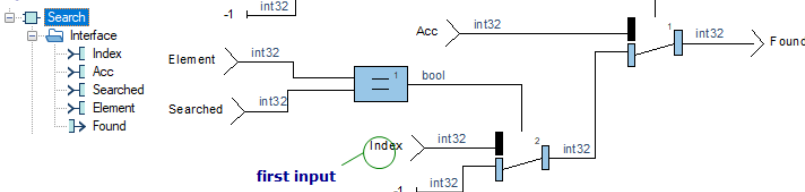


Figure 4.58: Foldi iterator example: searching an element in array and returning the index

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Foldw Iteration Pattern

You can create partial folding iterators that stop processing as soon as a condition is not fulfilled anymore (or the array is iterated completely). You use the **Foldw**  predefined iteration pattern with the following characteristics.

Warning

Partial iteration must be used carefully only in very specific cases of safety critical embedded applications. For details, consult [“Partial Iterators”](#) on page 84 in *Scade Language Primer*.

Table 4.7: Characteristics of Foldw iterator

(foldw op <<n>> if c)	Iteration Pattern Inputs	• The initial condition c that decides to enter the iteration or not.
	Resulting Operator Inputs	• Takes as many inputs as op. The first has the type of the first op input and all the others are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	• First output corresponding to the index where iteration stopped (<i>i.e.</i>, exact number of iterations where op was activated). • An output with the same type as op output.
Output accumulator is computed by applying successively op to the accumulator computed at the previous rank and to each element of the input arrays.		
op	Inputs	Takes at least one input. The first input is used to initialize the foldw accumulator.
	Outputs	• First output must be of type <code>bool</code>. In iterator, this output value stops iteration when false and continues iteration when true. • Second output of the same type as the first input.

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

The following figure shows how to use **Foldw** to search a value in an array.

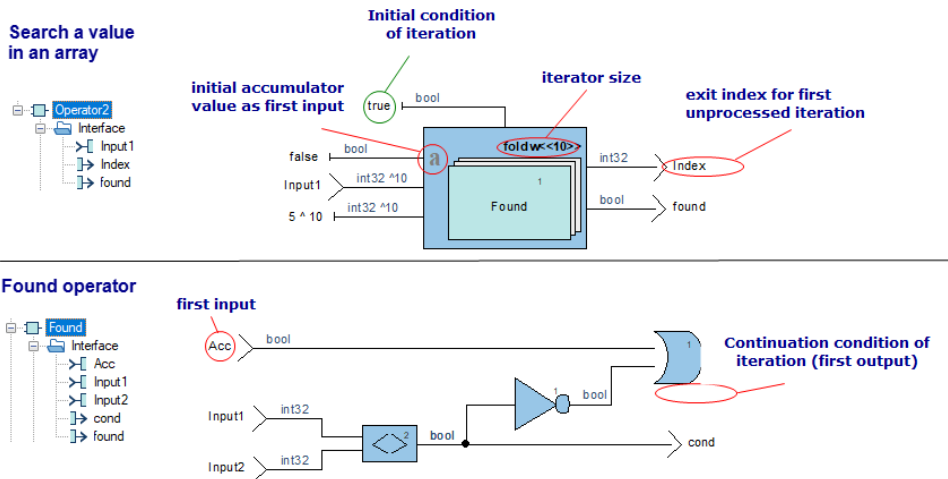


Figure 4.59: Foldw iterator example: searching a value in an array

Note

In this example, an array is filled with the constant 5 in order to respect the Scade syntax. However, SCADE Suite KCG identifies it and generates only the constant.

The execution of the example above gives:

```
For
Input1 = [1,2,3,4,5,0,0,0,0,0]
the result is
Output1 = 4
Found = true
```

Important

If the initial iteration condition is false or undefined, the operator iteration is not executed. In that case, the output accumulator value is its initial value.

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for Foldwi Iteration Pattern

You can create partial folding iterators with access to the index of iteration.

Warning

Partial iteration must be used carefully and only in very specific cases of safety critical embedded applications. For details, consult [“Partial Iterators”](#) on page 84 in *Scade Language Primer*.

You use the **Foldwi** , predefined iteration pattern with the following characteristics.

Table 4.8: Characteristics of Foldwi iterator

(foldwi op <<n>> if c)	Iteration Pattern Inputs	• The initial condition c that decides to enter the iteration or not.
	Resulting Operator Inputs	• Takes as many inputs as op minus one. The first is of the type of the second op input and all others are arrays of size n whose elements have the type of the corresponding op input starting at rank 3.
	Outputs	• First output, corresponding to the index where iteration stopped (<i>i.e.</i>, exact number of iterations where op was activated). • An output with the same type as the second op output.
Output accumulator is computed by applying successively op to the accumulator computed at the previous rank and to each element of the input arrays.		
op	Inputs	Takes at least two inputs: the first one is of type integer to capture the iteration index; the second one is used to initialize the foldwi accumulator.
	Outputs	• First output must be of type bool. In iterator, this output value stops iteration when false and continues iteration when true. • Second output of same type as the first input.

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

The following figure shows how to use **Foldwi** to know the length of a list.

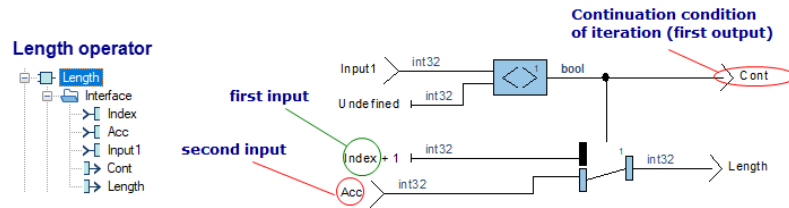
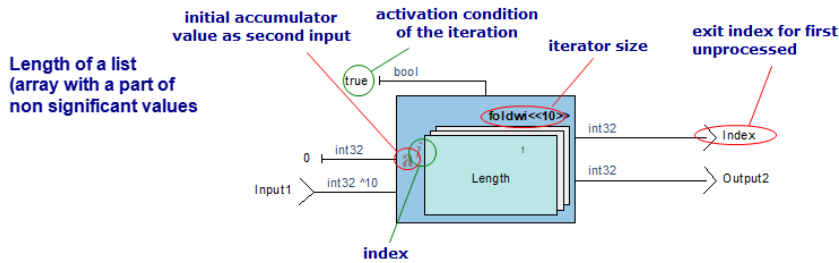


Figure 4.60: Foldwi iterator example: computing length of a list

Important

If the initial iteration condition is false or left undefined, the operator iteration is not executed. In that case, the value of the output accumulator is its initial value.

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for MapFold Iteration Pattern

You can use the **MapFold**  predefined iteration pattern to define operators combining the behavior of **map** and **fold** patterns. They can carry several accumulators and produce output vectors with respect to the following characteristics.

Table 4.9: Characteristics of MapFold iterator

(mapfold a op <<n>>)	Inputs	Takes as many inputs as op . The a first ones have the type of the a first op inputs and all the others are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	• a outputs (from rank 1 to a) in the same order and having the same type as the a outputs (from rank 1 to a) of the op operator. • next outputs (starting from rank a+1) are the same next outputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op output.
“ a ” is an immediate integer value that specifies the number of accumulators; it is optional, and its default is one. Output accumulators are computed by applying successively op to the accumulators computed at the previous rank, and to each element of the input arrays. Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	Takes at least a inputs.
	Outputs	Produces at least a outputs with the a first one having the same type as the a first inputs.
Warning: Respect the order of inputs/outputs as set in the operator's interface in Scade View .		

The following figures show examples of design using a **MapFold** iteration pattern.

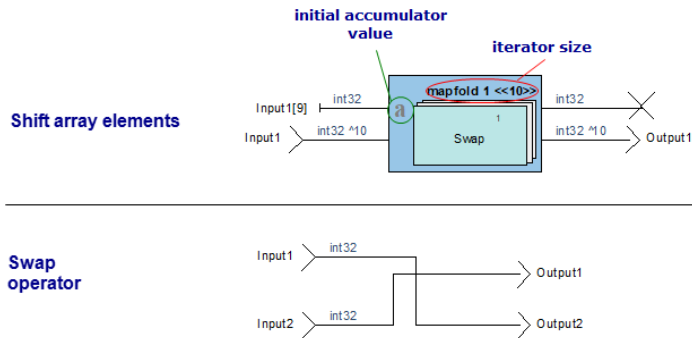


Figure 4.61: Mapfold iterator example: shifting array elements

The execution of the example above gives:

For **Input1** = [0,1,2,3,4,5,6,7,8,9]
the result is **Output1** = [9,0,1,2,3,4,5,6,7,8]

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for MapFoldi Iteration Pattern

You can use the **MapFoldi**  predefined iteration pattern to define operators combining the behavior of **map** and **foldi** patterns. They can carry several accumulators and produce output vectors with respect to the following characteristics.

Table 4.10: Characteristics of MapFoldi iterator

(mapfoldi a op <<n>>)	Inputs	Takes as many inputs as op minus one. The a first ones is of the type of the a consecutive op inputs starting at rank 2 and all the others are arrays of size n whose elements have the type of the corresponding op input.
	Outputs	• a outputs (from rank 1 to a) in the same order and having the same type as the a outputs (from rank 1 to a) of the op operator. • next outputs (starting from rank a+1) are the same next outputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op output.
“ a ” is an immediate integer value that specifies the number of accumulators; it is optional, and its default is one. Output accumulators are computed by applying successively op to the accumulators computed at the previous rank, and to each element of the input arrays. Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	Takes at least a+1 inputs. The first one is of type integer to capture the iteration index.
	Outputs	Produces at least a outputs with the a first one having the same type as inputs 2 to a+1 .

Warning: Respect the order of inputs/outputs as set in the operator's interface in **Scade View**.

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for MapFoldw Iteration Pattern

You can use the **MapFoldw** predefined iteration pattern to define operators combining the behavior of **map** and **foldw** patterns. They can carry several accumulators and produce output vectors with respect to the following characteristics.

Table 4.11: Characteristics of MapFoldw iterator

(mapfoldw a op <<n>> if c default d1, d2, ...)	Iteration Pattern Inputs	• The initial condition c that decides to enter the iteration or not. • The default values d1, d2, . . . one per array outputs.
	Resulting Operator Inputs	• Takes as many inputs as op. • a first inputs in the same order and having the same type as the a first inputs of the op operator. • next inputs (starting from rank a+1) are the same next inputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op input.
	Resulting Operator Outputs	• An integer that returns the last index of iteration. • A Boolean giving the last value of the condition that continues iteration or not. • a outputs (from rank 3 to a+2) in the same order and having the same type as the a outputs (from rank 2 to a+1) of the op operator. • next outputs (starting from rank a+2) are the same next outputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op output.
“a” is an immediate integer value that specifies the number of accumulators; it is optional, and its default is one. Output accumulators are computed by applying successively op to the accumulators computed at the previous rank, and to each element of the input arrays. Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	Takes at least a inputs.
	Outputs	Produces at least a+1 outputs. The first one of type <code>bool</code> and the a following ones having the same type as the a first inputs.

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Definition for MapFoldwi Iteration Pattern

You can use the **MapFoldwi** predefined iteration pattern to define operators combining the behavior of **map** and **foldwi**. They can carry several accumulators and produce output vectors with respect to the following characteristics.

Table 4.12: Characteristics of MapFoldwi iterator

(mapfoldwi a op <<n>> if c default d1, d2, ...)	Iteration Pattern Inputs	• The initial condition c that decides to enter the iteration or not. • The default values d1, d2, . . . one per array outputs.
	Resulting Operator Inputs	• Takes as many inputs as op minus one. • a first inputs in the same order and having the same type as the a first inputs of the op operator. • next inputs (starting from rank a+1) are the same next inputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op input.
	Resulting Operator Outputs	• An integer that returns the last index of iteration. • A Boolean giving the last value of the condition that continues iteration or not. • a outputs (from rank 3 to a+2) in the same order and having the same type as the a outputs (from rank 2 to a+1) of the op operator. • next outputs (starting from rank a+3) are the same next outputs of the op operator and their types are arrays of size n whose elements have the type of the corresponding op output.
“a” is an immediate integer value that specifies the number of accumulators; it is optional, and its default is one. Output accumulators are computed by applying successively op to the accumulators computed at the previous rank, and to each element of the input arrays. Output arrays are computed by applying op point-wise to each element of the input arrays.		
op	Inputs	Takes at least a+1 inputs. The first one is of type integer to capture the iteration index.
	Outputs	Produces at least a+1 outputs. The first one of type <code>bool</code> and the a following ones having the same type as inputs 2 to a+1 .

Warning: Respect the order of inputs/outputs as set in the operator’s interface in **Scade View**.

For pattern creation, see [“Creating and Setting Iteration Patterns”](#) on page 400.

Creating and Setting Iteration Patterns

To design array iterations in models, there are different methods available for applying any of the available iteration patterns on user-defined and predefined operator calls:

- Inserting operator in diagrams and applying iteration pattern on operator.
- Inserting iteration pattern in diagrams and applying operator on pattern.
- Modifying operator to apply iteration pattern by contextual menu.

You can use the **Higher Order** toolbar to apply iteration patterns.

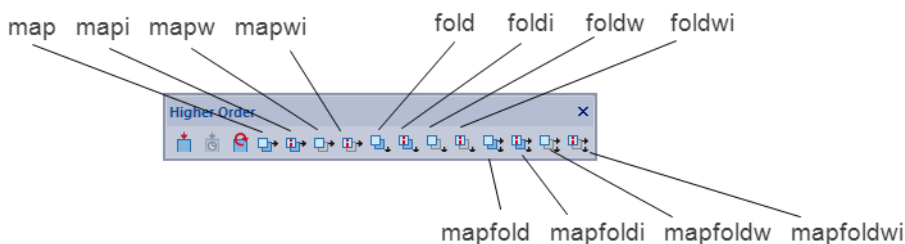


Figure 4.62: Higher Order toolbar with all iteration patterns

To use any iteration pattern in diagrams

- 1 Click the pattern button in **Higher Order** toolbar and apply the iteration pattern to an operator in one of the following ways:
 - Click on any operator instance in diagram to apply the selected pattern.
 - Click in diagram to create an iterator instance first: for *user-defined operators*, select the operator in **Scade View** and drop it on the created iterator instance; for *predefined operators*, click the corresponding button in toolbars and again on the created iterator instance.Alternatively, it is possible to:
 - Right-click any user-defined or predefined operator instance in diagrams, select *Apply Higher Order*, and choose the iteration pattern to apply from the drop list.

- Modify user-defined or predefined operator instance properties as explained in [“Editing Use Properties of Operator Calls”](#) on page 448.
- 2 Display the *Properties* of the iterated operator in the diagram and select **Use** on the left to set required properties on the right.

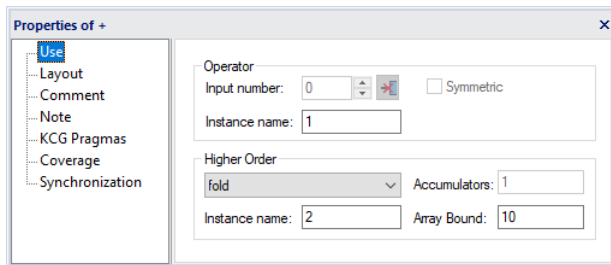


Figure 4.63: Defining parameter size of iterators

- 3 Specify the iterator size in **Array Bound**.

Tip

You can set the iterator size directly from the iterator symbols in diagrams. Click twice on the iterator <<size>> parameter and type the size.

- 4 For some iterators (mapfold, mapfoldi, mapfoldw, and mapfoldwi), specify the number of accumulators in **Accumulators**.
- 5 Connect the inputs and outputs of the operator to the design and make sure you correctly set the data flow types.

Designing with Operator Activation

You can condition the activation of a node instance by an arbitrary Boolean expression. This is done through various activation patterns which are particular cases of higher-order patterns since one of their argument is an operator (user-defined or not).

There are different methods available for creating activation patterns:

- Inserting operator in diagrams, then substituting it by activation pattern.
- Inserting higher-order pattern in diagrams, then substituting by operator.
- Modifying operator to apply activation pattern by contextual menu.
- Modifying operator properties to apply activation pattern as explained in [“Editing Use Properties of Operator Calls”](#) on page 448.

You can set the following activation patterns on operator calls:

- [“Setting and Designing Boolean Activation”](#)
- [“Setting and Designing Clocked Activation”](#)

RELATED TOPICS

- [“Creating and Designing If Blocks”](#) on page 180

Setting and Designing Boolean Activation

You can create node activation by connecting the outputs to a default flow and maintaining a value if the activation condition is false, or by memorizing the outputs to maintain their values when the node is not active. You use the **Boolean Activate** predefined pattern.

Note

This pattern takes as first input the condition expression, as many inputs as the called operator, and as parameter the default values; it produces as many outputs as the called operator. The returned type is a built-in type computed from the called operator outputs.

To define Boolean activation of nodes

- 1 Click **New Boolean Activate**  in **Higher Order** toolbar and create the activation operator in one of the following ways:
 - Click again on the existing operator instance in **Diagram Views** to apply the selected pattern.
 - Click again in **Diagram Views** to create an activation instance. For user-defined operators, select the operator and drop it on the created instance. For predefined operators, click the corresponding button in toolbars and click again on the created instance.

Tip

Right-click an existing operator instance in Diagram Views and select *Apply Higher Order > Boolean Activate*.

- 2 Right-click the instance in the **Diagram View** and select *Properties*.
- 3 Select **Use** on the left and set required properties on the right.

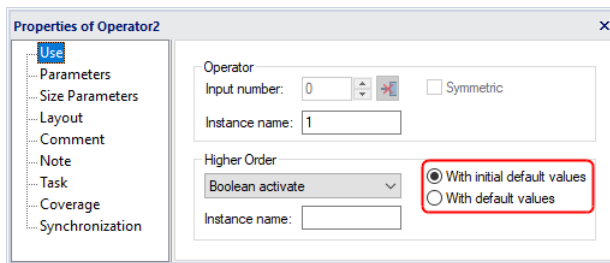


Figure 4.64: Tuning activation mechanism: default init values or default values

- 4 Set the radio buttons as appropriate:
 - **With initial default values** to produce the default values only when the condition is false at the first cycle. In this case, the operator introduces a memory.

- **With default values** to produce the default values when the condition of activation is false.

Note

For an introduction to these activation mechanisms in Scade language, consult respectively [“Node Activation with Output Memorization”](#) on page 14 and [“Node Activation with Default Flows”](#) on page 15 in Chapter 2 / of *Scade Language Primer*.

- 5 Define the default values either way:
 - Display **Parameters** options and enter each value on corresponding line.
 - Connect the appropriate inputs that define these values.
- 6 Define the condition of activation with any Boolean expression and connect it to the condition input (on top of activate symbols).

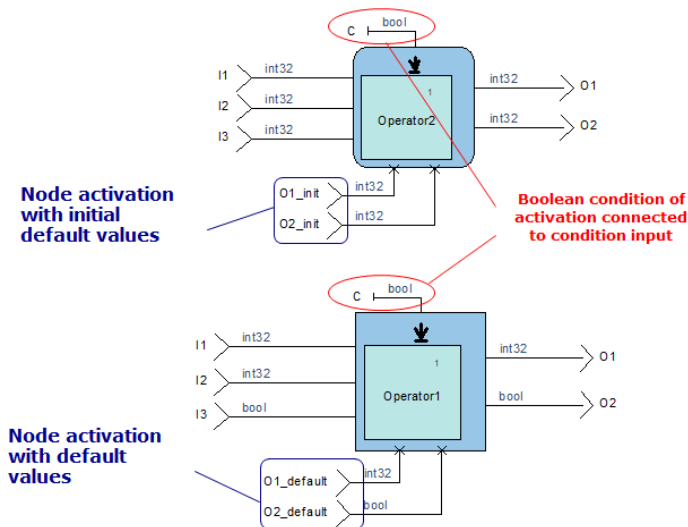


Figure 4.65: Examples of Boolean activation patterns

Setting and Designing Clocked Activation

You can create node activation by constraining the availability of outputs to the expression of a clock condition. This pattern is equivalent to when constructions on node inputs. You use the **Clocked Activate** predefined pattern.

Note

By default, the Clocked Activate creation is disabled. To be able to design such language construct, go to *Tool > Options* menu, in Language options expand the Higher order list, and check Clocked Activate.

To define clocked activation of nodes

- 1 Click **New Clocked Activate**  in **Higher Order** toolbar and create the activation operator in one of the following ways:
 - Click again on the existing operator instance in **Diagram Views** to apply the selected pattern.
 - Click again in **Diagram Views** to create an activation instance. For user-defined operators, select the operator and drop it on the created instance. For predefined operators, click the corresponding button in toolbars and click again on the created instance.
-

Tip

Right-click an existing operator instance in Diagram Views and select *Apply Higher Order > Clocked Activate*.

- 2 Right-click the instance in the **Diagram View** and select *Properties*.
- 3 Select **Use** on the left and set required properties on the right.

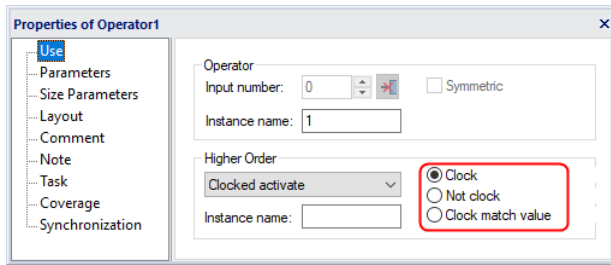


Figure 4.66: Tuning clocked activation mechanism with clock expressions

- 4 Set the clock expression with the radio buttons as appropriate:
 - **Clock** to execute the node when the specified clock is true.
 - **Not clock** to execute the node when the specified clock is false.
 - **Clock match value** to execute the node when the clock matches a specified value.

Note

For an introduction to this activation mechanism in Scade language, consult [“Simple Node Activation”](#) on page 16 in Chapter 2 / of *Scade Language Primer*.

- 5 Select **Parameters** on the left and set required properties on the right.

Tip

Press F2 to enter in editing mode or press Esc to exit editing mode.



Figure 4.67: Setting clock value for operator clocked activation

- 6 In **Clock**, specify the clocked variable that assigns the value of the incoming flow to the outgoing flow.

Note

You can also specify clocked variables by connecting the appropriate variables to the hidden inputs that represent the clock expression (on top of symbols).

- 7 In **Pattern**, specify the value with which the clocked variable has to match with (enabled only when the clock match value expression is set). The value must be a Boolean or enumerated value.

Note

The top hidden input corresponding to the pattern of the match expression cannot be connected to.

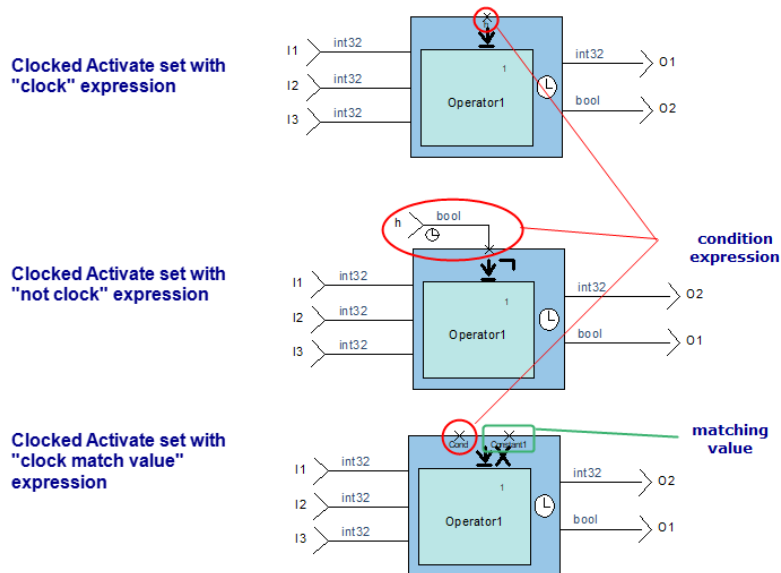


Figure 4.68: Examples of clock activation patterns

Designing with Restart Instantiation

You can enable any predefined or user-defined node instances to reset automatically on specific condition: you use the **Restart** predefined pattern. Restarted nodes reset their behavior to the first cycle. All node instances present in the definition of restarted nodes are reset automatically. Notice that the flow initialization of memory constructs (Init, Pre, Fby, or State machines) defined in restarted nodes is impacted. For an introduction to restart pattern in Scade language, consult [“Operator Instantiation”](#) on page 9 in Chapter 2 / of *Scade Language Primer*.

To define resettable instances

- 1 Click **New Restart**  in **Higher Order** toolbar and create the operator instance in one of the following ways:
 - Click again on the existing operator instance in **Diagram Views** to apply the selected pattern.
 - Click again in **Diagram Views** to create a restart instance. For user-defined operators, select the operator to define as resettable and drop it on the created restart. For predefined operators, click the corresponding button in toolbars and click again on the created restart. Alternatively, right-click an existing operator instance in **Diagram Views** and select *Apply Higher Order > Restart*.
- 2 Define the condition of activation with any Boolean expression and connect it to the condition input (on top of restart symbol).

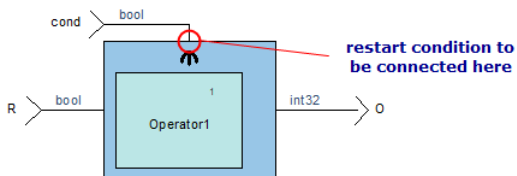


Figure 4.69: Example of restart activation pattern

5 / Navigating in Models

This chapter covers all the features that facilitate navigation between various model representations and search into the model design.

- [“Finding and Locating Design Elements”](#)
- [“Searching for Design Dependencies”](#)
- [“Navigating in Model Content”](#)
- [“Filtering Model Content”](#)

Finding and Locating Design Elements

The search feature allows you to locate specified design elements in models. In addition to standard search options, you can set the type of objects or the scope of the search. For more advanced search in model data, see [“Searching for Design Dependencies”](#) on page 412.

Tip

For quick search facilities, use the Find toolbar.

Table 5.1: Commands of the Find toolbar

Command	Icons	Description
Pick Text to Find		Copies the text of the element selected in views as the text to search for in the drop-down list
Find All		Launches the text search in the current project. Search results are displayed on the Find output tab in the Output Window.

To find and locate any design element

1 Select *Edit > Find* (Ctrl+F) to open the **Find** window.

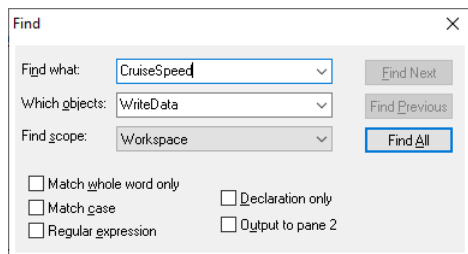


Figure 5.1: Specifying design element in Find window

- 2 Enter the name of an element or the name of a predefined operator (according to language keywords available from [“Keyword List”](#) in *Scade Language Reference Manual*) in the **Find what** field or select previously entered name from drop-down list.
- 3 Specify the type of objects from the **Which objects** drop-down list.
- 4 Specify the scope of the search from the **Find scope** drop-down list.
 - **Workspace**: search applies to all projects (ETP files) present in SCADE Suite workspace.
 - **Context**: search applies to the context of the active view or to the context of the selection in graphical diagrams or in views.
 - **<ProjectName>**: search applies to a specific project (ETP file) present in SCADE Suite workspace.
- 5 Select options by checking the following checkboxes as required:
 - **Match whole word only**: to restrict the search to the name you entered.
 - **Match case**: to match the search element with the letter case you typed.
 - **Regular expression**: to allow expressions in the entered search strings.
 - **Declaration only**: to exclude all element instances from the search and locate only element declarations.

- **Output to pane 2:** to display search results in a second tab and keep the content of a previous search in the first tab.
- 6 Click **Find All**. The **Find** window closes and search results are listed on **Find** output tab. If no results, the window remains open allowing new search.

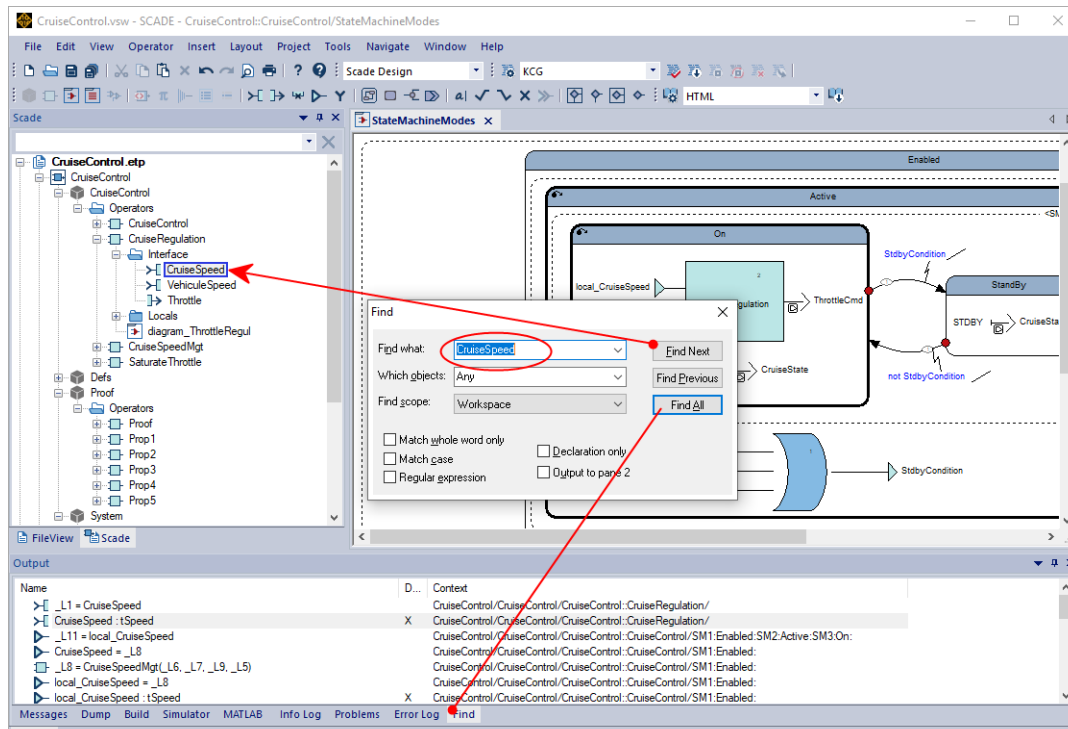


Figure 5.2: Search results displayed on Find output tab

- 7 Right-click an item on **Find** output tab and select **Locate** to highlight the item in **Scade View** and open its corresponding view (if any).

Double clicking items on the **Find** output tab has the same effect as *Locate*. The **Next Item from output tab**  (F4) or **Previous Item from output tab**  (Shift+F4) in **Go To** toolbar allow to switch to the next item while locating it.

Searching for Design Dependencies

In addition to standard search, it is possible to search dependencies between design elements defined in the models. You can look for specific elements as described in [“Finding and Locating Design Elements”](#) on page 409. For advanced search purposes, you browse through models with the following features:

- [“Searching for Design Element Definitions”](#)
- [“Searching for Design Element References”](#)
- [“Searching for Producers or Deep Producers”](#)
- [“Searching for Consumers or Deep Consumers”](#)
- [“Searching for Operator Calls”](#)
- [“Searching for Operator Callers”](#)

All elements found via the browsing search feature are displayed on the **Browse** output tab in the Output Window.

Searching for Design Element Definitions

You can search for the definition of any specified design element. The selection is possible from any view or tab that displays design elements.

To search for design element definition

- 1 Select a design element in SCADE Suite model.
- 2 Select *Navigate > Browse > Definitions* or click  in **Browse** toolbar.
- 3 Right-click an item on **Browse** output tab and select *Locate* to see its definition in the corresponding view.

Searching for Design Element References

You can search for the reference(s) of any specified design element. The selection is possible from any view that displays design elements.

To search for design element references

- 1 Select a design element in SCADE Suite model and click  in **Browse** toolbar, or right-click the element in diagrams and select *Browse References* in contextual menu.
- 2 Right-click an item in **Browse** output tab and select *Locate* to see its reference in the corresponding view.

Searching for Producers or Deep Producers

You can search for variable production in models from two dependency perspectives.

- Searching for *variables that directly produce another variable* within the same operator (*i.e.*, all variable whose values produce the selected one). In this case, you look for the producers of a given variable.
- Searching for *variables that produce another variable within the same operator or outside* (*i.e.*, across operator calls). In this case, look for deep producers of any variable.

To search for producers

- 1 Select a variable, input or output in the design and use either:
 - *Navigate > Browse > Producers* or click  in **Browse** toolbar to look for the direct producer dependency of the selected variable.
 - *Navigate > Browse > Deep Producers* or click  in **Browse** toolbar to look deeper for the dependency inside all operator calls.

- 2 Right-click an item in **Browse** output tab and select *Locate* to see the element in the corresponding view.
- 3 Click again on one of the buttons to keep following the dependencies. Repeat from step 2 to search for dependent consumers.

Note

The search commands for producers and deep producers are not recursive. They cannot look for indirect producers possibly obtained by transitivity. To search for different (deep) producers one after another, select a variable in the Browse output tab and reapply the search command, and so on.

Example: In the Cruise Control model distributed in SCADE installation, consider the ThrottleCmd output of the CruiseControl operator:

- The producer of ThrottleCmd is the variable (`_L3`) that produces this output; the producers of `_L3` are `_L11` and `_L8` (as inputs of the calling operator); and the producer of `_L8` is Speed which has in turn no producer within the same operator (Speed is an input of the node and the search remains within the node).
- The deep producer of ThrottleCmd is the variable (`_L3`) that produces this output; the deep producer of `_L3` is Throttle (corresponding output in the CruiseRegulation operator); and the deep producers of Speed are `_L13` in SystemSimul operator and `_L25` in Proof operator (both operators calling the CruiseControl operator).

Searching for Consumers or Deep Consumers

You can search for variable consumption in models from two dependency perspectives.

- Searching for *variables that directly consume another variable* within the same operator (*i.e.*, all variable whose values depend on the selected one). In this case, you look for the consumers of a given variable.

- Searching for *variables that consume another variable within the same operator or outside (i.e., across operator calls)*. In this case, you look for deep consumers of any variable.

To search for consumers

- 1 Select a constant, variable, input/output in the design and use either:
 - *Navigate > Browse > Consumers* or click  in **Browse** toolbar to look for the direct consumer dependency of the selected variable.
 - *Navigate > Browse > Deep Consumers* or click  in **Browse** toolbar to look deeper for the dependency inside all operator calls.
- 2 Right-click an item on **Browse** output tab and select *Locate* to see the element in the corresponding view.
- 3 Click again on one of the buttons to keep following the dependencies. Repeat from step 3 to search for dependent consumers.

Note

Search commands for consumers and deep consumers are not recursive. They cannot look for indirect consumers possibly obtained by transitivity. To search for different (deep) consumers one after another, select a variable in the Browse output tab and reapply the search command, and so on.

Example: In the Cruise Control model distributed in SCADE installation, consider the Speed input of the CruiseControl operator:

- The consumers of Speed are all variables (_L8, _L5, _L1, _L4) that use this input; the consumer of _L8 is _L3 (as output of the called operator); and the consumer of _L3 is ThrottleCmd which has in turn no consumer within the same operator (ThrottleCmd is an output of the node and the search remains within the node).

- The deep consumers of Speed are all variables (_L8, _L5, _L1, _L4) that use this input; the deep consumer of _L8 is VehiculeSpeed (corresponding input in CruiseRegulation operator); and the deep consumers of ThrottleCmd are _L2 in SystemSimul operator and _L22 in Proof operator (both operators calling CruiseControl operator).

Searching for Operator Calls

You can search for all the operator calls contained inside a given operator.

To search for operator calls

- 1 Select an operator in SCADE Suite.
- 2 Select *Navigate > Browse > Calls* or click  in **Browse** toolbar to see the list of operators called in the selected operator.
- 3 Right-click an operator instance on **Browse** output tab and select *Locate* to display the corresponding view.

Searching for Operator Callers

You can search for all the operators which contain a call to a given operator.

To search for operator callers

- 1 Select an operator in SCADE Suite.
- 2 Select *Navigate > Browse > Callers* or click  in **Browse** toolbar to see the list of operators which call the selected one.
- 3 Right-click an operator instance on **Browse** output tab and select *Locate* to display the corresponding view.

Navigating in Model Content

While looking in SCADE Suite models for specific design elements or hierarchy information, you can easily navigate through displayed trees and views and quickly locate the owner of most design objects. For basic navigation purposes, you can navigate in models with the following actions:

- [“Navigating in Model and Package Hierarchy”](#)
- [“Navigating in Output Window”](#)
- [“Navigating in Views”](#)
- [“Navigating to Declaration from Diagrams”](#)
- [“Navigating to Type Definition”](#)
- [“Navigating between Opened Views and Files”](#)
- [“Navigating in HTML Documents”](#)

Navigating in Model and Package Hierarchy

It is possible to navigate between design elements located in trees and views with respect to model hierarchy. It varies with the element context in model and package hierarchy.

To navigate in hierarchy of design elements

- 1 Select element in **Scade View**, **Call Graph**, or [Model Editing Views](#) (except Textual).
- 2 Click the appropriate **Go To** button.

Up		Alt+Up arrow	Moves up to the owner of the object in the model
Down		Alt+Down arrow	Moves down to the first diagram, target state, branch in If/When Blocks, or fork transition
Next		Alt+Right arrow	Moves to next object of the same category on same owner level
Previous		Alt+Left arrow	Moves to previous object of the same category on same owner level

You move from selected elements as described in the following table.

Table 5.2: Navigation action possible from design elements

Element	Up ...	Down ...	Next/Previous ...
Package declaration	to package owner or model localization	n/a	to next/previous package in same package owner
Operator	to package owner or model localization	to operator diagram	to next/previous operator in same package owner while displaying diagram
Constant, type, sensor	to package owner or model localization	n/a	to next/previous item declaration in owner package (by name order)

Navigating in Output Window

While working on lists of design elements in views, you can move up/down between items of the same list and locate them simultaneously in views.

To move up/down between elements listed in Output Window

- Select an item from the list displayed on a tab. If selection focus is outside Output window tabs, navigation resumes from last selection focus in the active output list when applying commands. Use either:
 - *Navigate > Next Item From Output* (or press F4) or  in **Go To** toolbar to move down in the list of items and display instances in views.
 - *Navigate > Previous Item From Output* (or press Shift+F4) or  in **Go To** toolbar to move up in the list of items and display instances in views.

Navigating in Views

While working on design elements from model browsers, you can move up/down between items of the same tree level and switch the display of properties between design elements of the same category.

To move up/down between tree elements in Views

- Select an item from the tree displayed in views. Use either:
 - *Navigate > Next Item From Browser* (or press F8) or  in **Go To** toolbar to move down in items listed at the same tree level.
 - *Navigate > Previous Item From Browser* (or press Shift+F8) or  in **Go To** toolbar to move up in items listed at the same tree level.

Navigating to Declaration from Diagrams

While editing or simulating models, you can navigate to the declaration that corresponds to the design item selected in the active diagram.

To navigate to design element declaration

- 1 Select a design element in a **Diagram View** (or **Network Simulation View**).
- 2 Click **Show in Browser**  in **Navigate** toolbar to highlight in **Scade View** or **Instances View** the declaration of the selected item. Alternatively, right-click the element and select **Show in Browser** in contextual menu.

Navigating to Type Definition

While editing models, you can navigate to the declaration of types directly from design elements such as variable references in Diagram Views, any typed element declared with user types, and any Make or Flatten equation. Such navigation is possible from **Scade View**, **Diagram Views**, **Types Views**, and **Constants Views**.

To navigate to type definition of selected element

- Right-click a typed design element and select *Type Definition* in contextual menu to show the type in the corresponding **Types View**.

Navigating between Opened Views and Files

While editing models, you may open a lot of different views or files. Navigation between recently opened windows is easier and more efficient using the **Navigate** toolbar. Opening the same window several times is memorized for navigation. You cannot navigate through deleted elements or files (not part of navigation memory). Any file opened from *File > Open* is ignored and not memorized for navigation.

To switch between opened windows

- Click **Back** ← or **Forward** → to switch active views between recently opened windows or use the corresponding *Navigate > Back* or *Navigate > Forward* menus.

Navigating in HTML Documents

You can browse in HTML documents produced by SCADE Suite using the **Navigate** toolbar or standard HTML browsing features.

To navigate in HTML document pages

- 1 Use either of the following to move to previous page
 - Click **Back** ← in **Navigate** toolbar.
 - Right-click in the page and select *Back*.
- 2 Use either of the following to move to next page
 - Click **Forward** → in **Navigate** toolbar.
 - Right-click in the page and select *Forward*.

To use browsing features on opened HTML document

- From within opened HTML document, select either of the following menus:
 - *View > HTML Browser > Home* menu to go back to the first page
 - *View > HTML Browser > Refresh* menu to reload the document
 - *View > HTML Browser > Stop* menu to interrupt document loading

RELATED TOPICS

- Chapter 2 about [“Reporting about Suite Design”](#) in *SCADE LifeCycle User Manual*

Filtering Model Content

While working on large SCADE Suite models and looking for specific elements, files, or parts of design, you can easily filter out views content to focus on selected items.

To filter model content and set focus on

- Right-click any folder or any element displayed in [model browsers](#) and select *Focus on Selection*. Applicable to a multi-selection of folders or elements.

To remove model filtering in views

- Right-click anywhere in views and select *Remove Focus*.

To filter model content by text string

- Enter tree item name or any text string in text area on top of [model browsers](#). Filtering applies to any tree item containing typed text string. Reuse already entered text from the selection arrow list.

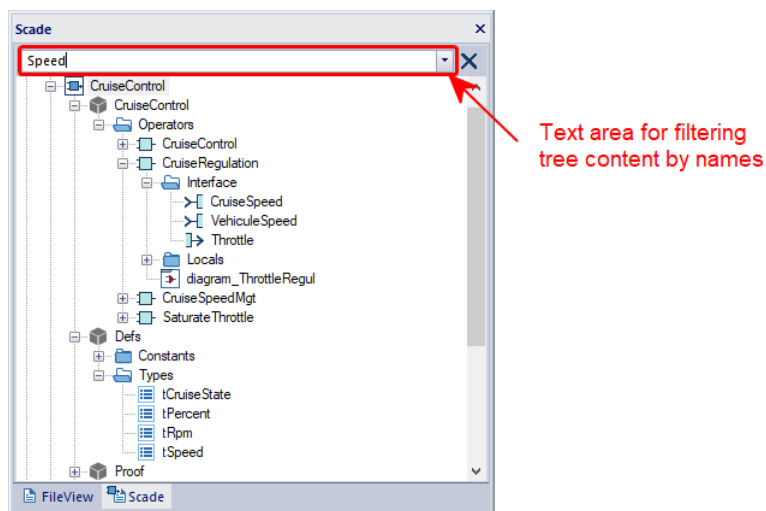


Figure 5.3: Searching for design element in Scade View

To remove model content filtering

- Click X button next to the text area.

6 / Editing Properties of Design Elements and Objects

This chapter identifies all design elements and objects whose properties can be modified with a set of attributes once declared in the models.

Note

A general description of the Properties window and how to use it is available at [“Properties Window”](#) on page 37.

- [“Editing Project Properties”](#)
- [“Editing File Folder Properties”](#)
- [“Editing Project Files Properties”](#)
- [“Editing Package Properties”](#)
- [“Editing Diagram Properties”](#)
- [“Editing Type Properties”](#)
- [“Editing Constant Properties”](#)
- [“Editing Sensor Properties”](#)
- [“Editing Operator Properties”](#)
- [“Editing Operator Call Properties”](#)
- [“Editing Input/Output Properties”](#)
- [“Editing Probe Properties”](#)
- [“Editing Local Variable Properties”](#)
- [“Editing Signal Properties”](#)
- [“Editing If Block and If Node Properties”](#)
- [“Editing When Block and When Branch Properties”](#)
- [“Editing State Machine Properties”](#)
- [“Editing State Properties”](#)
- [“Editing Transition Properties”](#)
- [“Editing Wire Properties”](#)
- [“Editing Assertion Properties”](#)
- [“Editing Textual Expression Properties”](#)
- [“Changing Design Element Layout”](#)
- [“Adding and Editing Comments”](#)
- [“Adding and Editing Notes”](#)
- [“Editing Instance Names”](#)
- [“Accessing Traceability Information”](#)

Editing Project Properties

The projects you create in SCADE Suite can be configured with a set of properties.

- [“To edit the general properties of a project”](#)
- [“To edit the description properties of a project”](#)
- [“To edit the document properties of a project”](#)
- [“To edit the SCADE properties of a project”](#)
- [“To edit the page format properties of a project”](#)
- [“To edit the layout properties of a project”](#)
- About simulation properties in projects, see [“Setting Project for Simulation Sessions”](#) on page 585.

To edit the general properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.
- 2 Select **General** and set properties on the right.

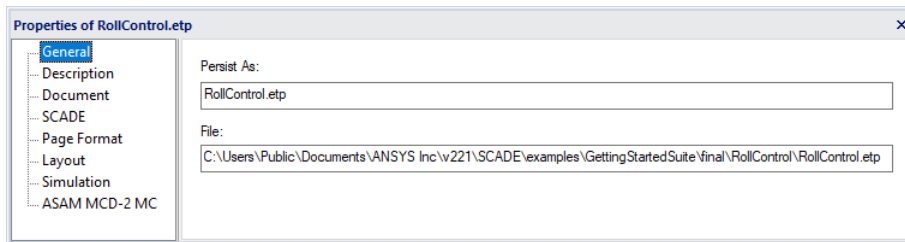


Figure 6.1: General properties of a project

- 3 Rename the project in **Persist As** if needed and save the new project file in the project folder location shown in **File** below.

Note

File is uneditable and shows the absolute path where the project is saved.

To edit the description properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.

- 2 Select **Description** and set properties on the right.

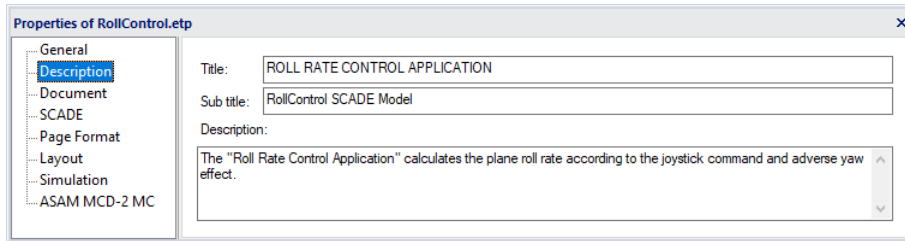


Figure 6.2: Description properties of a project

- 3 In **Title**, specify the title of the project.
- 4 In **Sub-title**, specify the sub-title of the project.
- 5 In **Description**, provide a description of the project.

To edit the document properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.
- 2 Select **Document** and set properties on the right.

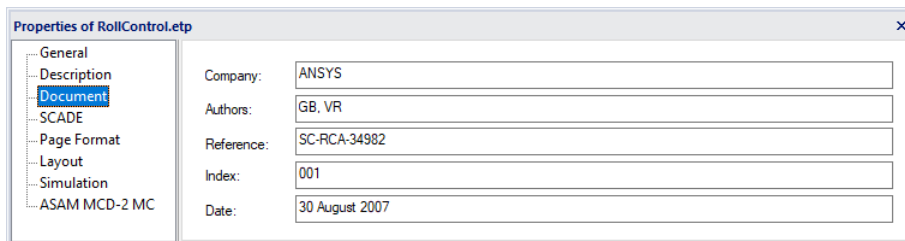


Figure 6.3: Document properties of a project

- 3 In **Company**, specify the name of one's own company.
- 4 In **Authors**, specify the name of project authors.
- 5 In **Reference**, provide any reference information about the project.

- 6 In **Index**, provide any index information about the project.
- 7 In **Date**, specify a project date.

To edit the SCADE properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.
- 2 Select **SCADE** and set properties on the right.

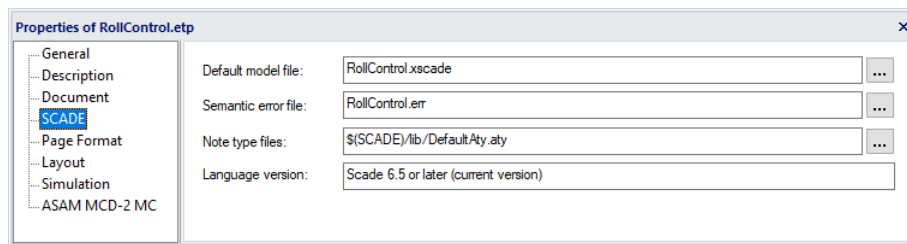


Figure 6.4: SCADE properties of a project

- 3 In **Default model file**, change the file generated by default to store any constants, types or sensors declared in models outside any package.

Note

The properties of constants, types, or sensors allow to change this default file. Operators and packages are created in their own file. For general information, see [“Multifile Management”](#) on page 66.

- 4 In **Semantic error file**, specify the path of the file where semantic check diagnosis are saved. This field may contain environment variables.
- 5 In **Note type files**, specify the path of the .aty file where annotation types are described. This field may contain environment variables. It is possible to specify several files to use all their descriptions in the model. For related usage, see [“Adding and Editing Notes”](#) on page 484.
- 6 **Language version** gives the Scade language version of the loaded project.

To edit the page format properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.
- 2 Select **Page Format** and set properties on the right.

Note

These properties specify the default page format for diagram print-outs. You can set default values for each kind of diagram available for print in SCADE Suite projects.

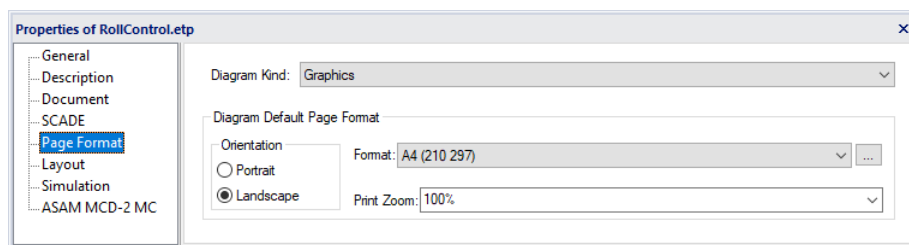


Figure 6.5: Page format properties of a project

- 3 From **Diagram Kind** drop-down list, specify the type of diagrams for which the format properties are specified with default values for print.
 - **Graphics:** page format options apply to any graphical diagram. These values also set the design area available by default to all graphical operators in the model.
 - **Text:** page format options apply to any text diagram.
 - **Constants:** page format options apply to any Constants View.
 - **Sensors:** page format options apply to any Sensors View.
 - **Types:** page format options apply to any Types View.

Note

For details, see [“Filtering Model Content and Reporting”](#) on page 41 in *SCADE LifeCycle User Manual*.

4 Set the **Diagram Default Page Format** options for all diagrams of the same kind specified above as follows:

- Set **Orientation** and select **Portrait** or **Landscape** page orientation for print-outs.
- From **Print Zoom** drop-down list, specify the zoom applied to the page print size. You set values between 10 and 500% or select **Fit To Page**.
- From **Format** drop-down list, set page size: A3, A4, B5, Legal, or Letter. These formats do not apply to SCADE LifeCycle Reporter output, but only to the use of print as described in [“Printing Active Views”](#) on page 153.

Click  to set more custom formats by defining their name, width, and height. Width is a value from 26 to 1000 and height a value from 37 to 1014.

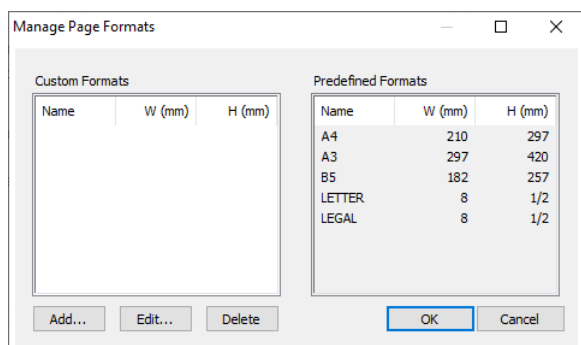


Figure 6.6: Customizing page format

To edit the layout properties of a project

- 1 Right-click a project file (.etp) in **Scade View** or **File View** and select *Properties*.
- 2 Select **Layout** and set properties on the right.

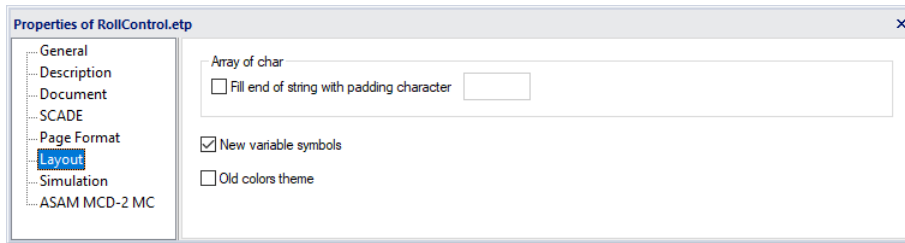


Figure 6.7: Layout properties of a project

- 3 Check **Fill end of string with padding character** to complete the end of entered strings if smaller than the specified dimension of char arrays. Specify the padding character for text field (default is space character).

Note

Without this option, you cannot enter a string smaller than the specified array dimension in the active project. The resulting display of string values depends on the activation of the Display char arrays as string option, see [“Selecting Views Options”](#) on page 79. If you display char arrays as string, set the use of \$ as padding character, and specify the following string `Constant1 : char ^8 = "stop"`, the displayed value is `Constant1 : char ^8 = "stop"`. Instead, without displaying char arrays as string, the displayed value is `Constant1 : char ^8 = ['s', 't', 'o', 'p', '$', '$', '$', '$']`.

- 4 Check **New variable symbols** to enable the usage of new graphical symbols on input/outputs, hidden inputs, local variables, probes, and signals. These symbols allow to save vertical space on diagrams. Uncheck this option to use symbols available before SCADE 6.
- 5 Check **Old color theme** to keep using the color theme applied to diagrams before 2022 R1.

RELATED TOPICS

- [“Editing Project Description Information”](#) on page 66 in *SCADE LifeCycle User Manual*
- [“Editing Document Information”](#) on page 67 in *SCADE LifeCycle User Manual*

Editing File Folder Properties

The folders you create in SCADE Suite to organize project files can be configured with a set of properties from **File View**.

To edit the general properties of a file folder

- 1 Right-click a project file folder in **File View** and select *Properties*.
- 2 Select **General** and set properties on the right.

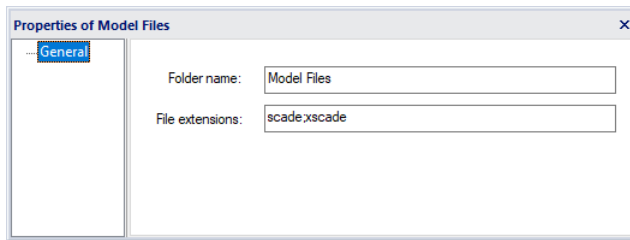


Figure 6.8: General properties of a file folder

- 3 In **Folder name**, specify the name of the folder.
- 4 In **File extensions**, specify which extensions are allowed for project files inserted in the folder. Use semi-colon (;) to list file extensions.

Editing Project Files Properties

The files created/inserted in SCADE Suite projects can be configured with a set of properties from **File View**.

- [“To edit the general properties of a file”](#)
- [“To edit the Simulation properties of a file”](#)
- [“To display and locate the content of a file”](#)

To edit the general properties of a file

- 1 Right-click a file in **File View** and select *Properties*.
- 2 Select **General** and set properties on the right.

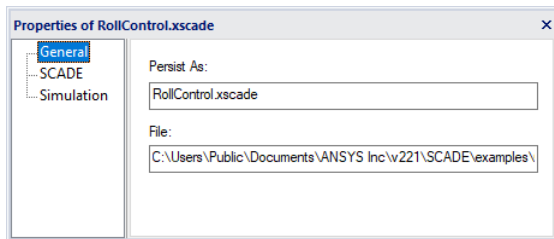


Figure 6.9: General properties of a file

- 3 Edit the name of the file in **Persist As** (resolved path of the file must be the same to allow using relative paths or environment variables). **File** shows the location of the file in the project folders.

To display and locate the content of a file

- 1 Right-click a file in **File View** and select *Properties*.
- 2 Select **SCADE** and set properties on the right. (Only available for XSCADE files)

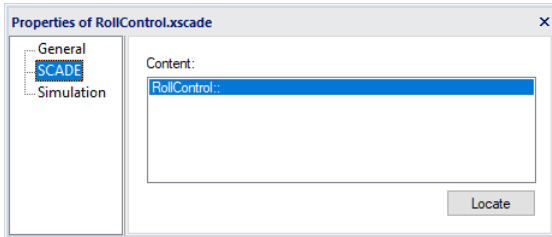


Figure 6.10: SCADE Properties of a file

The properties of a file list the top-level objects described in that file. These objects are designed by their path in the SCADE Suite model.

Note

In case of separate file, only one SCADE object is described in the file.

- 3 Select an object from the list and click **Locate** to highlight it in **Scade View**.

To edit the Simulation properties of a file

- 1 Right-click a file in **File View** and select *Properties*.
- 2 Select **Simulation** and set properties on the right.

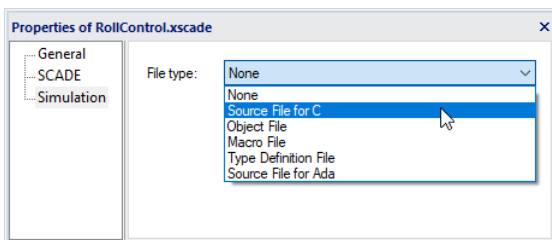


Figure 6.11: Simulation properties of a file

- 3 Specify from the **File type** drop-down list how the selected file is taken into account by SCADE Suite Simulation. Possible values are:

- **None** (default value) to specify that the file does not impact the simulation executable
- **Source File for C** to specify that the C file must be compiled and linked with SCADE Suite KCG C code to build an executable (Simulation, FMU, Twin Builder, ...).
- **Object File** to specify that the object file is linked to the simulation executable.
- **Macro File** to specify that the C header file containing overrides to the default macro definitions provided by Code Generator is included in the generation of the simulation executable (only for KCG C). For more details, see Part 4 [“Code Generation”](#).
- **Type Definition File** to specify that the C header file containing imported type definitions (see [“Setting Imported Types”](#) on page 214) is included in the generation of the simulation executable. Such files correspond to the -ext option of the compiler. For more details, see Part 4 [“Code Generation”](#). Only for C.
- **Source File for Ada** to specify that the Ada file must be compiled and linked with SCADE Suite KCG Ada code to build an executable (Simulation, FMU, Twin Builder, ...).

Editing Package Properties

The packages created in SCADE Suite projects can be configured with a set of properties.

- [“To edit the general properties of a package”](#)
- [“To edit the opening of packages”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About and notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a package

- 1 Right-click a package in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

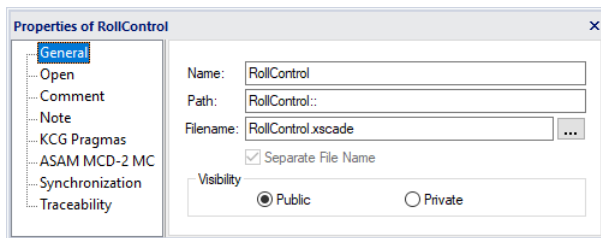


Figure 6.12: General properties of a package

- 3 Edit the name of the package in **Name**. **Path** shows the path to the package in the package hierarchy. In **File name**, change the file where the content of the selected package is stored.
- 4 Check/uncheck **Separate File Name** to specify whether the content of the selected package is stored in a separate file or not.

Note

For details, see [“Multifile Management”](#) on page 66. Separate File Name option is only available with operators and packages declared within a package. If unchecked, File name is uneditable.

- 5 Set the **Visibility** radio buttons to select the visibility status of the object declaration (see [“Managing Visibility of Package Declarations”](#) on page 130).

To edit the opening of packages

- 1 Right-click a package in **Scade View** and select *Properties*.
- 2 Select **Open** and set properties on the right.
- 3 Check/uncheck the name of other packages to enlarge/reduce the scope of the current namespace.

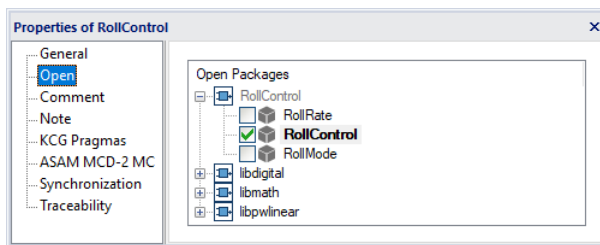


Figure 6.13: Opening properties of packages

Tips

- Check a project name or package to check all its packages or sub-packages.
 - Hold **Ctr L**-key to set separate items (project or package) one by one in multiselection and press **Space** bar to check/uncheck the highlighted list.
 - Hold **Shift**-key to set consecutive items (project or package) in multiselection and press **Space** bar to check/uncheck the highlighted list.
 - To check/uncheck by mouse click a multiselection list, keep the **Ctr L** or **Shift** keys pressed while clicking on check boxes.
-

Editing Diagram Properties

The diagrams created in SCADE Suite projects can be configured with a set of properties from **Scade View**.

- [“To edit the general properties of a diagram”](#)
- [“To edit the page format properties of a diagram”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484

To edit the general properties of a diagram

- 1 Right-click a diagram in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

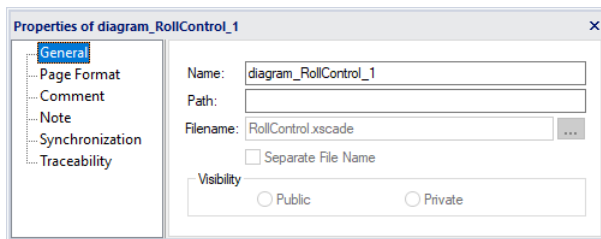


Figure 6.14: General properties of a diagram

- 3 Edit the name of the diagram in **Name**. The rest of the properties remains uneditable for diagrams.

To edit the page format properties of a diagram

- 1 Right-click a diagram in **Scade View** and select *Properties*.
- 2 Select **Page Format** and set properties on the right.

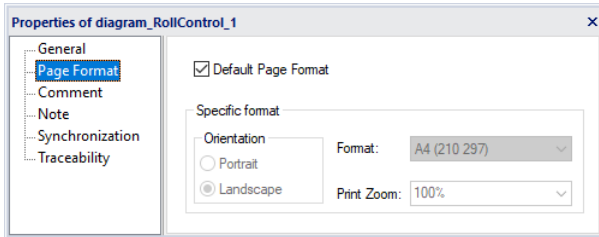


Figure 6.15: Page format properties of a project

- 3 Check **Default Page Format** to use the page format set by default at [properties level](#).
- 4 Uncheck **Default Page Format** to set **Specific Format** options for the selected diagram(s) as follows:
 - From **Format** drop-down list, set the page size: A3, A4, B5, Legal, or Letter. These formats do not apply to SCADE LifeCycle Reporter output, but only to the use of print as described in [“Printing Active Views”](#) on page 153.
 - Set **Orientation** and select **Portrait** or **Landscape** page orientation for print-outs.
 - From **Print Zoom** drop-down list, specify the zoom applied to the page print size. You set values between 10 and 500% or select **Fit To Page**.

Editing Type Properties

The types you define in SCAD Suite projects can be configured with a set of properties from **Scade View**.

- [“To edit the general properties of a type”](#)
- [“To edit the declaration properties of a type”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a type

- 1 Right-click a type in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

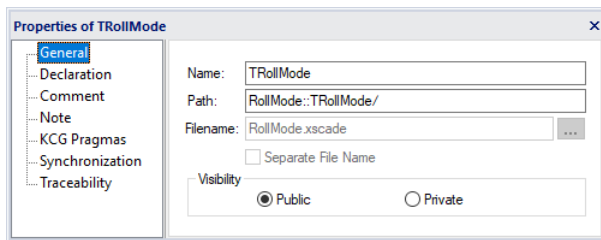


Figure 6.16: General properties of a type

- 3 Edit the name of the type in **Name**. You can also edit type names in **Types Views**. **Path** shows the path to the type in package hierarchy.
- 4 In **File name**, change the default file for types declared outside packages (see [“Multifile Management”](#) on page 66). Uneditable for types declared in packages.
- 5 Set the **Visibility** radio buttons to select the visibility status of the object declaration (see [“Managing Visibility of Package Declarations”](#) on page 130).

To edit the declaration properties of a type

- 1 Right-click a type in **Scade View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

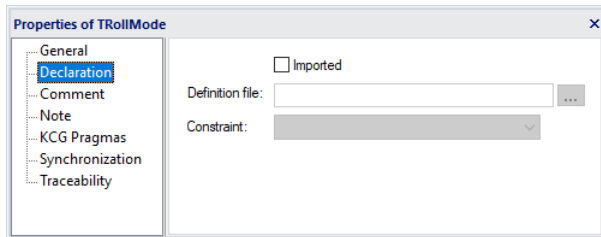


Figure 6.17: Declaration properties of a type

- 3 Check **Imported** to declare an imported type and enable **Definition file**.
- 4 Browse for files and specify the file path to the file that contains the definition of imported types (see [“Setting Imported Types”](#) on page 214). SCAD Suite prompts you for confirmation before adding the selected file into the project.
- 5 Set **Constraint** of imported type: select none, numeric, integer, float, signed, or unsigned in the drop-down list.

Editing Constant Properties

The constants you define in SCAD Suite projects can be configured with a set of properties from **Scade View** or from **Diagram Views**.

- [“To edit the general properties of a constant”](#)
- [“To edit the declaration properties of a constant”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a constant

- 1 Right-click a constant in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

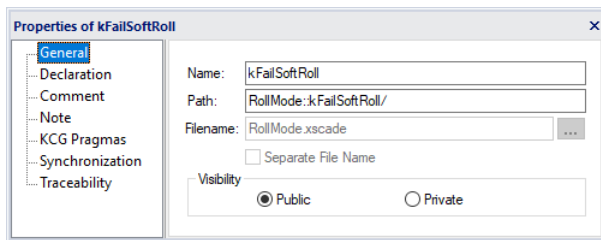


Figure 6.18: General properties of a constant

- 3 Edit the name of the constant in **Name**. **Path** shows the path to the constant in package hierarchy. You can also edit constant names in **Constants Views**.
- 4 In **File name**, change the default file for constants declared outside packages (see [“Multifile Management”](#) on page 66). Uneditable for constants declared in packages.
- 5 Set the **Visibility** radio buttons to select the visibility status of the object declaration (see [“Managing Visibility of Package Declarations”](#) on page 130).

To edit the declaration properties of a constant

- 1 Right-click a constant in **Scade View** and select *Properties*.

Note

Open the declaration properties from object instances in graphical diagrams.

- 2 Select **Declaration** and set properties on the right.

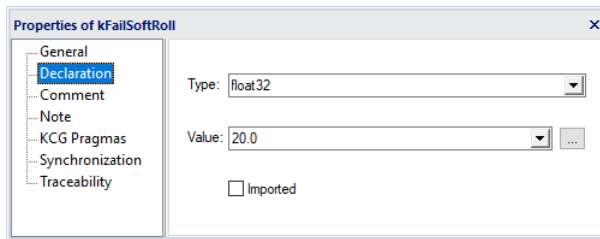


Figure 6.19: Declaration properties of a constant

- 3 In **Type**, select the type that defines the constant based from project types in the drop-down list, or enter directly a type definition or name.

Tip

While typing the first letters, you can enable the editing help to select from the list of type names available in the current project (see how [“To complete text automatically”](#) on page 262).

- 4 In **Value**, specify the constant with values from the drop-down list, or enter directly a value with respect to Scade language syntax.

Note

Value can be a char array value specified as string or character by character. For details, see [“To define char array values”](#) on page 210.

- 5 Check **Imported** to set a constant as imported (checked). If checked, it is impossible to assign a value to the constant. Changing a non-imported constant into an imported one deletes the specified value.

Editing Sensor Properties

The sensors you define in SCAD Suite projects can be configured with a set of properties from **Scade View** or from **Diagram Views**.

- [“To edit the general properties of a sensor”](#)
- [“To edit the declaration properties of a sensor”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About code integration, see [“Setting Sensors as Calibration Parameters”](#) on page 631, and [“Setting Sensors as Calibration Parameters”](#) on page 5 of *Gateway Guidelines for VeriStand*.
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a sensor

- 1 Right-click a sensor in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.



Figure 6.20: General properties of a sensor

- 3 Edit the name of the sensor in **Name**. **Path** shows the path to the sensor in package hierarchy. You can also edit sensor names in **Sensors Views**.
- 4 In **File name**, change the default file for sensors declared outside packages (see [“Multifile Management”](#) on page 66). Uneditable for sensors declared in packages.

To edit the declaration properties of a sensor

- 1 Right-click a sensor in **Scade View** and select *Properties*.

Note

You can open the properties of a sensor by selecting sensor references in Diagram Views. The Declaration properties correspond to the sensor itself. Additional Use properties display uneditable fields showing the graphical position and size of the constant.

- 2 Select **Declaration** and set properties on the right.

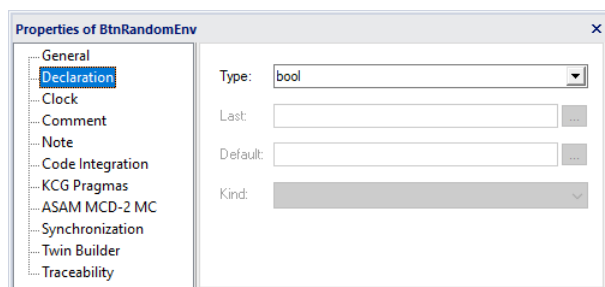


Figure 6.21: Declaration properties of a sensor

- 3 In **Type**, select the type that defines the sensor from project types in the drop-down list, or enter directly a type definition or name.

Tip

While typing the first letters, enable auto-completion to select from the list of types available in the project (see how ["To complete text automatically"](#) on page 262).

Editing Operator Properties

The operators you define in SCADE Suite projects can be configured with a set of properties from **Scade View**.

Note

The properties of operators and imported operators defined in Scade View are edited similarly as described below.

- [“To edit the general properties of an operator”](#)
- [“To edit the type variables properties of an operator”](#)
- [“To edit the declaration properties of an operator”](#)
- [“To edit the code integration properties of operators”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488
- About Coverage, see [“Defining Local Variables as Observable”](#) on page 67 in *SCADE Test User Manual*

To edit the general properties of an operator

- 1 Right-click an operator in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

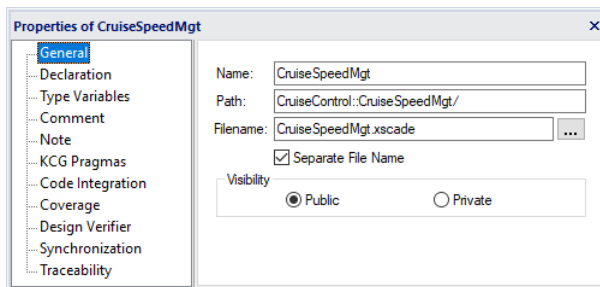


Figure 6.22: General properties of an operator

- 3 Edit the name of the operator in **Name**. **Path** shows the path to the operator in the package hierarchy.

- 4 In **File name**, change the file where the content of the selected operator is to be stored. For details, see [“Multifile Management”](#) on page 66.
- 5 Check/uncheck **Separate File Name** to specify whether the content of the selected operator is stored in a separate file or not. For details, see [“Multifile Management”](#) on page 66.

Note

The Separate Name File option is only available with operators and packages declared within a package. If unchecked, File name becomes uneditable.

- 6 Set the **Visibility** radio buttons to select the visibility status of the object declaration (see [“Managing Visibility of Package Declarations”](#) on page 130).

To edit the declaration properties of an operator

- 1 Right-click an operator in **Scade View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

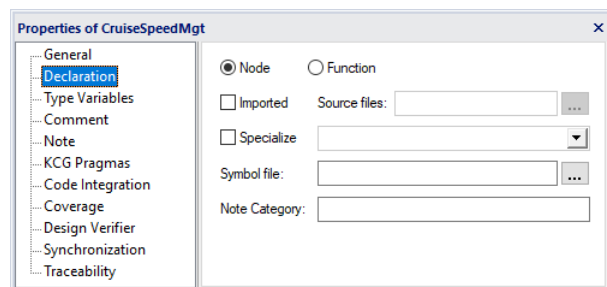


Figure 6.23: Declaration properties of an operator

- 3 Set **Node** or **Function** radio button to declare the operator either as a node with memory or as a function without memory.
- 4 Check **Imported** to declare imported nodes or functions, and set the external code implementation with the operator in **Source files**. Double clicking the operator opens a Textual View showing the file content.

- 5 Check **Specialize** to declare the operator as a specializing operator and select the operator to specialize from the associated drop-down list.

Note

Setting this property is required when specializing an imported operator as described in [“Declaring and Editing Specialized Operators”](#) on page 196.

- 6 Specify a symbol file associated with the operator in **Symbol file**.
- 7 Determine if the operator is associated with a given note category from **Note Category**.

To edit the type variables properties of an operator

- 1 Right-click an operator in **Scade View** and select *Properties*.
- 2 Select **Type Variables** and set properties on the right.

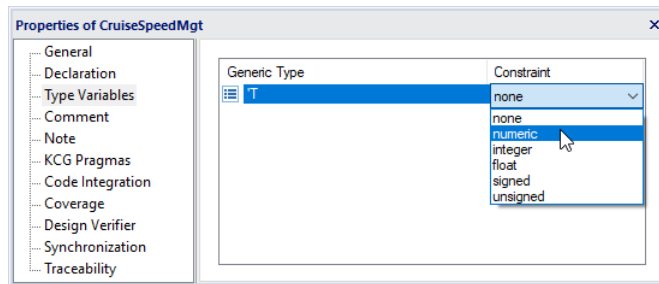


Figure 6.24: Type variables properties of an operator

- 3 Select the generic type and set its **Constraint** as none, numeric, integer, float, signed, or unsigned in the drop-down list.

To edit the code integration properties of operators

- 1 Right-click an operator in **Scade View** and select *Properties*.

- 2 Select **Code Integration** and set properties on the right.

Note

Code generator extensions may require information about the periodicity of code execution for the specified root operator. For details, see [“Setting Periodicity of Code Execution”](#) on page 852.

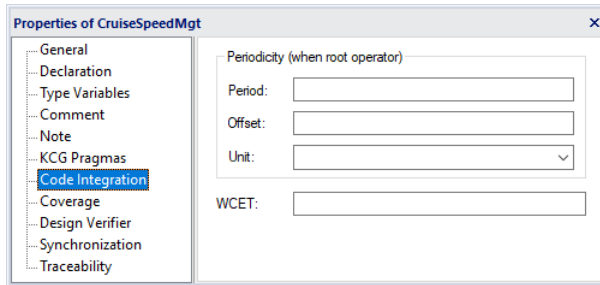


Figure 6.25: Code integration properties of an operator

- 3 Specify the period of execution in **Period**.
- 4 Specify the offset of the period in **Offset**.
- 5 Set the unit applied to the periodicity in seconds (s), milliseconds (ms), or microseconds (μ s) from the **Unit** drop-down list.
- 6 Specify worst-case execution time information in **WCET**. For details, see [“Specifying WCET Information from TSO Analysis on KCG-Generated Code”](#) on page 901.

Editing Operator Call Properties

The operator calls (including calls to textual diagrams) are instances you set and define graphically in SCADE Suite projects. They can be configured with a set of properties from **Diagram Views**. Operator calls are user-defined operators or any of the predefined or library operators.

Note

By definition, predefined and library operators have no editable declaration properties. Instead, the declaration properties of user-defined operators can be edited from graphical instances in Diagram Views or from project declaration in Scade View. For details, see how [“To edit the declaration properties of an operator”](#).

- [“Editing Use Properties of Operator Calls”](#)
- [“Editing Size Parameters of Operator Calls”](#)
- [“Editing Parameters of Operator Calls”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488
- About Coverage, see [“Defining Local Variables as Observable”](#) on page 67 in *SCADE Test User Manual*

Editing Use Properties of Operator Calls

The behavior and identification of operator instances used in graphical diagrams can be modified by properties as described below.

To edit the use properties of an operator call

- 1 Right-click an operator call in the **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

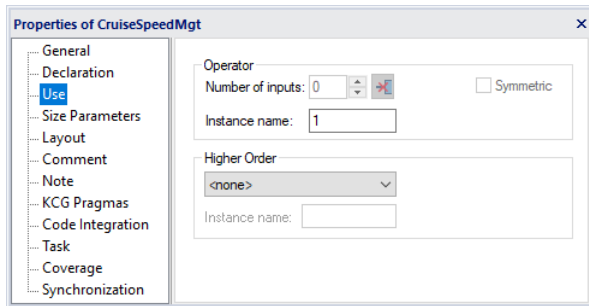


Figure 6.26: Use properties of an operator call

3 Set the **Operator** call properties as follows:

- Add/remove inputs from the **Number of inputs** selector (only for predefined operators).

Tips

Enter the number of inputs directly in the field. Press  and click on the symbol to remove the last input or click a given pin to remove the corresponding input.

- Check **Symmetric** to display the symmetric form of an operator (only for some predefined operators).
- Change the instance name of the selected call in **Instance name** with caution. For details, see [“Editing Instance Names”](#) on page 487.

Note

To ensure the uniqueness of instance names, SCADE Suite Editor computes automatically an instance name for each operator instance used in the model.

4 Set the **Higher Order** properties of the operator call as follows:

- Enable the call of predefined or user-defined operators with condition activation, restart instantiation, or iteration patterns from the drop-down list. Available patterns:

<none> basic operator call without any pattern

Boolean activate	to enable the Boolean activate pattern
Clocked activate	to enable the clocked activate pattern
restart	to enable the restart pattern
map	to enable the map iteration pattern
fold	to enable the fold iteration pattern
mapfold	to enable the mapfold iteration pattern
mapfoldi	to enable the mapfoldi iteration pattern
mapfoldw	to enable the mapfoldw iteration pattern
mapfoldwi	to enable the mapfoldwi iteration pattern
mapi	to enable the mapi iteration pattern
mapw	to enable the mapw iteration pattern
mapwi	to enable the mapwi iteration pattern
foldi	to enable the foldi iteration pattern
foldw	to enable the foldw iteration pattern
foldwi	to enable the foldwi iteration pattern

- Specify the number of accumulators for any mapfold iterator variant in **Accumulator**. Not editable when selecting other array iterations.
- Specify the array bound for any iterator in **Array Bound**. Only editable when selecting array iterations.

Note

For details about accumulators and array bounds, see [“Designing with Array Iterators”](#) on page 378.

-
- Set default values for Boolean activate see [“Setting and Designing Boolean Activation”](#) on page 402.
 - Set clock for Clocked activate see [“Setting and Designing Clocked Activation”](#) on page 405.
 - Modify the iterator instance name in **Instance name**.

Editing Parameters of Operator Calls

Hidden inputs on user-defined operator instances and on a few predefined operator instances are used in graphical diagrams to display parameters. Such parameters can be defined in properties as described below.

Note

Hidden inputs do not belong to Scade language syntax.

To edit the parameter properties of operator calls

- 1 Right-click an operator call in the **Diagram View** and select *Properties*.
- 2 Select **Parameters** and set properties on the right.

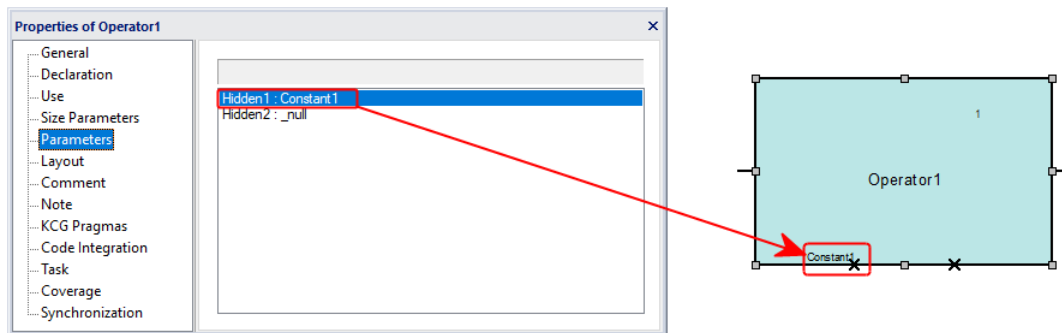


Figure 6.27: Hidden input properties of an operator call

- 3 Specify on each line the value of hidden inputs associated with the operator. Some parameters may be specified from dedicated option fields.

Tip

Drag and drop valid variables on the hidden input mark of the symbol to define parameters directly from graphical diagrams. The value of the corresponding parameter is automatically updated in the Properties.

Editing Size Parameters of Operator Calls

The size parameters of parameterized operator instances used in graphical diagrams can be defined in properties as described below.

To edit the size parameter properties of operator calls

- 1 Right-click an operator call in the **Diagram View** and select *Properties*.
- 2 Select **Size Parameters** and set properties on the right.

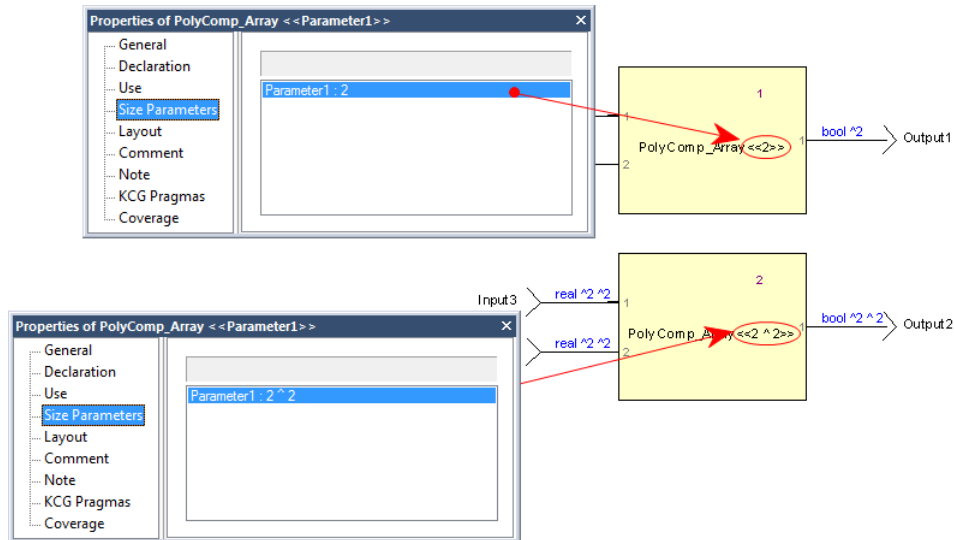


Figure 6.28: Size parameter properties of parameterized operator calls

- 3 Specify the size of the parameterized operator instance as wanted.

Editing Input/Output Properties

The input, output, or hidden input variables created to define operator interfaces can be configured with a set of properties from **Scade View**.

- [“To edit the general properties of an input, output, or hidden input”](#)
- [“To edit the declaration properties of an input, output, or hidden input”](#)
- [“To edit the use properties of interface variable references”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488
- About Coverage, see [“Defining Local Variables as Observable”](#) on page 67 in *SCADE Test User Manual*

To edit the general properties of an input, output, or hidden input

- 1 Right-click an item from operator interface in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

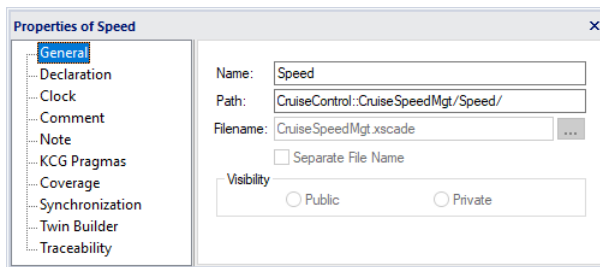


Figure 6.29: General properties of input/output, or hidden input

- 3 Edit the name of the item in **Name**. **Path** shows the path to the item in the package hierarchy.
- 4 In **File name**, check the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of an input, output, or hidden input

- 1 Right-click an item from operator interface in **Scade View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

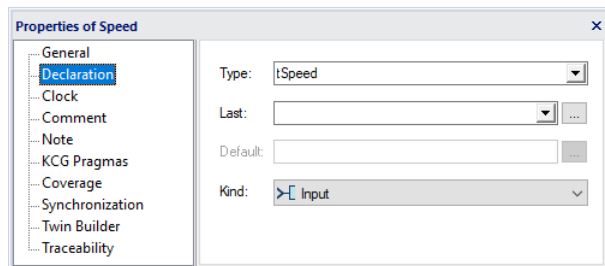


Figure 6.30: Declaration properties of input/output, or hidden input

- 3 In **Type**, select the type that defines the element based on project types in the drop-down list, or enter directly a type definition or name.

Tip

While typing the first letters, use editing help to select type the list of types defined in the current project (see how [“To complete text automatically”](#) on page 262).

- 4 In **Last**, specify the last value to be initialized with the input/output (for the first instant). The last value can be any expression. Click the browsing button to enter long expressions in the **Edit Value** window.

Note

Last value can be a char array value specified as string or character by character. For details, see [“To define char array values”](#) on page 210.

- 5 In **Default** (enabled for outputs), specify a default value when the variable is not defined in a given state. Click the browsing button to enter long expressions in the **Edit Value** window.

Note

If no default value set, the default is defined by the 'last' of the variable. For details about last and default values with variables, see [“Creating Lasts of Variables and Signals”](#) on page 243.

- 6 Change interface declaration into **Input**, **Hidden**, **Output**, **Local**, or **Probe** from **Kind** drop-down list.

To edit the use properties of interface variable references

- 1 Right-click an interface variable instance in any **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

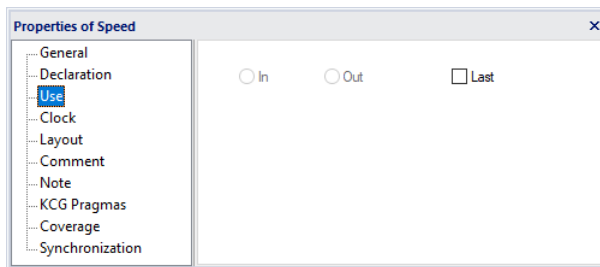


Figure 6.31: Use properties of an interface variable instance

- 3 Check **Last** to define the last reference to the variable instance.

Note

Creating last references to interface variables can be done directly from diagram instances as explained in [“Creating Lasts of Variables and Signals”](#) on page 243.

Editing Probe Properties

The probes you create in SCADE Suite projects can be configured with a set of properties from **Scade View**.

- [“To edit the general properties of a probe”](#)
- [“To edit the use properties of probe references”](#)
- [“To edit the declaration properties of a probe”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488
- About Coverage, see [“Defining Local Variables as Observable”](#) on page 67 in *SCADE Test User Manual*

To edit the general properties of a probe

- 1 Right-click a probe from an operator interface in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

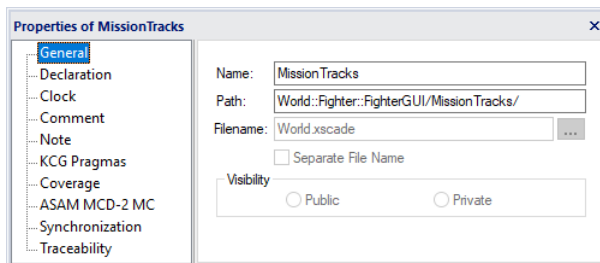


Figure 6.32: General properties of probes

- 3 Edit the name of the item in **Name**. **Path** shows the path to the item in the package hierarchy.
- 4 In **File name**, check the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of a probe

- 1 Right-click a probe(s) from an operator interface in **Scade View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

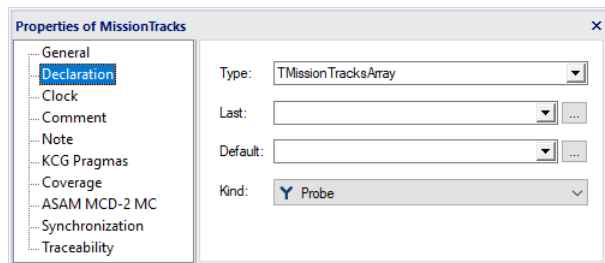


Figure 6.33: Declaration properties of probes

- 3 In **Type**, select the type that defines the element based on project types in the drop-down list, or enter directly a type definition or name.

Tip

While typing the first letters, enable auto-completion to select from the list of types available in the project (see how ["To complete text automatically"](#) on page 262).

- 4 In **Last**, specify the last value to be initialized with the input/output (for the first instant). The last value can be any expression. Click the browsing button to enter long expressions in the **Edit Value** window.

Note

Last value can be a char array value specified as string or character by character. For details, see ["To define char array values"](#) on page 210.

- 5 In **Default**, specify a default value when the probe is not defined in a given state. Click the browsing button to enter long expressions in the **Edit Value** window.

Note

If no default value is set, the default is defined by the 'last' of the variable. For details about last and default values with variables, see [“Creating Lasts of Variables and Signals”](#) on page 243.

- 6 Change probe declaration into **Input**, **Hidden**, **Output**, or **Local** from **Kind** drop-down list.

To edit the use properties of probe references

- 1 Right-click a probe instance(s) in any **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

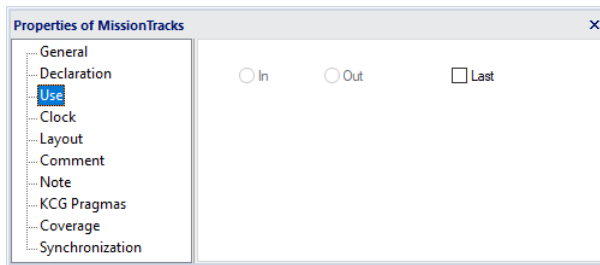


Figure 6.34: Use properties of a probe instance

- 3 Check **Last** to define the last reference to the probe instance.

Note

Creating last references to local variables can be done directly from diagram instances as explained in [“Creating Lasts of Variables and Signals”](#) on page 243.

Editing Local Variable Properties

The local variables you define in SCAD Suite projects can be configured with a set of properties from **Scade View** or from **Diagram Views**.

- [“To edit the general properties of a local variable”](#)
- [“To edit the clock properties of local variable”](#)
- [“To edit the declaration properties of a local variable”](#)
- [“To edit the use properties of local variable instances”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a local variable

- 1 Right-click a local variable in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

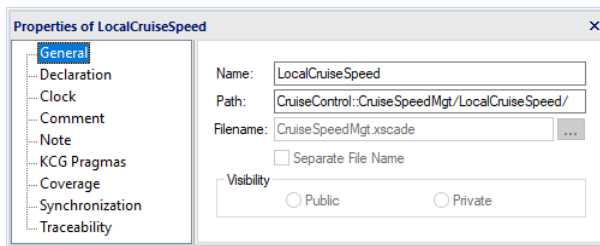


Figure 6.35: General properties of a local variable

- 3 Edit the name of the item in **Name**. **Path** shows the path to the item in the package hierarchy.
- 4 In **File name**, check the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of a local variable

- 1 Right-click a local variable(s) in **Scade View** and select *Properties*.

- 2 Select **Declaration** and set properties on the right.

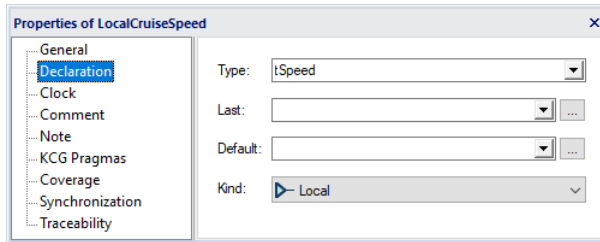


Figure 6.36: Declaration properties of a local variable

- 3 In **Type**, set the type that defines the element based on any project types using the completion assistant or entering directly its definition.

Tip

While typing the first letters, enable auto-completion to select from the list of types available in the project (see how [“To complete text automatically”](#) on page 262).

- 4 In **Last**, specify the last value to be initialized with the input/output (for the first instant). The last value can be any expression. Click the browsing button to enter long expressions in the **Edit Value** window. Last value can be a char array value specified as string or character by character. For details, see [“To define char array values”](#) on page 210.
- 5 In **Default**, specify a default value when the local variable is not defined in a given state. Click the browsing button to enter long expressions in the **Edit Value** window.

Note

If no default value is set, the default is defined by the 'Last' of the variable. For details, see [“Creating Lasts of Variables and Signals”](#) on page 243.

- 6 Change variable declaration into **Input**, **Hidden**, **Output**, or **Probe** from **Kind** drop-down list.

To edit the use properties of local variable instances

- 1 Right-click a local variable instance(s) in any **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

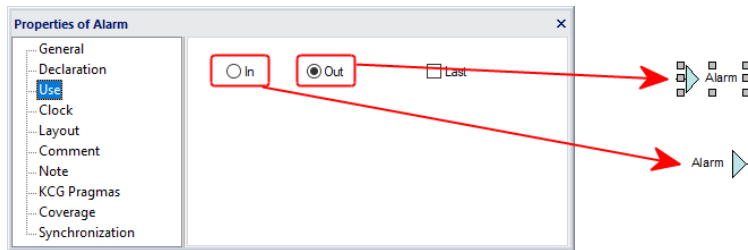


Figure 6.37: Use properties of a local variable instance

- 3 Set the local variable instance as **In** or **Out** to indicate whether the variable is used as an input or an output in the dataflow.
- 4 Check **Last** to define the last reference to the variable instance. Creating last references to local variables can be done from diagram instances as explained in [“Creating Lasts of Variables and Signals”](#) on page 243.

To edit the clock properties of local variable

- 1 Right-click a local variable in **Scade View** and select *Properties*.
- 2 Select **Clock** and set properties on the right.

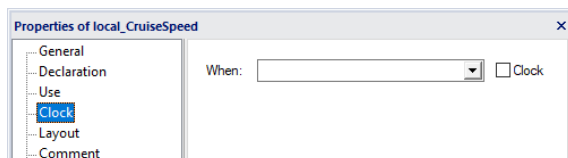


Figure 6.38: Clock properties of a local variable

- 3 Check **Clock** to set a given variable as a clock and specify the clock expression of the variable in **When** or select it from the drop-down list.

Editing Signal Properties

The signals you define in SCADE Suite projects can be configured with a set of properties from **Scade View** or **Diagram Views**.

- [“To edit the general properties of a signal”](#)
- [“To edit the use properties of signal instances”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a signal

- 1 Right-click a signal in **Scade View** and select *Properties*.
- 2 Select **General** and set properties on the right.

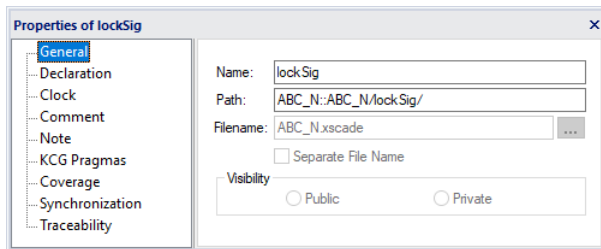


Figure 6.39: General properties of a signal

- 3 Edit the name of the item in **Name**. **Path** shows the path to the item in the package hierarchy.
- 4 In **File name**, check the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the use properties of signal instances

- 1 Right-click a signal instance in any **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

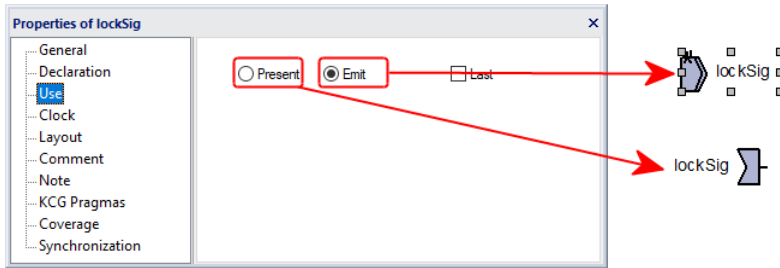


Figure 6.40: Use properties of a signal instance

- 3 Set the signal reference as **Present** or **Emit** from the radio buttons.
- 4 Check **Last** to define the last reference to the signal instance.

Note

Creating last references to local variables can be done directly from diagram instances as explained in [“Creating Lasts of Variables and Signals”](#) on page 243.

Editing If Block and If Node Properties

The If blocks you define in diagrams can be configured with a set of properties from **Scade View** and/or from **Diagram Views**.

- [“To edit the general properties of If Blocks”](#)
- [“To edit the declaration properties of If Node actions”](#)
- [“To edit the declaration properties of If Nodes”](#)
- [“To edit the textual definition body of If Node actions”](#)
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of If Blocks

- 1 Right-click an If Block in **Scade View** or in **Diagram Views** and select *Properties*.
- 2 Select **General** and set properties on the right.

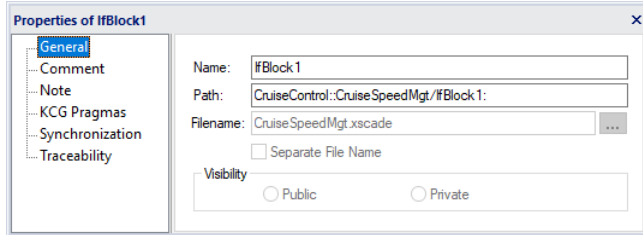


Figure 6.41: General properties of an If Block

- 3 Edit the name of the If Block in **Name** (only editable property). **Path** shows the path to the conditional block in the control flow hierarchy. **File name** shows the file in which the declared element is saved.

To edit the declaration properties of If Nodes

- 1 Right-click an If Node and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

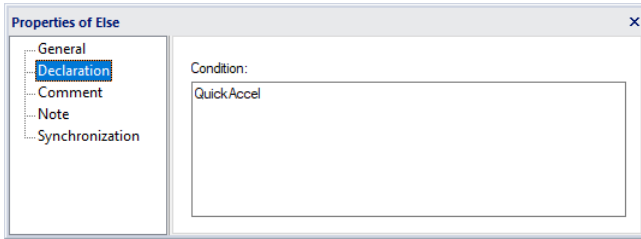


Figure 6.42: Declaration properties of an If Node

- 3 Edit the value of the condition associated with current If Node.

To edit the declaration properties of If Node actions

- 1 Right-click the action of an If Node and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

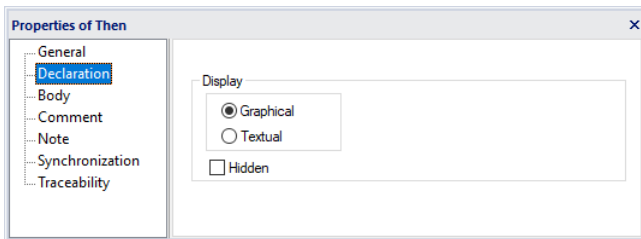


Figure 6.43: Declaration properties of an If Node action

- 3 Set the display modes of the action as follows:
 - Set definition mode as **Graphical** or **Textual** from radio buttons (see [“Defining Textual States or Actions”](#) on page 266).
 - Select **Hidden** to hide embedded display into a separate diagram (see [“Switching Embedded/Hidden Action Display”](#) on page 201).

To edit the textual definition body of If Node actions

- 1 Right-click the action of an If Node and select *Properties*.

- 2 Select **Body** and edit the action definition on the right. If the action is set in graphical definition mode from its **Declaration** properties, the action definition is uneditable. To define body textually, right-click the action in diagram and select *Convert to Textual*.

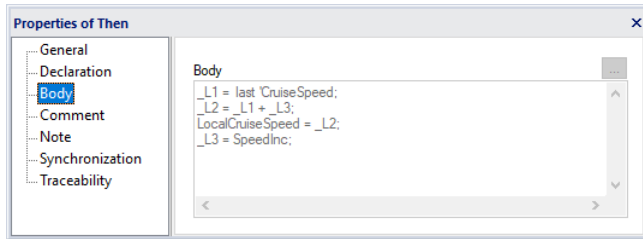


Figure 6.44: Body properties for editing textual action definition

Tip

For long text edition, click the upper right button to open the Edit Value window and edit the action definition.

Editing When Block and When Branch Properties

The conditional blocks you define in diagrams can be configured with a set of properties from **Scade View** and/or from **Diagram Views**.

- [“To edit the general properties of When Blocks”](#)
- [“To edit the declaration properties of When Blocks”](#)
- [“To edit the declaration properties of When Branches”](#)
- [“To edit the declaration properties of When Branch actions”](#)
- [“To edit the textual definition body of When Branch actions”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- For When Branches, you can add comments or notes on the corresponding action only
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859

To edit the general properties of When Blocks

- 1 Right-click a When Block in **Scade View** or in **Diagram Views** and select *Properties*.
- 2 Select **General** and set properties on the right.

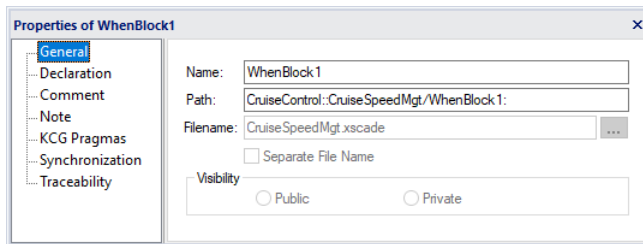


Figure 6.45: General properties of a When Block

- 3 Edit the name of the When Block in **Name** (only editable property). **Path** shows the path to the conditional block in the control flow hierarchy. **File name** shows the file in which the declared element is saved.

To edit the declaration properties of When Blocks

- 1 Right-click a When Block in **Scade View** or in **Diagram Views** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

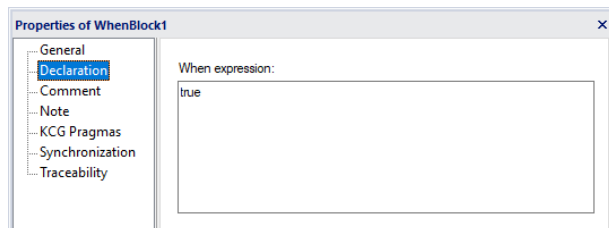


Figure 6.46: Declaration properties of a When Block

- 3 Edit the conditional expression associated with current When Block.

To edit the declaration properties of When Branches

- 1 Right-click a When branch and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

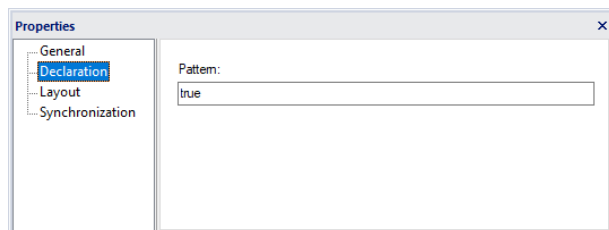


Figure 6.47: Declaration properties of a When Branch

- 3 Edit the value of the When Branch (value must be an enumerated type).

To edit the declaration properties of When Branch actions

- 1 Right-click the action of a When Branch and select *Properties*.

- 2 Select **Declaration** and set properties on the right.

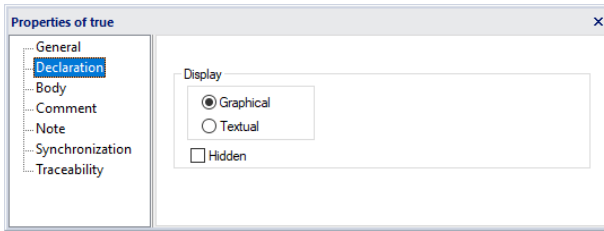


Figure 6.48: Declaration properties of a When Branch action

- 3 Set the display modes of the action as follows:
 - Set definition mode as **Graphical** or **Textual** (see [“Defining Textual States or Actions”](#) on page 266).
 - Select **Hidden** to hide embedded display into a separate diagram (see [“Switching Embedded/Hidden Action Display”](#) on page 201).

To edit the textual definition body of When Branch actions

- 1 Right-click the action of a When Branch and select *Properties*.
- 2 Select **Body** and edit the action definition on the right. If the action is set in graphical definition mode from its **Declaration** properties, the action definition is uneditable. To define body textually, right-click the action in diagram and select *Convert to Textual*. For long text editing, click the upper right button to open the **Edit Value** window and edit the action definition.

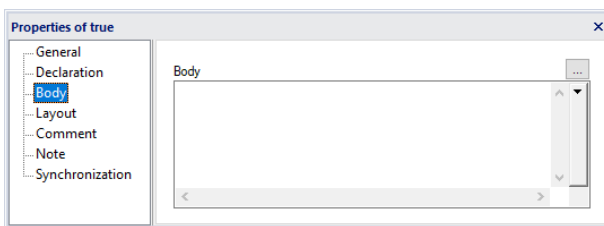


Figure 6.49: Body properties for editing textual action definition

Editing State Machine Properties

The State Machines you define in diagrams can be configured with a set of properties from **Scade View** or **Diagram Views**.

- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of State Machine

- 1 Right-click a State Machine in **Scade View** or in **Diagram Views** and select *Properties*.
- 2 Select **General** and set properties on the right.

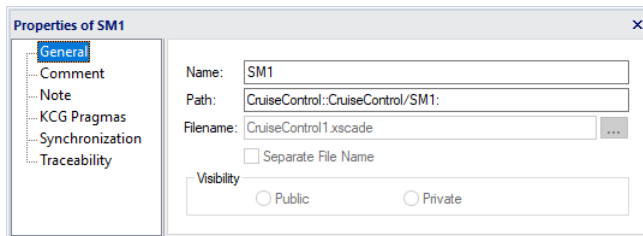


Figure 6.50: Declaration properties of a State Machine

- 3 Edit the name of the State Machine in **Name**. **Path** is uneditable and shows the path to the State Machine in the control flow hierarchy.
- 4 **File name** shows the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

Editing State Properties

The states you define when designing State Machines can be configured with a set of properties from **Scade View** or **Diagram Views**.

- [“To edit the general properties of a state”](#)
- [“To edit the declaration properties of a state”](#)
- [“To edit the outgoing transition priorities of a state”](#)
- [“To edit the textual definition body of a state”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859
- About traceability, see [“Accessing Traceability Information”](#) on page 488

To edit the general properties of a state

- 1 Right-click the state of a State Machine and select *Properties*.
- 2 Select **General** and set properties on the right.

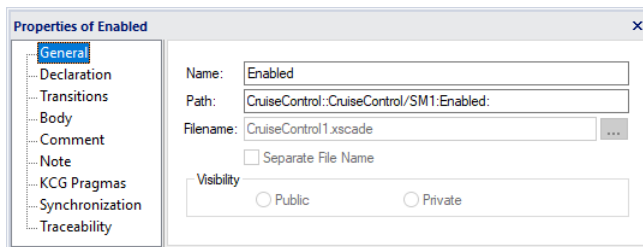


Figure 6.51: General properties of states

- 3 Edit the name of the state in **Name**. **Path** is uneditable and shows the path to the state in the control flow hierarchy.
- 4 **File name** shows the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of a state

- 1 Right-click the state of a State Machine and select *Properties*.

2 Select **Declaration** and set properties on the right.

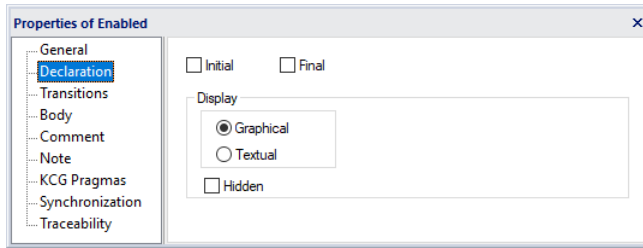


Figure 6.52: Declaration properties of states

3 Check **Initial** to set an initial state and/or check **Final** to set a final state.

4 Set the display modes of the state as follows:

- Set **Graphical** or **Textual** (see [“Defining Textual States or Actions”](#) on page 266).
- Select **Hidden** to hide embedded display into a separate diagram (see [“Switching Embedded/Hidden Action Display”](#) on page 201).

To edit the outgoing transition priorities of a state

1 Right-click a state and select *Properties*.

2 Select **Transitions** and set properties on the right.

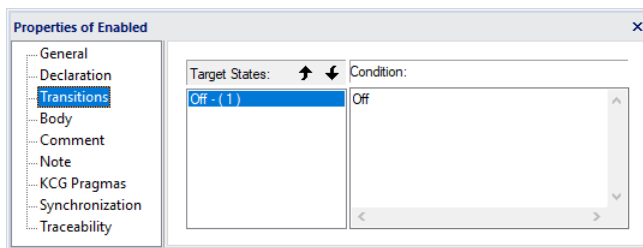


Figure 6.53: Priorities of outgoing state transitions

3 In **Target States** list, use Move Up/Down arrows to modify the priority of transitions. The definition of the selected transition displays on the right.

To edit the textual definition body of a state

- 1 Right-click the state and select *Properties*.
- 2 Select **Body** and edit the state definition on the right. If the state is set in graphical definition mode from its **Declaration** properties, the state definition is uneditable. To define body textually, right-click the state in diagram and select *Convert to Textual*. For long text edition, click the upper right button to open the **Edit Value** window and edit the state definition.

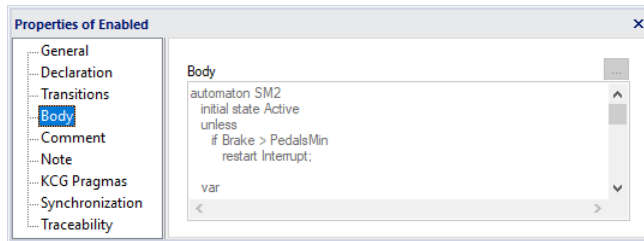


Figure 6.54: Body properties for editing textual state definition

Editing Transition Properties

The transitions you draw to connect states when designing State Machines can be configured with a set of properties from **Diagram Views**.

- [“To edit the general properties of a transition”](#)
- [“To edit the activation properties of a transition”](#)
- [“To edit the declaration properties of a transition”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484

To edit the general properties of a transition

- 1 Right-click a transition between states and select *Properties*.
- 2 Select **General** and set properties on the right.

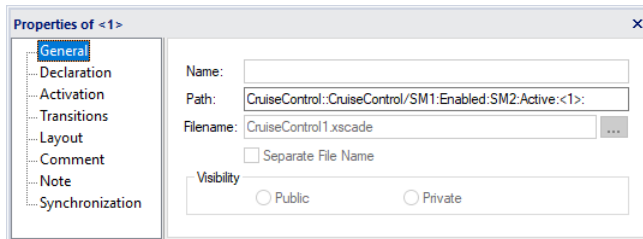


Figure 6.55: Declaration properties of transitions

- 3 **Name** is uneditable. **Path** is uneditable and shows the path to the transition in the control flow hierarchy.
- 4 **File name** shows the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of a transition

- 1 Right-click a transition between states and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

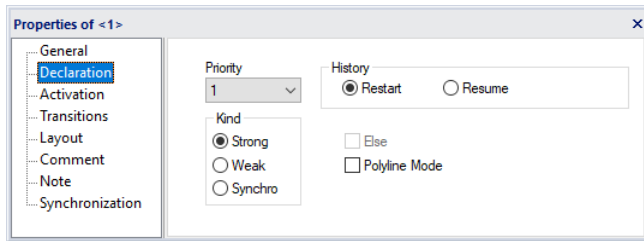


Figure 6.56: Declaration properties of transitions

3 Change the priority of the transition from **Priority** drop-down list.

Tip

To manage several transition priorities at once, it is more convenient to switch the order of priorities from the state properties as described in [“To edit the outgoing transition priorities of a state”](#) on page 472. Changing a priority modifies the priorities of all outgoing transitions from the source state. Priority is exchanged only between transitions of the same type (weak, strong, synchro).

- 4 Set the type of transition preemption as **Strong**, **Weak**, or **Synchro** with the corresponding radio buttons.
- 5 Determine the behavior of the transition with **History** radio buttons:
 - **Restart** (default): to activate and reinitialize the target state execution.
 - **Resume**: to resume the target state execution when last activated.
 About transitions in diagrams, see [“Drawing and Editing State Transitions”](#) on page 172.
- 6 Check/uncheck the following check boxes as needed:
 - **Else**: to set a fork transition as an “else” branch with no condition. About fork transitions, see [“Drawing and Editing Fork Transitions”](#) on page 175.

- **Polyline Mode:** to switch the display of transition in diagrams between straight lines and curved lines. About default modes, see [“Setting Transition Creation Modes”](#) on page 178
-

Tip

Change transition directly from Diagram Views. Right-click the transition graphical item and select *Set transition layout as Bezier* or *Set transition layout as Polyline*.

To edit the activation properties of a transition

- 1 Right-click a transition between states and select *Properties*.
- 2 Select **Activation** and set properties on the right.

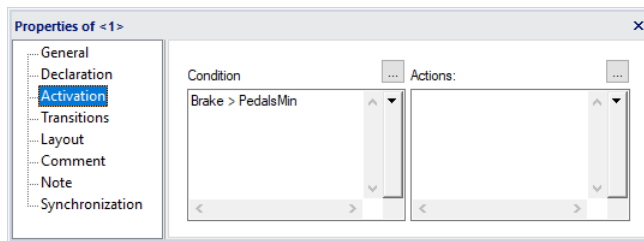


Figure 6.57: Activation properties of transitions

- 3 Write the expression that defines the trigger condition of the transition in the **Condition** panel.
 - 4 Write the expression defining the effect of the transition in **Actions** panel.
-

Tip

Click the browsing button to enter long expressions in the Edit Value window.

For expressing conditions and actions, see [“Setting Transition Conditions and Actions”](#) on page 178.

Editing Wire Properties

The wires you draw to connect design element in SCADE Suite projects can be configured with a set of properties from **Diagram Views**.

- [“To check the general properties of a connecting wire”](#)
- [“To edit the declaration properties of a connecting wire”](#)
- [“To edit the local definition properties of a connecting wire”](#)
- [“To edit the clock properties of a connecting wire”](#)
- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859

To check the general properties of a connecting wire

- 1 Right-click a wire between elements in diagrams and select *Properties*.
- 2 Select **General** and set properties on the right.

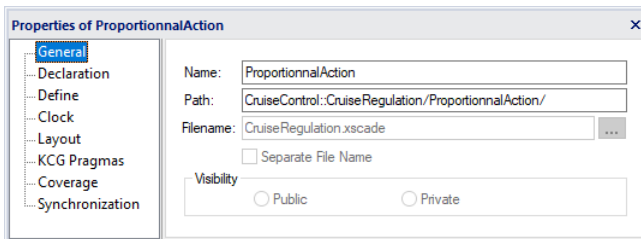


Figure 6.58: General properties of connecting wires

- 3 **Name** shows the name of the element. By default, it shows the internal variable reference assigned automatically by SCADE Suite when drawing wires between elements. **Path** is uneditable and shows the path to the internal variable in the control and data flow hierarchy.
- 4 **File name** shows the file in which the declared element is saved. For details, see [“Multifile Management”](#) on page 66.

To edit the declaration properties of a connecting wire

- 1 Right-click a wire between elements in diagrams and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

Note

Data on these properties relates to the variable conveyed by the wire. This variable is defined by the symbol at the origin of the wire (left part of equation). Deleting the wire does not delete the variable definition. Delete it from the project in Scade View.

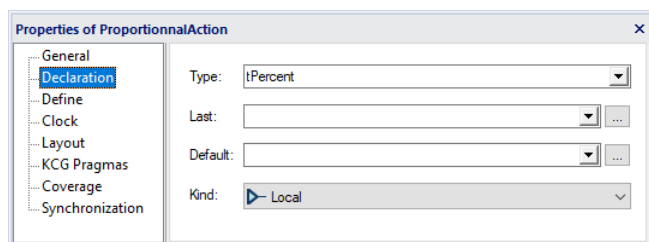


Figure 6.59: Declaration properties of connecting wires

- 3 In **Type**, check the computed type of the internal variable. Changing the type is not necessary except for named variables.
- 4 In **Last**, select the last initialization value assigned to the local variable from the drop-down list or enter the value directly.
- 5 In **Default**, select the default value assigned to the local variable from the drop-down list or enter the value directly.

To set Default or Last values, declare first a local variable from the Define properties as described below.

To edit the local definition properties of a connecting wire

- 1 Right-click a wire between elements in diagrams and select *Properties*.
- 2 Select **Define** and set properties on the right.

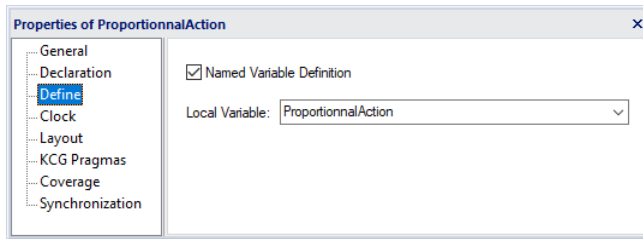


Figure 6.60: Define properties of connecting wires

- 3 Check **Named Variable Definition** to enable the production of a local variable by the selected connection.
- 4 Specify the name of the variable that the connection produces from the **Local Variable** drop-down list or enter a new name. If you enter a new name in the list, a new local variable is created.

To edit the clock properties of a connecting wire

- 1 Right-click a wire between elements in diagrams and select *Properties*.
- 2 Select **Clock** and set properties on the right.



Figure 6.61: Define properties of connecting wires

- 3 Check **Clock** to set a given flow as a clock and type the clock expression of the flow in **When** or select it from the drop-down list. To set clock options, declare first a local variable from the Define properties as described above.

Editing Assertion Properties

The type of assertions you create in SCADE Suite projects can be defined from **Diagram Views**.

- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484
- About Code Generator pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859

To edit the declaration properties of an assertion

- 1 Right-click an assertion in the **Diagram View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

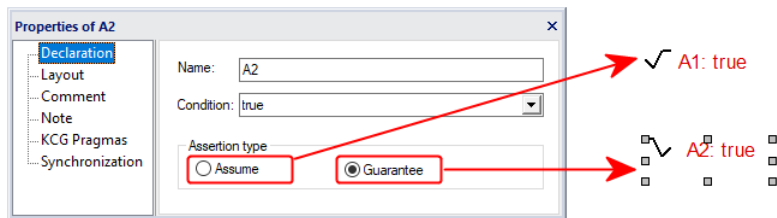


Figure 6.62: Declaration property of assertions

- 3 Edit the following properties as needed:
 - **Name** to specify the name of the assertion.
 - **Condition** to specify the assertion expression. It can be any Boolean expression (*i.e.*, composed of variable/constant references, textual expressions and/or operator calls). Default value is `true`.

Tip

To edit the assertion value, select the element in Diagram View and press F2 or click twice on the element to highlight the value.

- **Assume** or **Guarantee** to toggle the type of the created assertion.

Editing Textual Expression Properties

The textual expression (*i.e.*, literals) you create in SCAD Suite projects can be configured from **Diagram Views**.

- About layout in diagram, see [“Changing Design Element Layout”](#) on page 482
- About comments, see [“Adding and Editing Comments”](#) on page 483
- About notes, see [“Adding and Editing Notes”](#) on page 484

To edit the declaration properties of a textual expression

- 1 Right-click a textual expression in the **Diagram View** and select *Properties*.
- 2 Select **Declaration** and set properties on the right.

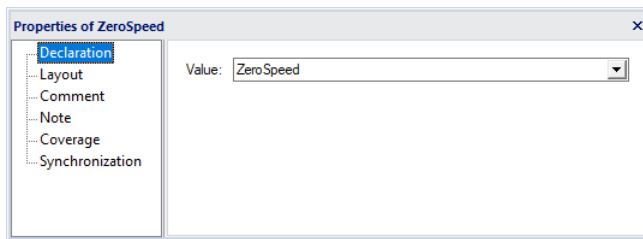


Figure 6.63: Declaration property of textual expressions

- 3 Set the expression **Value** using textual expression from Scade language.

Tips

- To edit the expression value, select the element in the Diagram View and press F2 or click twice on the element to highlight the value.
 - While typing the first letters, enable auto-completion to select from the list of names available in the project (see how [“To complete text automatically”](#) on page 262).
 - Expression value can be a char array value specified as string or character by character. For details, see [“To define char array values”](#) on page 210.
 - Integer or floating literals are typed respectively by the Default literal type option for Integer or Floating (see [“Selecting General Options”](#) on page 77).
-

Changing Design Element Layout

The layout and display of symbols representing elements instantiated in **Diagram Views** can be modified from properties by changing the style associated with them.

- About style management, see [“Creating and Modifying Styles”](#) on page 134

To apply styles to design elements from properties

- 1 In a **Diagram View**, select one or several element symbols.
- 2 Right-click on the selected element and select *Properties*.
- 3 Select **Layout** and set properties on the right.

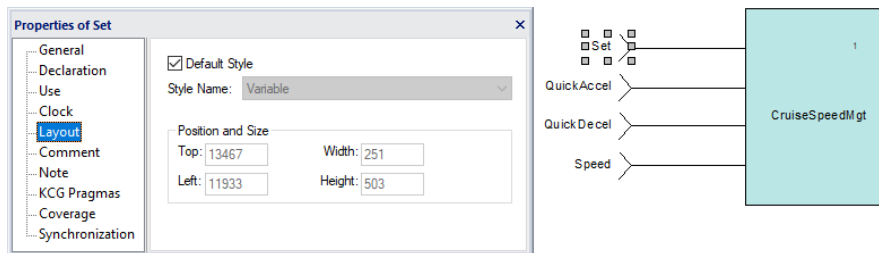


Figure 6.64: Applying styles to design elements

Note

Uneditable fields show the graphical position and size of the element in the diagram.

- 4 Unselect **Default** and set the new style from **Style** name drop-down (see [Table 2.3](#) about predefined styles).
-

Note

When Default is selected, the current style in the drop-down list is defined as the default style for the selected symbol(s). In case of multiselection of symbols with different styles, the list of styles is set as <unchanged>.

Adding and Editing Comments

Most of the elements you define in SCADE Suite projects can have comments associated with. Depending on selected objects, the comments are associated either with the declaration (object selected in Scade View) or with the instance (object selected in diagrams). Comments for C code are saved as KCG pragmas. For details, see [“doc Pragma”](#) on page 244 in *SCADE Suite Technical Manual*. You can freely type the text of the comments in a dedicated field and may decide to propagate this text in the generated code as described below. A tooltip allows to read comments on object declaration directly from **Scade View**.

To add and edit comments

- 1 Right-click an element in **Scade View** or **Diagram Views** and select *Properties*.
- 2 Select **Comment** and set properties on the right.

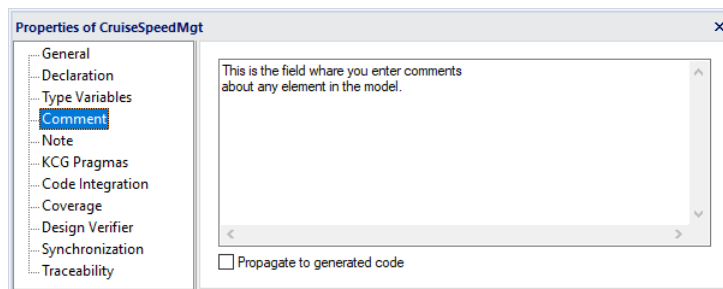


Figure 6.65: Comment properties of design elements

- 3 Type the comment in the free text field.
- 4 Check **Propagate to generated code** to propagate the comment in the generated code using a doc pragma. If generating C code, the comment propagated with the doc pragma cannot contain the `*/` character sequence.

Adding and Editing Notes

The elements defined in SCAD Suite projects can have annotations associated with. The type of annotations varies whether selecting an object declaration (in Scade View), or an object instance (in diagrams). You can type and edit the text of annotations in a dedicated dialog and may decide to propagate this text in the generated code. Design elements have different types of annotation allowed. For details, consult [“Inlayable Annotations Predefined Attributes”](#) on page 29 in *SCADE Suite Technical Manual*.

Note

The allowed types of notes are defined in ATY files, see [“Annotation Type Files”](#) on page 63. At least one ATY file must be specified in the project properties to be able to define new notes, see [“To edit the SCAD Suite properties of a project”](#) on page 426.

- [“To define new notes from Properties”](#)
- [“To edit notes from Properties”](#)
- [“To edit note text directly in Diagram Views”](#)

To define new notes from Properties

- 1 Right-click an element in **Scade View** or **Diagram Views** and select *Properties*.
- 2 Select **Note** and set properties on the right.

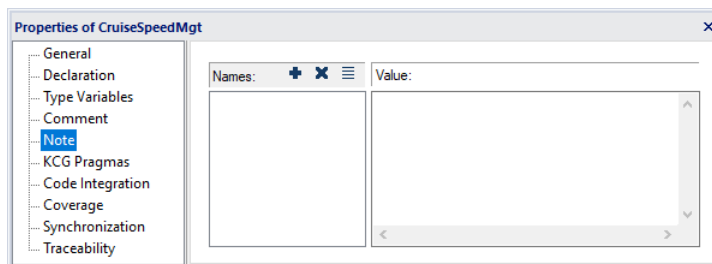


Figure 6.66: Adding and editing notes from Note Properties

- 3 Click **New**  to open the **New Note** window. Depending on the element, different windows can open and different note types are available. Additional notes may be available for model synchronization between SCADE Suite and SCADE Architect as detailed in [“Tuning Translation to Architect by Annotations”](#) on page 456 of *SCADE Architect User Manual*.
- 4 Specify the type of the new note in **Note types** drop-down list:

Remark	to annotate elements in SCADE View with general information
GdC	to annotate elements in SCADE View with creation information (author, creation date, modification date and version)
DesignElement	to annotate elements in SCADE View with methodological level of design information (<i>i.e.</i> , LLR, derived LLR, architecture)
File	to annotate elements in SCADE View with an attached file
Title	to annotate diagrams with title within text box
Author	to annotate diagrams with author information within text box
Date	to annotate diagrams with date within text box
Version	to annotate diagrams with version information within text box
TextInFrame	to annotate diagrams with floating notes in text box
Graphical	to annotate diagrams and elements in diagrams with floating notes
HiddenGraphical	to annotate diagrams and elements in diagrams with hidden floating notes (tooltip text)
TextFile	to annotate diagrams and elements in diagrams with content from text file
ImageFile	to annotate diagrams and elements in diagrams with image file (BMP, EMF, or JPEG)

All annotations inserted on diagrams are movable and resizable.

- 5 Specify **Note name** with a unique name for the selected element.
- 6 Specify information required by the selected type of note.
- 7 Check **Propagate to generated code** to propagate the note (if available).

To edit note text directly in Diagram Views

- 1 Select a note displayed in a **Diagram View** and press F2.

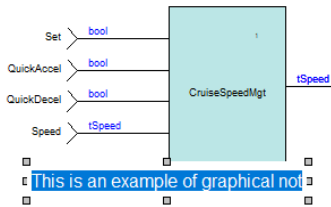


Figure 6.67: Editing note in Diagram Views

- 2 Edit the text and click outside the note to validate.

To edit notes from Properties

- 1 Select an element in **Scade View** or in diagrams, and display **Note** in the **Properties**. Read the list of notes associated with the element and sorted by type. The **Value** area shows descriptive information about the note selected in the list.

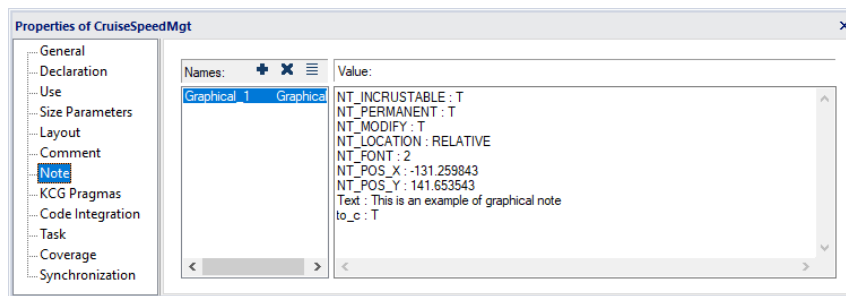


Figure 6.68: Note properties of design elements

- 2 Select the note to edit from the list and edit it as follows:
 - Click **Edit** ≡ to open the **Edit Note** window and modify the annotation. Switch between the defined notes of the selected element from the **Edit Note** window by selecting names in the **Note name** drop-down list.
 - Click **Delete** ✕ to remove the note definition from the properties.

Editing Instance Names

When you insert operators in **Diagram Views**, SCADE Suite Editor automatically computes an instance name based on incremental numbers and assign it to the operator call properties (see [“Editing Operator Call Properties”](#) on page 448). This allows to characterize any library or model operator call by an instance name which is unique for each operator in the current model. SCADE Suite displays instance names for user-defined and predefined operator instances in **Diagram Views**.

Warning

It is always possible to change instance names or remove the instance name automatically assigned to an operator call. Overloading instance names may lead to further collision if already existing. It is recommended to apply instance naming rules that are different from the automatic computation: for example, by setting a prefix and avoiding numbers.

On predefined operator’s symbol the instance name is at the top right of the symbol. If there is little or no room at this place, the instance name is at the bottom right of the symbol. For symmetrical symbols, the instance name is often moved to avoid overlap with parameters.

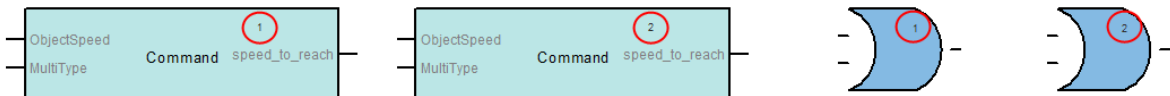


Figure 6.69: Display of instance names in Diagram Views

The style and location of instance names in the graphical symbols of user-defined operators can be edited using Symbol Editor. For user instructions, see [“Editing Design Elements with Symbol Editor”](#) on page 294. Instance names can also be used in textual expressions and diagrams. For details, see [“Managing Instance Names Textually”](#) on page 273.

To edit the instance name of operator calls

- 1 Right-click an operator instance in **Diagram View** and select *Properties*.
- 2 Select **Use** and set properties on the right.

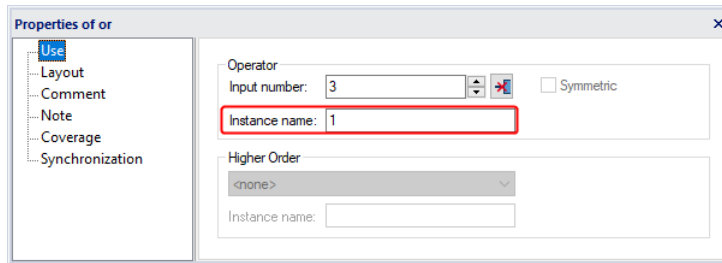


Figure 6.70: Specifying instance names

- 3 Modify the instance name in **Instance name**. You can define one's own instance names using any identifier. You can edit the instance names of user-defined operator calls from the operator symbol on **Diagram Views**.

Accessing Traceability Information

When extending SCADE Suite models with Requirements Management and traceability using SCADE LifeCycle ALM Gateway, you can easily check whether declared model objects are associated with a specific requirement (or test case).

Read user guidelines in *SCADE LifeCycle User Manual*:

- To set traceability via ALM tools: [“Selecting ALM Tool”](#) on page 92
- To check Traceability properties of traceable elements: [“Traceability Information at Element Level”](#) on page 87
- To navigate from traceability data in projects to requirements: [“Navigating between SCADE and Requirements”](#) on page 133

7 / Computing Metrics and Checking Rules on Model

This chapter introduces SCADE Suite Metrics and Rules Checker and explains how to compute metrics about SCADE Suite models and how to check models with design or methodological rules. Users have access to a set of predefined metrics and rules, but can also define their own metrics and rules in Python.

- [“Metrics and Rules Checker Overview”](#)
- [“Setting Metrics and Rules Checker”](#)
- [“Computing Metrics and Checking Rules”](#)
- [“Creating User-Defined Metrics and Rules”](#)

Metrics and Rules Checker Overview

This general overview presents the basic concepts and terminology that support metrics computing and rule checking on SCADE Suite models.

Purpose of Metrics and Rules Checker

SCADE Suite Metrics and Rules Checker consists of the following capabilities:

- *SCADE Suite Metrics Checker* computes metrics on selected parts of the model and generates the results in XML format.
- *SCADE Suite Rules Checker* checks methodological rules on selected parts of the model and generates an HTML report.

A set of metrics and rules are available and can be used as example to define one's own metrics and rules.

Metrics Computation and Rules Checker Fundamentals

A metric gives a number, such as the number of inputs for an operator, or the depth of nested state for a state machine. The tool produces the metrics for each object it applies to, and the result can be used for complexity analyses of the model.

A rule describes a constraint that the model should follow. Each rule has a severity, and can be customized using a parameter for some of them. The tool checks whether each selected rule is respected or violated for each object it applies to. You may end up with certain violations, that you can address by fixing the model or by adding justifications in specific situations.

The possible severities for a rule are:

- **Mandatory (M)**: the rule must be valid and cannot be justified.
- **Required (R)**: the rule must be valid and can be justified.
- **Advisory (A)**: the rule should be valid and can be justified, but it is not mandatory.

The rule may have a parameter. In this case, the check is performed based on the value of this parameter.

Metrics and rules are defined using Python scripts. They should be defined with respect to a dedicated interface and used in the tool as Python metrics and rules files.

LIST OF PREDEFINED METRICS

SCADE Suite Metrics Checker comes with the following set of predefined metrics. They are listed in categories indicating the scope of computation.

Table 7.1: Predefined metrics available for computation

Category	Metric ID	Metric Name	Metric Description
Model	SCADE_M001	Total number of operators	Compute the overall number of operators defined in the model
	SCADE_M002	Total number of types	Compute the overall number of types defined in the model
	SCADE_M003	Total number of constants	Compute the overall number of constants defined in the model
	SCADE_M004	Total number of sensors	Compute the overall number of sensors defined in the model
	SCADE_M005	Total number of packages	Compute the overall number of packages defined in the model
	SCADE_M006	Total number of diagrams	Compute the overall number of diagrams defined in the model
Package	SCADE_M007	Number of operators	Compute the number of operators defined in this package
Operator	SCADE_M008	Number of inputs	Compute the number of operator's inputs
	SCADE_M009	Number of outputs	Compute the number of operator's outputs
	SCADE_M010	Total number of probes	Compute the number of probes defined in this operator including inner scopes
	SCADE_M011	Total number of local variables	Compute the number of local variables defined in this operator including inner scopes
	SCADE_M012	Total number of signals	Compute the number of signals defined in this operator including inner scopes
	SCADE_M013	Total number of user operator instances	Compute the number of user operators instances in the operator and in included scopes

Table 7.1: Predefined metrics available for computation (Continued)

Category	Metric ID	Metric Name	Metric Description
StateMachine	SCADE_M014	Number of states	Compute the number of states of the state machine
	SCADE_M015	Depth of state machines	Compute the maximum depth of nested state machines
	SCADE_M016	Number of transitions	Compute the number of transitions of the state machine

Advanced users can add their own metrics. For adding a metrics definition file in projects, see [“Managing Project Metrics and Rules”](#) on page 501.

Note

Custom metrics can be defined in a Python file or package installed in SCADE Python environment. Consult respectively [“Defining Python Metrics”](#) on page 510 or [“Python Packages for Metrics and Rules”](#) on page 155 in *SCADE Python API Guide*.

LIST OF PREDEFINED RULES

SCADE Suite Rules Checker comes with a set of default rules. Several methodological rules are described in *SCADE Suite® Application Software Development Standard* available in installation at SCADE\help\Common Help Resources\Design Standard. Some of these methodological rules can be automatically verified using the default rules described below. They are listed in categories indicating the design scope to be checked.

Table 7.2: Predefined rules available for design check

Category ^a	Rule ID	Rule Name	Rule Description
Clocked Flows (SCS 2.1.0)	SDR-41	No extra clock	when construct and explicit clocks shall not be used in a SCADE Model

Table 7.2: Predefined rules available for design check (Continued)

Category ^a	Rule ID	Rule Name	Rule Description
Naming (SCS 2.5)	SDR-11-12-1	Pascal case name	A name shall be composed of a sequence of words. Each word composing a name shall start with an uppercase letter; the remainder of the word shall consist of lowercase letters and digits (PascalCase convention). Applies to types, operators, diagrams, state machines, states, if blocks, and when blocks.
	SDR-11-12-2	Camel case name	A name shall be composed of a sequence of words. Each word composing a name shall start with an uppercase letter, except the first one that shall start with a lowercase letter; the remainder of the word shall consist of lowercase letters and digits (camelCase convention). Applies to variables, fields, and signals.
	SDR-18	Diagram name	When there are several diagrams in a scope, the name of each diagram shall be characteristic of its functionality. Otherwise, the default name created by the editor shall be updated to the name of its scope.
	SDR-19	Accumulator name (iterated operators)	For the operators iterated with fold constructs ([map]fold[w][i]): - The accumulators are defined by a name prefixed by acc. - The input/output variables of an accumulator shall be discriminated by suffix In for the inputs and Out for the outputs. Parameter: a different naming can be specified for accumulators by setting in and out. When strict is set to true, the rules verifies the names are not used for variables which are not accumulators.
	SDR-20	Exit condition name (iterated operators)	For iterators which use an exit condition (mapw[i], foldw[i], mapfoldw[i]), the output corresponding to the exit condition should be named continue. Parameter: a different naming can be specified for the exit condition by setting 'continue'.

Table 7.2: Predefined rules available for design check (Continued)

Category ^a	Rule ID	Rule Name	Rule Description
Naming (SCS 2.5)	SDR-21	Uppercase name	The name of a constant, size parameter, enumeration value, or polymorphic type shall use uppercase letters, digits, and underscores.
	SDR-22	Package name	Package names shall be short. Parameter: maximum number of characters for package names.
	SDR-24	Local name uniqueness	The names of local variables and signals shall be unique within an operator.
	SDR-51	Index name (iterated operators)	For iterators using an index (map[w]i, fold[w]i, mapfold[w]i), the input corresponding to the index shall be named or prefixed by index. Parameter: a different naming can be specified for the index by setting index.
Data Typing and Data Flow Restructuring (SCS 2.6)	SDR-52	Literal	Literals shall only be used in constant values. Parameter: list of exceptions separated by comma.
	SDR-64	No anonymous type in interface	Only named types shall be used in interfaces of root operator/imported operator. Anonymous types (like int^3) shall be avoided. Parameter: configuration is set with the name of the configuration used to determine root operators. If not set, all operators eligible to be root operators are considered.
	SDR-65	Pragma manifest for types in interface	A manifest pragma shall be used for each type declaration used in the signature of a root operator / imported operator. Parameter: configuration is set with the name of the configuration used to determine root operators. If not set, all operators eligible to be root operators are considered.

Table 7.2: Predefined rules available for design check (Continued)

Category ^a	Rule ID	Rule Name	Rule Description
Readability and Complexity (SCS 2.7)	SDR-30-1	Number of diagrams per element	Number of diagrams defining an operator, a state, or an action. Parameter: maximum number of diagrams.
	SDR-30-2	Number of user operators in diagram	Number of graphical user-operator instances within a single diagram. Parameter: maximum number of user-operator instances.
	SDR-30-3	Number of predefined operators in diagram	Number of graphical primitive operator instances within a diagram. This metric also includes textual equations. Parameter: maximum number of graphical primitive operator instances.
	SDR-30-4	Number of nested state machines	Number of hierarchical levels of nested state machines. Parameter: maximum number of levels.
	SDR-30-5	Number of nested conditional blocks	Number of hierarchical levels of nested conditional blocks ('If Block', 'When Block'). Parameter: maximum number of levels.
	SDR-30-6	Number of outgoing transitions per state	Number of outgoing transitions per state (counting forks as multiple transitions). Parameter: maximum number of outgoing transitions.
	SDR-31	Diagram print out readability	A diagram's content shall always be printable and readable without any ambiguity. All elements of the diagram shall be displayed within a page, given the page format. Parameter: margin contains margins to apply horizontally and vertically, separated by semicolon.

Table 7.2: Predefined rules available for design check (Continued)

Category ^a	Rule ID	Rule Name	Rule Description
Libraries (SCS 2.8)	SDR-32	SCADE libraries	Only SCADE libraries that are under full control of the project (configuration management, verification) shall be used. In particular, designers shall not use the SCADE product installation libraries, as they are delivered as examples. Parameter: if upper_level equals 0, only libraries defined in the project folder or a sub-folder are accepted; if equal to 1, libraries defined in the parent folder of the project folder are accepted. Set the value to the number of folders between the root folder under control and the project folder.
Robustness (SCS 2.9)	SDR-38	Real numbers comparison	Comparison operators (< / > / <= / >= / = / <>) between real numbers shall always use an appropriate justified tolerance. Parameter: list of forbidden operators between real numbers, separated by coma.
	SDR-39	Case default	For any enumerated data, a Scade case construct using such data shall provide a default to catch any abnormal value.

a. Rule categories and IDs use references from *SCADE Suite® Application Software Development Standard*.

Advanced users can add their own rules. For adding a rules definition file in your project, see [“Managing Project Metrics and Rules”](#) on page 501.

Note

Custom rules can be defined in a Python file or package installed in SCADE Python environment. Consult respectively [“Defining Python Rules”](#) on page 516 or [“Python Packages for Metrics and Rules”](#) on page 155 in *SCADE Python API Guide*.

METRICS AND RULES CHECKER TOOLBAR

The **Metrics and Rules Checker** toolbar provides shortcuts to set and start metrics computations and rules checks. For general information on toolbars display, see [“Toolbars”](#) on page 45.

Metrics and Rules Checker	Setting Rules Checker and Metrics: • “Setting Metrics and Rules Checker” on page 498
	Launching Rules Checker and Metrics: • “Computing Metrics and Checking Rules” on page 503

Setting Metrics and Rules Checker

Before launching metrics computation and rules checks, you must set Metrics and Rules Checker. New SCADE Suite projects come with a configuration containing predefined metrics and rules. You can also add your own metrics and rules to the tool.

- [“Setting Active Metrics and Rules”](#)
- [“Managing Project Metrics and Rules”](#)

Setting Active Metrics and Rules

You can select model parts on which selected metrics are computed and selected rules are checked.

To select metrics and rules

- 1 In **Metrics and Rules Checker** toolbar, select **MetricRule** configuration.
- 2 Click **Settings**  to display **Model** settings.

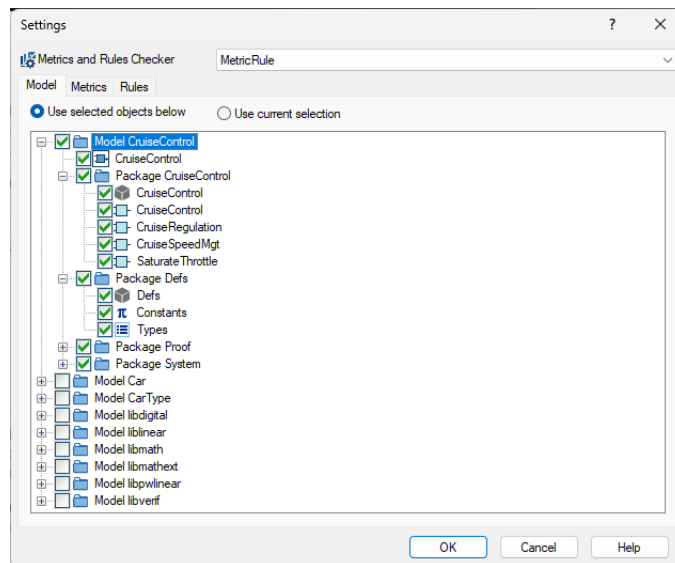


Figure 7.1: Setting model check options

- 3 Set the model elements on which metrics are computed and rules are checked (by default, all model elements are selected, except library ones):
 - Select **Use selected objects below** (default) and select in the tree the parts of the model on which to use metrics/rules. Hold **Ctrl**-key to select separate items one by one in multi-selection or **Shift**-key to set consecutive items in multi-selection, and press **Space** bar to check/uncheck the list highlighted by multi-selection.
 - Select **Use current selection** to perform metrics computation and rule checks on the packages or operators that are selected in **Scade View**.
- 4 Display **Metrics** settings to specify the metrics for computation.

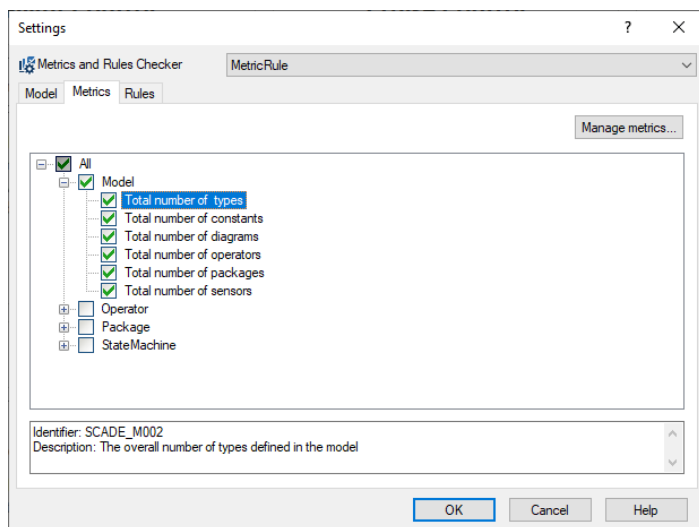


Figure 7.2: Setting active metrics for computation

- 5 Set the list of metrics for computation as follows:
 - Select a metric in the tree to display its information (identifier and description). For the complete list of predefined metrics and their description, see [“List of Predefined Metrics”](#) on page 491.
 - Check a metric or a metric category in the tree to apply for computation.

- Click **Manage metrics** to modify the list of metrics (defined in Python files). For details, see [“Managing Project Metrics and Rules”](#) on page 501.
- 6 Display **Rules** settings to specify the rules for check.

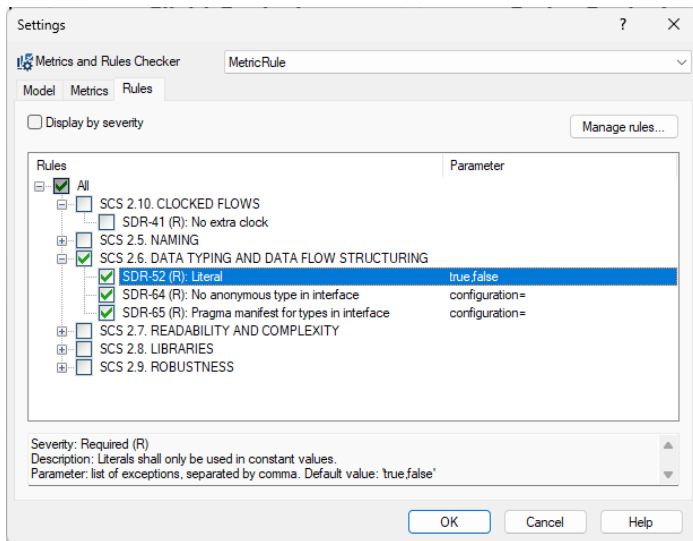


Figure 7.3: Setting active rules for check

- 7 Set the list of rules for check as follows:
- Check **Display by severity** to display the rules by level of severity: Mandatory (M), Required (R), and Advisory (A).
 - Select a rule in the tree to display its information (severity level and description).
 - Check a rule or a rule category in the tree to apply for check.
 - When the selected rule has a parameter, it is possible to edit its value by selecting the corresponding Parameter field.
 - Click **Manage rules** to modify the list of rules (defined in Python files). For details, see [“Managing Project Metrics and Rules”](#) on page 501.

Managing Project Metrics and Rules

SCADE Suite Metrics and Rules Checker comes with a set of predefined metrics and rules defined in Python files. You can specify Python files (.py) or Python packages containing your own metrics and rules definitions.

To set metrics and rules definition files in project

- 1 Select *Project > Metrics and Rules Checker > Manage Metrics and Rules* to open the **Metrics and Rules** dialog.

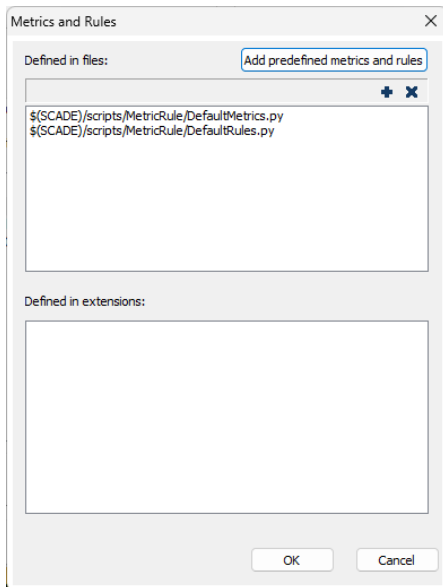


Figure 7.4: Setting user-defined metrics and rules

- 2 Edit the list of files defining the metrics or rules used by the tool:
 - Click **Add predefined metrics and rules** to add the definitions files for predefined metrics and rules.
 - Click **New +** to insert a new metrics or rules file path. For details about writing metrics and rules in Python, see [“Creating User-Defined Metrics and Rules”](#) on page 510.

- Click the browsing button to set the path to a Python file (.py) defining metrics or rules (double-click a file path to display the “browse” button).
 - Select a metrics or rules file in the list and click **Delete** ✕ to remove it from the tool.
- 3 The list of Python packages containing metrics or rules is displayed in **Defined in extensions**. Such Python packages were previously installed in SCADE Python environment using Extension Manager. For instructions, see Chapter 6 about [“Managing SCADE Extensions”](#) in *SCADE Python API Guide*.

Computing Metrics and Checking Rules

Once you have set the metrics to be computed and the rules to be checked, you can launch Metrics and Rules Checker.

- [“Computing Metrics”](#)
- [“Checking Rules”](#)

Computing Metrics

You can launch SCADE Suite Metrics Checker to compute the metrics set in the active configuration.

To launch metrics computation on model

- 1 Select the configuration set for metrics computation from **Metrics and Rules Checker** toolbar.
- 2 Click **Compute Metrics**  to compute the active metrics for the model part set in the active configuration or for the part selected in **Scade View**.

If error occurs, no results are generated as indicated in the **Messages** tab. When computation is successful, *End of Metrics Computation* is displayed in the **Messages** tab with the path to computed metrics contained in a file generated in the project directory (<ProjectName>_metrics.xml).

- 3 Open the generated XML file in Microsoft Excel, or in any text or XML editor to analyze the computed metrics.

	A	B	C	D	E	F	G	H	I	J	K	L
1	proje	configurat	dat	catego	metric	metric_name	description	mir	avera	ma	path	valu
2	C:\Userst	MetricRule	####	Package	PACKAGE_	Number of operators	The number of operators define	4	4	4	CruiseControl:	4
3	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of inputs	The number of operator's inputs	1	4	9	CruiseControl:CruiseSpee	4
4	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of inputs	The number of operator's inputs	1	4	9	CruiseControl:CruiseCont	9
5	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of inputs	The number of operator's inputs	1	4	9	CruiseControl:CruiseRegu	2
6	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of inputs	The number of operator's inputs	1	4	9	CruiseControl:SaturateThr	1
7	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of outputs	The number of operator's outpu	1	1	3	CruiseControl:CruiseSpee	1
8	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of outputs	The number of operator's outpu	1	1	3	CruiseControl:CruiseContr	3
9	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of outputs	The number of operator's outpu	1	1	3	CruiseControl:CruiseRegu	1
10	C:\Userst	MetricRule	####	Operator	OPERATOF	Number of outputs	The number of operator's outpu	1	1	3	CruiseControl:SaturateThr	2

Figure 7.5: Computed model metrics example displayed in Microsoft Excel

```

<?xml version="1.0" encoding="UTF-8"?>
<metricsResults project="C:\Users\Public\Documents\ANSYS Inc\193\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.g3x" configuration="MetricRule" date="2018-10-15T10:48:52">
  <categories>
    <category category_name="Package">
      <metrics>
        <metric metric_id="PACKAGE_001" metric_name="Number of operators" description="The number of operators defined in this package" min="4" average="4" max="4">
          <objects>
            <object path="CruiseControl::" value="4"/>
          </objects>
        </metric>
      </metrics>
    </category>
    <category category_name="Operator">
      <metrics>
        <metric metric_id="OPERATOR_001" metric_name="Number of inputs" description="The number of operator's inputs" min="1" average="4" max="9">
          <objects>
            <object path="CruiseControl::CruiseSpeed@gt/" value="4"/>
            <object path="CruiseControl::CruiseControl/" value="9"/>
            <object path="CruiseControl::CruiseRegulation/" value="2"/>
            <object path="CruiseControl::SaturateThrottle/" value="1"/>
          </objects>
        </metric>
        <metric metric_id="OPERATOR_002" metric_name="Number of outputs" description="The number of operator's outputs" min="1" average="1" max="3">
          <objects>
            <object path="CruiseControl::CruiseSpeed@gt/" value="1"/>
            <object path="CruiseControl::CruiseControl/" value="3"/>
            <object path="CruiseControl::CruiseRegulation/" value="1"/>
            <object path="CruiseControl::SaturateThrottle/" value="2"/>
          </objects>
        </metric>
      </metrics>
    </category>
  </categories>
</metricsResults>

```

Figure 7.6: Computed model metrics example displayed in a text editor

The generated file gives the result of each computed metric for each part of the model on which this metric was computed (**value** column). It also gives the minimum, maximum, and average results for all the computations of that metric in the current session.

Checking Rules

You can launch SCADE Suite Rules Checker to check the rules set in the active configuration. A report is automatically generated with all rule check results.

- [“Launching rules check on model”](#)
- [“Defining justifications for rule violations”](#)

Launching rules check on model

To launch rules check on model

- 1 Select the configuration set for rules checks from **Metrics and Rules Checker** toolbar.
- 2 Click **Check Rules**  to check the active rules for the parts of the model set in the active configuration or for the current selection in **Scade View**.

After a successful check, a report (<ProjectName>_rules.htm) with information about rule check results, applicable rules, and justifications is generated in the project directory and opens automatically. Detected rule violations or previously defined justification for rule violation are reported in the **Info Log** tab of the **Output** window showing error message with rule code and severity as (M) Mandatory, (R) Required, or (A) Advisory.

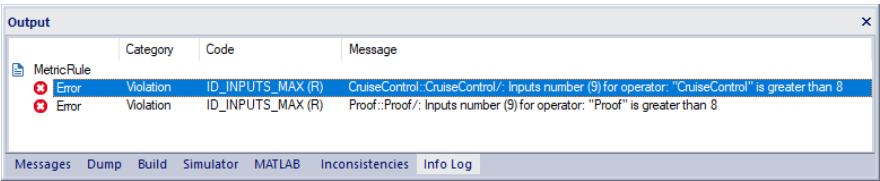


Figure 7.7: Info Log tab displaying violations

Double click reported errors to locate in SCADE Suite model the elements involved in the rule violation or manage justifications for rule violations as detailed in [“Defining justifications for rule violations”](#) on page 507.

The report provides various details about rule check results and allows to:

- Navigate to the definition of each violated rule.
- Locate in SCADE Suite models any element involved in rule violation.

Monday May 02 2022 14:24:15

Rules Checker

CruiseControl

Overall Results: 0 violated rule - 0 violation - 0 justification

1. Model

Model path:
C:\Users\Public\Documents\ANSYS Inc\v231\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp

Selected objects:
CruiseControl

2. Results synthesis

Rules	Checked	Failed	Passed
Mandatory rules	0	0	0
Required rules	0	0	0
Advisory rules	1	0	1

3. Overall rules checker summary

3.1. Mandatory rules summary

No Mandatory rules

3.2. Required rules summary

No Required rules

3.3. Advisory rules summary

0 violation detected - 0 justification

Category	Identifier	Synopsis	Violations	Justifications
Model	SCADE_R001	not coverable points detection	0	0

4. Rules violations

No violation detected.

5. Justifications

No justification.

6. Selected rules definitions

6.1. Mandatory rules definitions

No rule selected.

6.2. Required rules definitions

No rule selected.

6.3. Advisory rules definitions

Category	Identifier	Synopsis	Description	Parameter
Model	SCADE_R001	Not coverable points detection	Detect not coverable points for the model with the configuration given as parameter	Coverage

End of document.

Figure 7.8: Example of SCADE Suite Rules Checker report

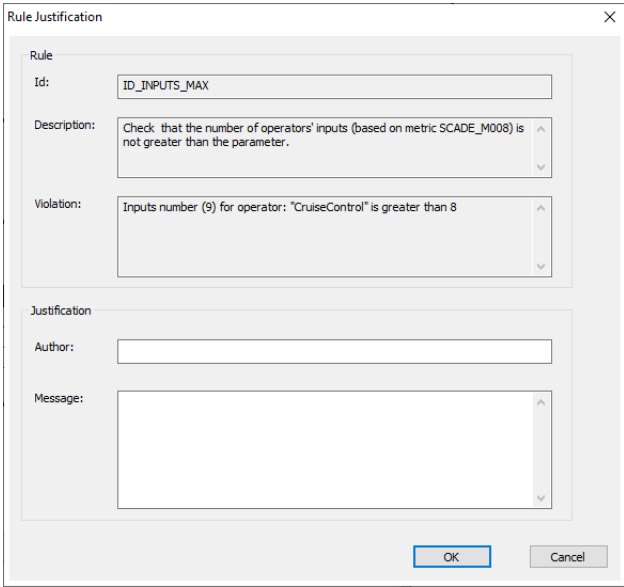
Defining justifications for rule violations

When a violation of a “Required” or “Advisory” rule is reported, you can give a justification for that violation. The justification is taken into account in next rules check. You can add, edit, and remove justifications from the **Output** window or have a full view of model justifications and update them by opening the justification file. Justifications are supported only for violations reported on elements that can be annotated. For a list of elements, see [“Annotable objects”](#) on page 6 in *SCADE Suite Technical Manual*.

- [“To add a justification”](#)
- [“To remove a justification”](#)
- [“To edit a justification”](#)
- [“To manage justifications from justifications file”](#)

To add a justification

- 1 In **Info Log** tab of **Output** window, right-click a violation of a “Required” or “Advisory” rule (reported as “error”) and select **Add justification**.



The screenshot shows a dialog box titled "Rule Justification" with a close button (X) in the top right corner. The dialog is divided into two main sections: "Rule" and "Justification".

Rule Section:

- Id:** A text field containing "ID_INPUTS_MAX".
- Description:** A text area containing "Check: that the number of operators' inputs (based on metric SCADE_M008) is not greater than the parameter." with up and down arrow icons on the right.
- Violation:** A text area containing "Inputs number (9) for operator: 'CruiseControl' is greater than 8" with up and down arrow icons on the right.

Justification Section:

- Author:** An empty text field.
- Message:** An empty text area with up and down arrow icons on the right.

At the bottom of the dialog, there are two buttons: "OK" and "Cancel".

Figure 7.9: Adding rule justification

- 2 In **Rule Justification** window, enter author and justification text as needed and validate. The modification is taken into account in next rules check. It is also possible to create justifications directly in the justification file, see [“To manage justifications from justifications file”](#) on page 509.

To edit a justification

- 1 In **Info Log** tab of **Output** window, right-click a rule justification (reported as “Information”) and select **Edit justification**.

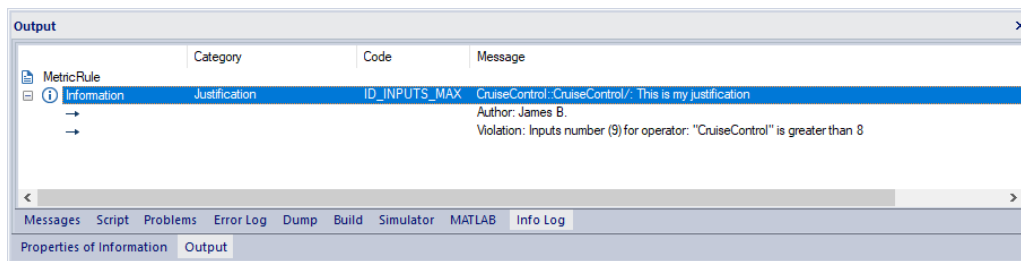


Figure 7.10: Info Log tab displaying justification

- 2 In **Rule Justification** window, edit the author and justification message for the rule violation and validate. The modification is taken into account in next rules check.

Note

It is also possible to update justifications directly in the justification file, see [“To manage justifications from justifications file”](#) on page 509.

To remove a justification

- In **Info Log** tab, right-click a Justification (reported as “Information”) or an Invalid Justification (reported as “Error”) and select **Remove justification**.

To manage justifications from justifications file

- 1 Click **Edit Justifications**  in **Metrics and Rules Checker** toolbar to the open justifications file located in the project directory.

The following example describes justifications for rule violations:

```
{
  "justifications": [
    {
      "rule": "ID_INPUTS_MAX",
      "object_oid": "!ed/4A7A/59EE/B28/46A5B3D7/365",
      "object_path": "CruiseControl::CruiseControl/",
      "violation": "CruiseControl::CruiseControl/ VIOLATION",
      "author": "MV",
      "text": "Justification message on failed REQUIRED for CruiseControl::CruiseControl/"
    },
    {
      "rule": "ID_INPUTS_MAX",
      "object_oid": "win_1E8B/5096/6A4/3C3EC006/17B9",
      "object_path": "Proof::Proof/",
      "violation": "Inputs number (9) for operator: \"Proof\" is greater than 8",
      "author": "MV",
      "text": "Justification message on failed ADVISORY for CruiseControl::CruiseControl/"
    },
    {
      "rule": "ID_T00_MANY_STATES",
      "object_oid": "!ed/4AA0/59EE/B28/46A5B3D7/212A",
      "object_path": "CruiseControl::CruiseControl/SM1:Enabled:SM2:Active:SM3:",
      "violation": "More than \"1\" states in the state machine",
      "author": "MV",
      "text": "Justification message on failed ADVISORY for CruiseControl::CruiseControl/"
    }
  ]
}
```

- 2 Update the justifications and save the file. The modifications are taken into account in next rules check.

Creating User-Defined Metrics and Rules

This section explains how to specify additional Python metrics and rules in SCADE Suite Metrics and Rules Checker.

- [“Defining Python Metrics”](#)
- [“Defining Python Rules”](#)

Defining Python Metrics

Python metrics can be interpreted by SCADE Suite Metrics Checker if they respect the file structure and metrics format described below.

You can use as model the metrics from the `DefaultMetrics.py` Python file located in SCADE installation at `SCADE/scripts/MetricRule`.

- [“Python Metric File Structure”](#)
- [“Metric Example”](#)
- [“Metric Class Methods”](#)

Python Metric File Structure

A metric is defined as an instance of the Metric class. It is defined by:

- an *identifier* which should be a mnemonic tag like `METRIC_001`
- a *label* which should provide a short description used in the metrics tree view
- a *category* which corresponds to a folder in the metrics tree view of the settings

The metrics are applied to a list of Scade model items when computation is done.

Metric Class Methods

The `Metric` base class provides several methods to initiate the metrics and to perform computation. Any instantiation of the `Metric` class or of a derived class registers the metric to the SCADE Suite Metrics Checker tool. The tool implements the following algorithm:

- 1 When the tool is launched, `on_start` is called, then the tool starts browsing the model tree.
- 2 When an object matches the selected metric(s), `before_computing_subtree` is called for this object, then the tool browses the object's content, then `on_compute` is called for this object. The tool applies this behavior for each object matching the selected metric(s) while browsing the content.
- 3 When all metrics are computed, `on_stop` is called.

When a metric is computed, its result can be:

- `Metric.OK`: the metric is computed and the result is available (see [“set_result_metric”](#) on page 515)
- `Metric.NO_METRIC`: no computed result for a particular item (see [“set_message”](#) on page 514)
- `Metric.ERROR`: an internal error occurs ([“set_message”](#) on page 514)

The `Metric` base class methods are:

- [“init”](#)
- [“on_compute”](#)
- [“before_computing_subtree”](#)
- [“on_start”](#)
- [“on_stop”](#)
- [“set_message”](#)
- [“set_result_metric”](#)

INIT

The `init` method of the `Metric` base class is the constructor for the class. It is used to initialize the new metric. The method structure is:

```
def __init__(self, id: str,
             label: str,
             category: str,
             types: List['Object'],
             description: Optional[str] = '',
             on_compute: Callable[['Object'], int] = None,
             before_computing_subtree: Callable[['Object'], int] = None,
             on_start: Callable[[], int] = None,
             on_stop: Callable[[], int] = None
            )
```

where

- *id* is the unique identifier of the metric
- *label* is the name of the metric
- *category* is the name of the category to which the metric belongs
- *types* is a list of SCADE Suite classes: the metric is computed for each object which is an instance of those classes, or of one of their derived classes
- *description* is an optional text describing the metric (if this parameter is empty, the description must be set to the docstring of the class)
- *on_compute** is an optional parameter called instead the [“on_compute”](#) method if specified
- *before_computing_subtree** is an optional parameter called instead the [“before_computing_subtree”](#) method if specified
- *on_start** is an optional parameter called instead the [“on_start”](#) method if specified
- *on_stop** is an optional parameter called instead the [“on_stop”](#) method if specified

Observations: (*) *on_compute*, *before_computing_subtree*, *on_start*, and *on_stop* methods are optional parameters. If they are specified, the methods of the class having the same name are ignored.

A simple metric definition is therefore:

```
class MyMetric(Metric):
    """MYMETRIC_001 computes some metrics on all Scade Objects"""
    def __init__(self):
        Metric.__init__('MYMETRIC_001', 'My metric computation', 'Model', [Model])
        # other methods to be redefined
# Registration
MyMetric()
```

ON_COMPUTE

The `on_compute` method of the `Metric` base class computes the metric. It must be overridden in a derived class. The method structure is:

```
def on_compute(self, object: 'Object') -> int
```

where

- `object` is the object on which the metric is computed

It returns:

- `Metric.OK` if the metric succeeds
- `Metric.NO_METRIC` if the metric is not applicable to the selected object (e.g., some metrics of an operator, such as the number of equations, are not applicable on an imported operator)
- `Metric.ERROR` if an error occurred, leading the whole metrics computation to fail

Observations: This method is not called if the `on_compute` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

BEFORE_COMPUTING_SUBTREE

The `before_computing_subtree` method of the `Metric` base class is called before browsing the object's content and before `on_compute` is called to compute metrics on it. It must be overridden in a derived class. The method has the following structure:

```
def before_computing_subtree(self, object: 'Object') -> int
```

where

- `object` is the object on which the metric is about to be computed

It returns:

- `Metric.OK` if the method succeeds
- `Metric.ERROR` if an error occurred, leading the whole metrics computation to fail

Observations: This method is not called if the `before_computing_subtree` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

ON_START

The `on_start` method of the `Metric` base class is called once before any call to `on_compute` or `before_computing_subtree` is done. It is not called if no object matches the metric. The method structure is:

```
def on_start(self, model: 'Model') -> int
```

where

- `model` is the model on which the metric is computed

It returns:

- `Metric.OK` if the method succeeds
- `Metric.ERROR` if an error occurred, leading the whole metrics computation to fail

Observations: This method is not called if the `on_start` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

ON_STOP

The `on_stop` method of the `Metric` base class is called once all computations are done. It is not called if no object matches the metric. It must also be called after the last call to the `on_compute` method of the metric to clean all variables created during class instantiation. The method structure is:

```
def on_stop(self) -> int
```

It returns:

- `Metric.OK` if the method succeeds
- `Metric.ERROR` if an error occurred, leading the whole metrics computation to fail

Observations: This method is not called if the `on_stop` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

SET_MESSAGE

`set_message` method is called from any method before returning `Metric.ERROR` to specify the error message. The method structure is:

```
def set_message(self, message: str) -> None
```

where

- `message` is the error message

SET_RESULT_METRIC

`set_result_metric` method is called from `on_compute` before returning `Metric.OK` to specify the result metric. The method structure is:

```
def set_result_metric(self, result: int) -> None
```

where

- `result` is the metric computation result

Metric Example

The example below is a metric that computes the number of operator inputs:

```
import os
import scade
from scade.model.suite import Operator, Object
from scade.tool.suite.metrics import Metric
class NbOfInputs(Metric):
    """
    The number of operator's inputs
    """
    def __init__(self):
        Metric.__init__(self, 'OPERATOR_001',
                        'Number of inputs',
                        'Operator',
                        [Operator])
    def on_compute(self, object: 'Object') -> int:
        if (isinstance(object, Operator)):
            self.set_result_metric(len(object.inputs))
            return Metric.OK
        else:
            self.set_message("Bad object")
            return Metric.ERROR
#Instantiation of the metric
NbOfInputs()
```

Defining Python Rules

Python rules can be interpreted by SCADE Suite Rules Checker if they respect the file structure and rule format described below. Rule examples can be found from `Rules_Example.py` file located at `SCADE\APIs\Python\examples\Rules Checker`.

- [“Python Rule File Structure”](#)
- [“Rule Example”](#)
- [“Rule Class Methods”](#)

Python Rule File Structure

A rule is defined as an instance of the Rule class. It is defined by an *identifier* (which should be a mnemonic tag like `RULE_001`), a *label* which should provide a short description (which is used in the rules tree view), a *category* which corresponds to a folder in the rules tree view of the settings, a *severity*, and *types* (list of SCADE classes).

When the check is done, the rule is applied to Scade objects whose class is matching types.

The severity of rules is defined as:

- **Mandatory**: rule to fulfill. Use `Rule.MANDATORY` severity level.
- **Required**: rule to fulfill. In case of failure, the model can be fixed or a justification can be provided. Use `Rule.REQUIRED` severity level.
- **Advisory**: recommended rule. In case of failure, a justification can be given but it is not mandatory. Use `Rule.ADVISORY` severity level.

A rule can have a *parameter*. Such parameter is a simple string, and it is the responsibility of the check methods to handle it, according to what is expected by the given rule.

Rule Class Methods

The `Rule` class provides methods to initiate rules and perform checks. Any instantiation of the `Rule` class or of a derived class registers the rule to the SCADE Suite Rules Checker tool implementing the following algorithm:

- 1 When the tool is launched, `on_start` is called, then the tool starts browsing the model tree.
- 2 When an object matches the selected rule's types:
 - `before_checking_subtree` is called for this object
 - then the tool browses recursively the object's sub-tree
 - then `on_check` is called for this object
 - then `after_checking_subtree` is called for this object

The tool applies this behavior for each object matching the selected rule while browsing the content.

- 3 When all rules are checked, `on_stop` is called.

When a rule is checked, its result can be:

- `Rule.OK`: the check is passed
- `Rule.FAILED`: the check fails, additional information is provided (see [“set_message”](#) on page 521)
- `Rule.ERROR`: an internal error occurs (see [“set_message”](#) on page 521)
- `Rule.NO_RULE`: the rule does not apply to the specific object

To define each of the following methods (`on_check`, `before_checking_subtree`, `after_checking_subtree`, `on_start`, and `on_stop`), there are two strategies:

- giving a callable as parameter of the `Rule` class constructor (`init`)
- overriding the method in a derived class

The `Rule` class methods are:

- [“init”](#)
- [“on_check”](#)
- [“before_checking_subtree”](#)
- [“after_checking_subtree”](#)
- [“on_start”](#)
- [“on_stop”](#)
- [“set_message”](#)
- [“add_rule_status”](#)
- [“get_metric_result”](#)

INIT

The `init` method of the `Rule` base class is the constructor for the class. It is used to initialize the new rule.

The method must be defined as follows:

```
def __init__(self, id: str,
             label: str,
             category: str,
             severity: int,
             types: List['Object'],
             has_parameter: Optional[bool] = False,
             default_param: Optional[str] = '',
             description: Optional[str] = '',
             on_check: Callable[['Object', Optional[str]], int] = None,
             before_checking_subtree: Callable[['Object', Optional[str]], int] = None,
             on_start: Callable[[Optional[str], int] = None,
             on_stop: Callable[[], int] = None,
             metric_ids: Optional[List[str]] = [],
             after_checking_subtree: Callable[['Object', Optional[str]], int] = None
            )
```

where

- *id* is the unique identifier of the rule.
- *label* is the name/short description of the rule.
- *category* is the name of the category to which the rule belongs.
- *severity* is an integer describing the severity of the rule: `Rule.MANDATORY`, `Rule.REQUIRED`, or `Rule.ADVISORY`.
- *types* is a list of SCADE Suite classes. The rule is checked for each object which is an instance of one of those classes (or derived classes).
- *has_parameter* is an optional Boolean (True if the rule has a parameter).
- *default_param* is an optional default value of the parameter. This value appears in the rule tree view.
- *description* is an optional text describing the rule. If this parameter is empty, the description must be set to the docstring of the class.
- *on_check** (if specified) is called instead of [“on_check”](#) method.
- *before_checking_subtree** (if specified) is called instead of [“before_checking_subtree”](#) method.
- *on_start** (if specified) is called instead of [“on_start”](#) method.
- *on_stop** (if specified) is called instead of [“on_stop”](#) method.
- *metric_ids* is an optional list of metric identifiers used by the rule.
- *after_checking_subtree** (if specified) is called instead of [“after_checking_subtree”](#) method.

Observations: (*) *on_check*, *before_checking_subtree*, *after_checking_subtree*, *on_start*, and *on_stop*, are optional parameters. If they are specified, the methods of the class having the same name are ignored.

A simple rule definition is for instance:

```
class MyRule(RULE):
    """SRS_001 checks some property on Scade Objects"""
    def __init__(self):
        Rule.__init__('MYMETRIC_001',
                      "My specific check",
                      "category",
                      Rule.ADVISORY)
    # other methods to be redefined
# Registration
MyRule()
```

ON_CHECK

The `on_check` method of the `Rule` base class is used to check whether the rule succeeds or not. It is called when the object matches the rule's types.

The method must be defined as follows:

```
def on_check(self, object: 'Object', parameter: str = None) -> int
```

where

- `object` is the object on which the rule is checked
- `parameter` is the parameter value in case the rule has a parameter, else it must be ignored

It returns:

- `Rule.OK` if the rule succeeds
- `Rule.FAILED` if the rule fails
- `Rule.ERROR` if an error occurred - leading the whole rules checker computation to fail
- `Rule.NO_RULE` if the rule does not apply to the object

Observation: This method is not called if the `on_check` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

BEFORE_CHECKING_SUBTREE

The `before_checking_subtree` method of the `Rule` base class is called before browsing object sub-tree according to the algorithm described previously. It is called when the object matches the rule's types. The method must be defined as follows:

```
def before_checking_subtree(self, object: 'Object', parameter: str = None) -> int
where
```

- `object` is the object on which the rule is checked
- `parameter` is the parameter value in case the rule has a parameter, else it must be ignored

It returns:

- `Rule.OK` if the method succeeds
- `Rule.ERROR` if an error occurred, leading the whole Rules checker computation to fail

Observations: This method is not called if the *before_checking_subtree* optional parameter is specified when calling the *init* method. In such a case, the callable set as parameter is called instead.

AFTER_CHECKING_SUBTREE

The `after_checking_subtree` method is called after browsing subtree and after `on_check` call on the object according to the algorithm described previously. It can be used to clean all variables created during `before_checking_subtree` call. It is called when the object matches the rule's types.

The method must be defined as follows:

```
def after_checking_subtree(self, object: 'Object', parameter: str = None) -> int
where
```

- `object` is the object on which the rule is checked
- `parameter` is the parameter value in case the rule has a parameter

It returns:

- `Rule.OK`
- `Rule.ERROR` if an error occurred, leading the whole Rules checker computation to fail

Observations: This method is not called if the *after_checking_subtree* optional parameter is specified when calling the *init* method. In such a case, the callable set as parameter is called instead.

ON_START

The `on_start` method of the `Rule` base class is called once before any check is done. It is not called if no object matches the rule's types. It can be used to make some initialization in the class instantiation.

The method must be defined as follows:

```
def on_start(self, model: 'Model', parameter: str = None) -> int
```

where

- `model` is the model on which the rules are checked
- `parameter` is the parameter value in case the rule has a parameter, else it must be ignored

It returns:

- `Rule.OK` if the method succeeds
- `Rule.ERROR` if an error occurred, leading the whole Rules checker computation to fail

Observations: This method is not called if the `on_start` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

ON_STOP

The `on_stop` method of the `Rule` base class is called once all checks have been done. It is not called if no object matches the rule's types. It can be used to clean all variables created during class instantiation.

The method must be defined as follows:

```
def on_stop(self) -> int
```

It returns:

- `Rule.OK` if the method succeeds
- `Rule.ERROR` if an error occurred, leading the whole rules checker computation to fail

Observations: This method is not called if the `on_stop` optional parameter is specified when calling the `init` method. In such a case, the callable set as parameter is called instead.

SET_MESSAGE

The `set_message` method is called from the following methods:

- `on_check` before returning `Rule.FAILED` to specify the failure message

- any method before returning `Rule.ERROR` to specify the error message

The method must be defined as follows:

```
def set_message(self, message: str) -> None
```

where

- message is the error message

ADD_RULE_STATUS

The `add_rule_status` method is called from any method to set the rule status for any object.

The method must be defined as follows:

```
def add_rule_status (self, object: 'Object', status: int, message: str, [local_id: str]) -> None
```

where

- object is any object of the model.
- status is the status of the rule for this object (`Rule.OK` or `Rule.FAILED`).
- message is the error message when the rule is failed. It can be empty if the rule is successful.
- `local_id` (optional): is the local context. In some specific use cases, a rule could fail several times for the same object. Set this `local_id` in such case only, and make sure the `local_id` value is different for each failure and persistent through sessions, as it is used by justifications if any. It is expected that the failure message contains the failure specificity for clear reporting.

Observation: This method has the same behavior as if `on_check` callback is called for the current rule including justification management.

GET_METRIC_RESULT

The `get_metric_result` method is called to get the result of a metric for the object. The used metric must first be declared in the `metric_ids` rule argument.

The method must be defined as follows:

```
def get_metric_result(self, object: 'Object', metric_id: str) -> int
```

where

- object is the object for which the metric will be get
- `metric_id` is the identifier of the metric

Rule Example

The example below is a rule that checks the length of all names in a model:

```
import os
import scade
from scade.model.suite import Object
from scade.tool.suite.rules import Rule
class LongNameRule(Rule):
    """
    Check the name is not longer than a given number of characters. The max number of character
    must be set as parameter of the rule.
    """
    def __init__(self):
        Rule.__init__(self, 'ID_LONG_NAME',
                        'Name length',
                        'General',
                        Rule.MANDATORY,
                        [Object],
                        True, '10')
    def on_check(self, object: 'Object', parameter: str) -> int:
        if (hasattr(object, 'name')):
            if (len(object.name) > int(parameter)):
                self.set_message('Name: "' + object.name + '" is longer than ' + parameter
+ ' characters')
                return Rule.FAILED
            return Rule.OK
#Instantiation of the rule
LongNameRule()
```

Rules may be defined in several files: `myrules_def.py` containing the rules definition, and `myrules.py` referring to the definition file and performing rules instantiation. The second one is loaded into Rule Checker.

- In `myrules_def.py`, define company rules like:

```
class CpyRule(Rule):
    """ rule description """
    def __init__(self, id):
        Rule.__init__(self, id, "short descr", "category", Rule.MANDATORY).
```

- In `rules.py`:

```
from myrules_def.py import *
CpyRule("SRS_001")
```

This allows redefining rule identifier per projects. Category or severity can be adapted the same way without altering the core of the check.

8 / Change Control of Models

This chapter explains how to manage change control by comparing different SCADE Suite models or objects based on the analysis of semantic differences.

- [“Fundamentals about Model Diff”](#)
- [“Analyzing Model and Object Differences”](#)
- [“Generating Diff Reports”](#)
- [“Merging Differences between Models”](#)

Fundamentals about Model Diff

SCADE Suite allows to analyze semantic differences between two models (including libraries) or between two objects of the same nature, whether they are in different projects or in the same project, and identify explicit modifications made on a model (*i.e.*, information useful for subcontractors or another team). From identified SCADE Suite elements, one can easily locate differences between models/objects and generate reports listing all differences at model or object level.

- [“Semantic Model/Object Differences”](#)
- [“Reports about Model and Object Differences”](#)
- [“Analysis and Merge of Differences”](#)

Model Diff can also be used in command line. For details, see [“Model Diff Command Line”](#) on page 921.

Semantic Model/Object Differences

Diff Analyzer computes the differences between two models or two objects of the same nature. In command line, it is also possible to compare two .xscade files. This computation is semantics-based and discards graphical differences (*i.e.*, due to moving and resizing operations, or symbol modifications). Annotations and comments are taken into account. Diff Analyzer reports the following differences:

- An element is analyzed as *added* when present only in the second model or object.
- An element is analyzed as *deleted* when present only in the first model or object.
- An element is analyzed as *changed* when one of its attributes is modified (name, type, ...), or when one of its children is modified (with respect to the hierarchy tree) – provided these children do not have their own differences explicitly displayed.

Analysis and Merge of Differences

After computation of differences, the **Diff View** opens (in a dedicated Diff perspective), allowing to analyze differences, to locate analyzed differences in models, and to merge differences between models.

Merging allows to remove differences by updating a part of a model with identified content differences from the compared model. It replaces an element (and its content) from one model by a copy of the matching element (and its content) from the other model. Notice that you can merge differences only when comparing models. Merging is only available when comparing two projects.

DIFF VIEW

The **Diff View** is the docking window that displays the element(s) involved in any differences detected by Diff analysis. Both compared hierarchies are represented in side-by-side trees. Selecting an element on one side selects the matching element on the other side. All semantic attributes are listed under each tree for the selected element showing compared models/objects and value differences displayed with the color highlight of each panel side. Parts of the model containing differences are highlighted in pink (or dark pink when the element owns itself a difference). Parts of the model with no differences are non-colored. Whenever selecting compared elements in trees, they can be located in models by double-click or contextual command with respect to the same color highlight from each panel side.

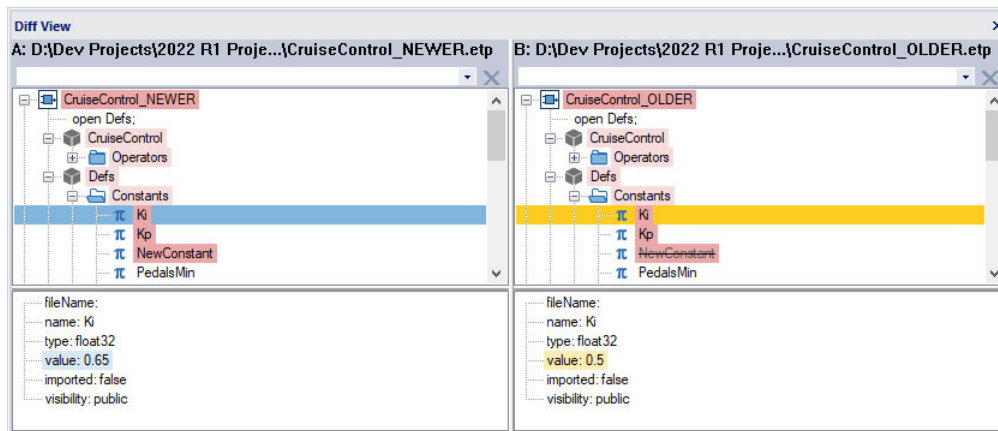


Figure 8.1: Diff View docking window

To show/hide the Diff View window

- Select *View > Docking Windows > Diff View*. Alternatively, right-click in the toolbar area and select *Diff*.

Closing the **Diff View** switches back to the previously used perspective.

Reports about Model and Object Differences

The Diff reports displays a complete list of all differences between both models/objects found by Diff Analyzer with respect to the SCADE hierarchy. As shown below, the report contains links to locate each element in currently loaded models.

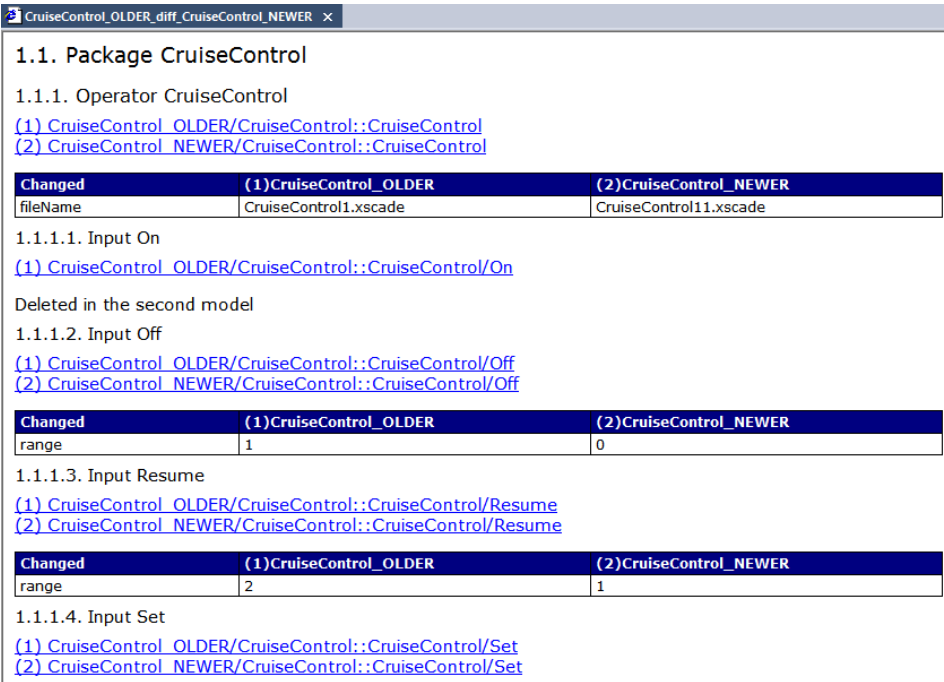


Figure 8.2: Example of Diff report

Note

If you ever need to customize the report data and presentation of the Diff Reporter, please contact SCADE Products Technical Support.

RELATED TOPIC

- [“Generating Diff Reports”](#) on page 536

Analyzing Model and Object Differences

After comparing different versions of SCADE Suite models or different SCADE Suite objects, you can examine semantic differences and locate them in models.

- [“Launching Model Comparison”](#)
- [“Examining Semantic Differences”](#)
- [“Launching Object Comparison”](#)
- [“Locating Differences in Diagrams and Views”](#)

Launching Model Comparison

You can launch the Model Diff analysis either between separate SCADE Suite projects, or between two projects from the same workspace.

To launch model comparison between separate projects

- 1 Load the first project in SCADE Suite.
- 2 Click **Model Diff Analyze**  in **Project** toolbar or select *Project > DiffAnalyzer > Model Diff Analyze* menu to open **Select Projects**.

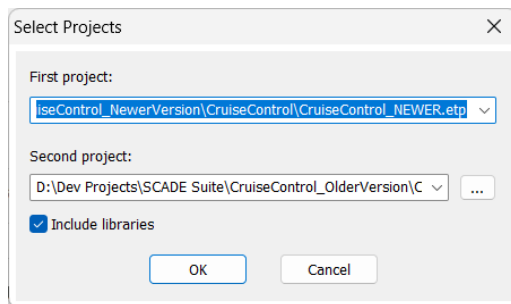


Figure 8.3: Selecting projects to be compared

- 3 Select the first project from **First project** list. The list corresponds to the projects belonging to the currently opened workspace.
- 4 Browse to select the SCADE Suite project (.etp file) to compare to the first project. The selected project path displays in **Second project** list.

- 5 Uncheck **Include libraries** to exclude libraries from comparison.
- 6 Click **OK** to launch Diff analysis on selected projects. Messages display in the **Diff** output tab to indicate whether comparison was successful. The second project is added to the current workspace and set as active project. It remains listed in the **Select Projects** window to facilitate next comparison sessions.

Launching comparison between projects in the same workspace requires to have both SCADE Suite projects contained by the currently opened workspace.

To launch model comparison between projects in the same workspace

- 1 Load the project workspace in SCADE Suite.
- 2 In **Scade View** or **File View**, select the first project (.etp) file.
- 3 Hold Ctr L-key down and select the second project file.
- 4 Click **Model Diff Analyze**  in **Project** toolbar or select *Project > DiffAnalyzer > Model Diff Analyze* menu to launch Diff analysis on selected projects. Messages display in the **DiffAnalyzer** output tab to indicate if the comparison was successful.

Launching Object Comparison

You can launch Diff analysis between two SCADE Suite objects of the same nature like: packages, operators, constants, variables, signals, sensors, named types, state machines, states, if blocks, when blocks, actions of if blocks and when blocks, assertions, and equation sets. Compared objects can be in the same model or in different models.

Note

Object comparison is enabled only when two objects of the same nature are selected. If none or only one object is selected, diff analysis is enabled on models. If more than two are selected or if selected objects are not of the same nature, diff analysis is disabled.

Comparing two objects from different models requires inserting both projects in the same workspace (see [“Inserting Existing Project or Files”](#) on page 112).

To launch comparison between two objects

- 1 Load the project that contains the objects to compare.
- 2 Select the first object to compare in **Scade View** or in **Diagram Views**.
- 3 Hold Ctr L-key down and select the second object in the same project (.etp) or in a different project from the same workspace. Objects must be of the same nature.
- 4 Click **Model Diff Analyze**  in **Project** toolbar or select *Project > DiffAnalyzer > Model Diff Analyze* menu to launch Diff analysis on selected objects. Messages display in the **DiffAnalyzer** output tab to indicate whether comparison was successful.

Examining Semantic Differences

When difference analysis is over, it is possible visualize all differences between compared models/objects in the **Diff View** window.

To examine model and object differences

- 1 In **Diff View**, navigate in the model tree hierarchy to identify and examine differences. The window shows the results from last analysis. Diff View content does not reflect model changes done after analysis. Elements highlighted in light pink indicate they contain child elements with differences. Notice that selecting any tree element automatically selects the corresponding element in the other tree with a different underline.

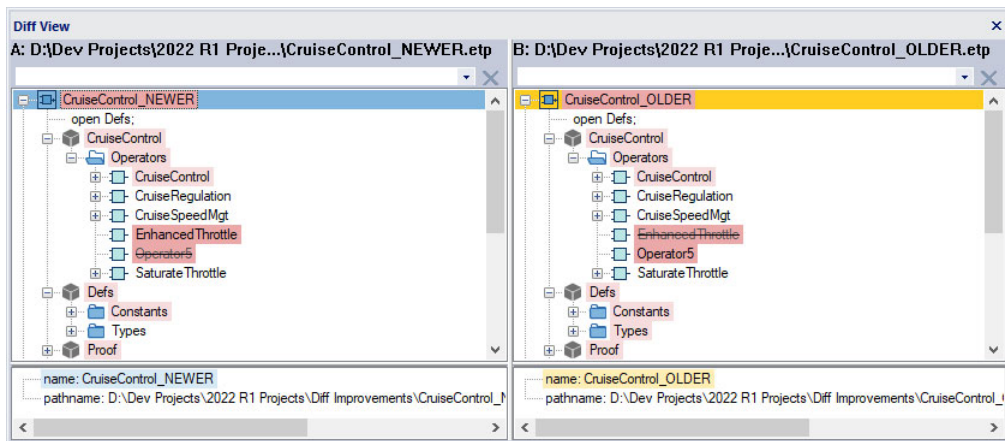
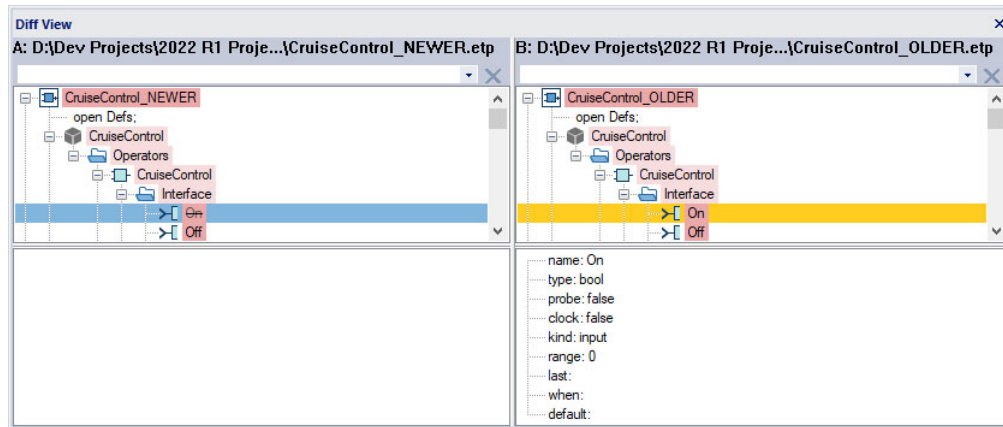


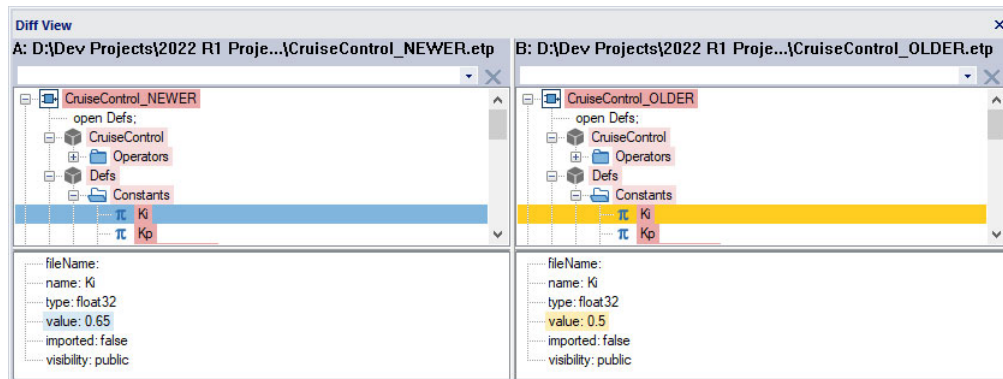
Figure 8.4: Displaying Diff View

- 2 Click **Next Difference**  (F4 key) or **Previous Difference**  (Shift+F4 key) in **Differences** toolbar to navigate between elements with differences (highlighted in dark pink) and locate those differences in models.
- 3 For each element with differences, read information under each tree panel to check for detected differences. Differences on semantic attributes are displayed as follows:

- *Added/removed elements* are highlighted in both trees but their semantic attributes are displayed only under the tree panel of the project where defined (strike-through text in tree where element is not defined).



- *Modified elements* are highlighted in both trees and their semantic attributes are displayed under each tree panel highlighting attribute differences.



- 4 Refresh difference results to reflect model changes since last Diff analysis: right-click anywhere in trees and select *Refresh*.

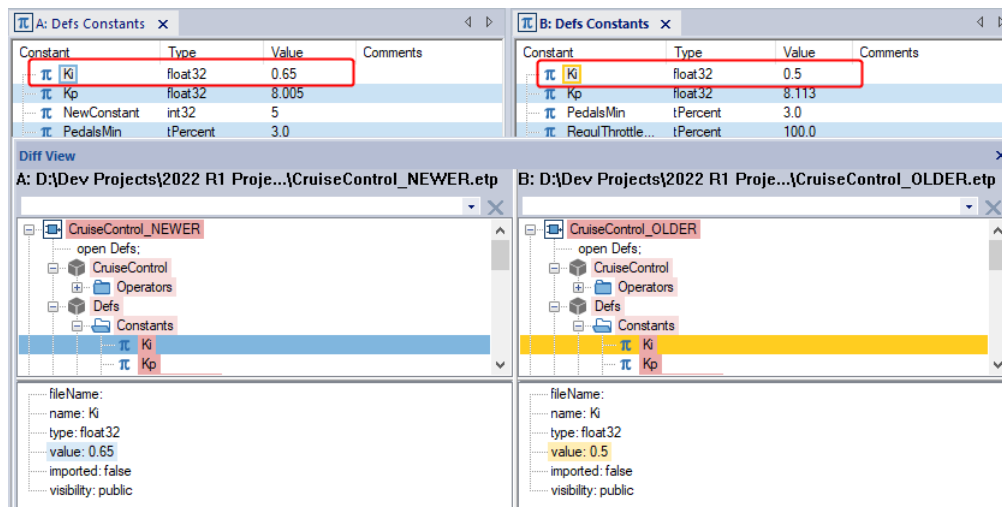
Locating Differences in Diagrams and Views

Model Diff allows to locate in diagrams and model views any difference between compared models/objects directly from the **Diff View**.

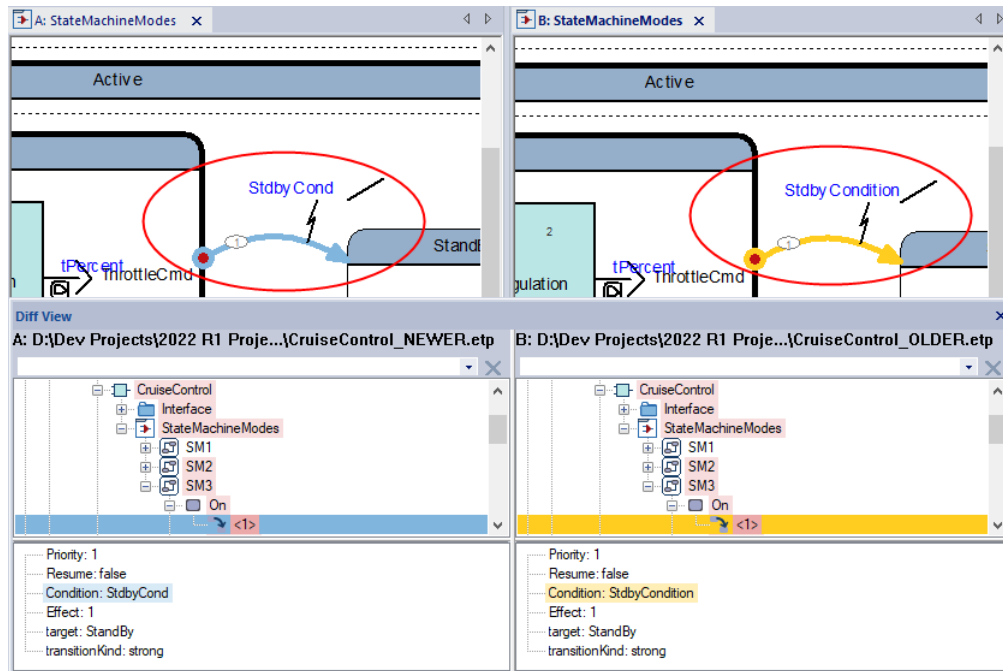
To locate differences from the Diff View

- 1 In **Diff View**, double-click any element in either tree and locate it where defined/instantiated in the model. When possible, the diagrams or views from each project open at the same time to locate element. Located elements are highlighted in the corresponding view(s) of the appropriate project by a frame or line in the same color as the corresponding Diff tree underline.

 - When locating a difference related to the definition of an element present in **Scade View**, the related element is highlighted in **Scade View**.
 - When locating a difference related to the definition of constants, sensors, or types, the related element is highlighted in **Constants View**, **Sensors View**, or in **Types View**.



- When locating a difference related to the instantiation of elements (*i.e.*, equations and diagrams), the related element instance is highlighted in the corresponding **Diagram View**.



Tip

To locate in project tree any instance highlighted in diagrams, use Show in Browser from Navigate toolbar to set the focus on the corresponding element in Scade View.

Generating Diff Reports

After analyzing differences between SCADE Suite models or objects, Diff Reporter allows for producing a complete HTML report which facilitates tracing all differences. Reports are stored in the directory of the first project of the comparison.

To generate Diff reports

- Click **Model Diff Report**  in **Project** toolbar or select *Project > DiffAnalyzer > Model Diff Report* menu to generate a report. Generation completion is displayed in the **DiffAnalyzer** output tab. Generated content relates to the last performed analysis. If no analysis was previously run, a warning indicates that no report can be generated.

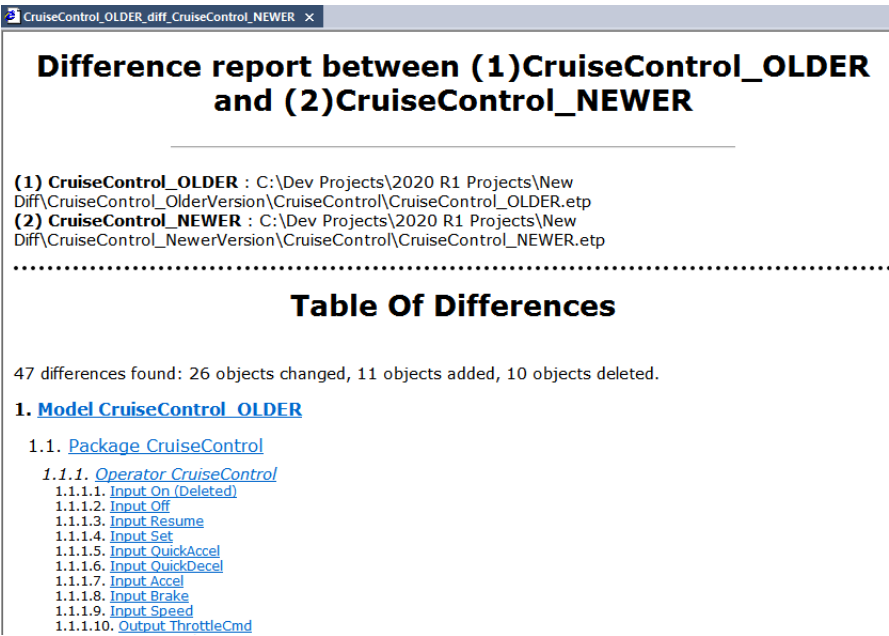


Figure 8.5: Generating a report about model difference analysis

The report starts by a summary showing the total number of differences with details about the number of changed, added, or deleted objects. The rest of the report provides the exhaustive list of all semantic differences found between compared models/objects. It is organized by sections reporting differences found for each construct in SCADE Suite projects (i.e., Constants, Types, Operators, Equation Sets, Libraries, etc.). Each difference is detailed, indicating its exact nature (added or deleted object, different attribute, different parent, different child). A link to the corresponding element in the models is also provided to facilitate the location of differences.

Merging Differences between Models

Model Diff allows to merge any differences found between compared models. It is possible to merge model differences at the level of Packages, Operators, Constants, Types, Sensors, Local variables, Signals, graphical diagrams, or textual diagrams. The differences from the source object and its children are merged to the target object and its children.

To merge differences found between objects

- 1 In **Diff View**, select in one of the tree any object highlighted for differences.
- 2 Right-click the selection and select:
 - *Merge to Right* (on Package, Operator, Constant, Type, Sensor, Local variable, Signal, graphical diagram, or textual diagram): to merge differences from left tree object and its children to right tree corresponding object and its children. Alternatively, click  in **Differences** toolbar.
 - *Merge to Left* (on Package, Operator, Constant, Type, Sensor, Local variable, Signal, graphical diagram, or textual diagram): to merge differences from right tree object and its children to left tree corresponding object and its children. Alternatively, click  in **Differences** toolbar.
 - *Merge Declaration Only to Right* or *Merge Declaration Only to Left* (on package selection only): to merge differences (only package attributes and open clauses) to the corresponding package at selected direction.
 - *Merge Package Declaration to Left* or *Merge Package Declaration to Right* to merge parent package declaration differences (package attributes and open clauses) to the corresponding package at selected direction.

- *Merge Diagram to Left* or *Merge Diagram to Right* to merge differences from parent operator diagram to the corresponding diagram at selected direction.
- *Merge Operator to Left* or *Merge Operator to Right* to merge differences from parent operator to the corresponding operator at selected direction.
- *Merge Constants to Left* or *Merge Constants to Right* (on Constants folders only): to merge all constant differences in the folder at selected direction.
- *Merge Types to Left* or *Merge Types to Right* (on Types folders only): to merge all type differences in the folder at selected direction.
- *Merge Sensors to Left* or *Merge Sensors to Right* (on Sensors folders only): to merge all sensor differences in the folder at selected direction.
- *Merge Locals to Left* or *Merge Locals to Right* (on Locals folders only): to merge all local variable differences in the folder at selected direction.

When merging two packages, if there are any moved children, so that one stays inside the source or target package and the other is outside, a confirmation message requests whether or not to merge the moved children too. Click **Yes** to proceed with a normal merge including the moved children; click **No** to exclude the moved children from merging (no changes on them); or click **Cancel** to stop merging.

9 / Managing Model Refactoring

This chapter explains how issues occurring while refactoring model design can be easily identified and fixed.

- [“Model Refactoring”](#)
- [“List and Visualization of Refactoring Issues”](#)
- [“Applying Quick Fixes to Refactoring Issues”](#)
- [“Applying User Fixes to Inconsistencies”](#)

Model Refactoring

Whenever users edit their project libraries, it is necessary to reconcile library changes with models that reference these libraries. This situation and some other refactoring activities may cause the following issues in models:

- **Pending references:** References to any constant, variable, sensor, type, operator, or package have no associated declaration.
- **Inconsistent operator calls:** The inputs/outputs set in a user operator definition are inconsistent with the operator call(s).
- **Inconsistent make/flatten calls:** The type definitions are inconsistent with the types usage in Make or Flatten calls.
- **Inconsistent constant values:** Constant values are inconsistent with the type definition of the constant.

There are several means for users to identify and fix refactoring issues in models:

- List of refactoring issues in the **Inconsistencies** output tab.
- Visual identification of refactoring issues across model design.
- Availability of quick fixes applicable to reported issues. For details, see [“Applying Quick Fixes to Refactoring Issues”](#) on page 544.
- Availability of user-defined fixes applicable to inconsistencies reported by the **Resolve Inconsistencies** wizard. For details, see [“Applying User Fixes to Inconsistencies”](#) on page 545.

List and Visualization of Refactoring Issues

Whenever opening SCADE Suite models containing inconsistencies or making design changes that cause refactoring issues, the **Inconsistencies** output tab is automatically refreshed and reports information about the nature and location of inconsistencies.

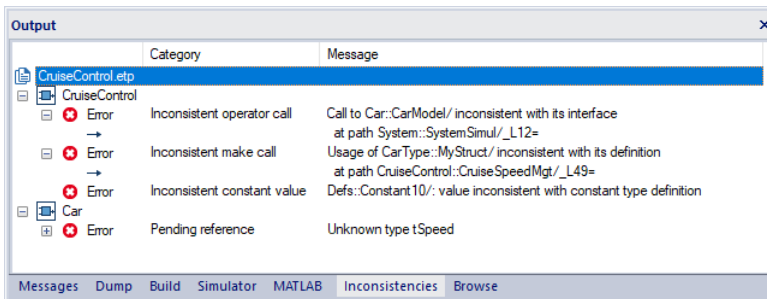


Figure 9.1: Reporting of refactoring issues in Output window

Notice that reported issues are grouped by model in a hierarchical view.

To locate elements with refactoring issues in model

- In **Inconsistencies** tab, double-click any item, or right-click it and select *Locate*. The element is highlighted in the appropriate view.

Elements with refactoring issues are identified by a visual cue  in diagrams, in Scade Model Browser, and in Constants/Sensors/Types Views.

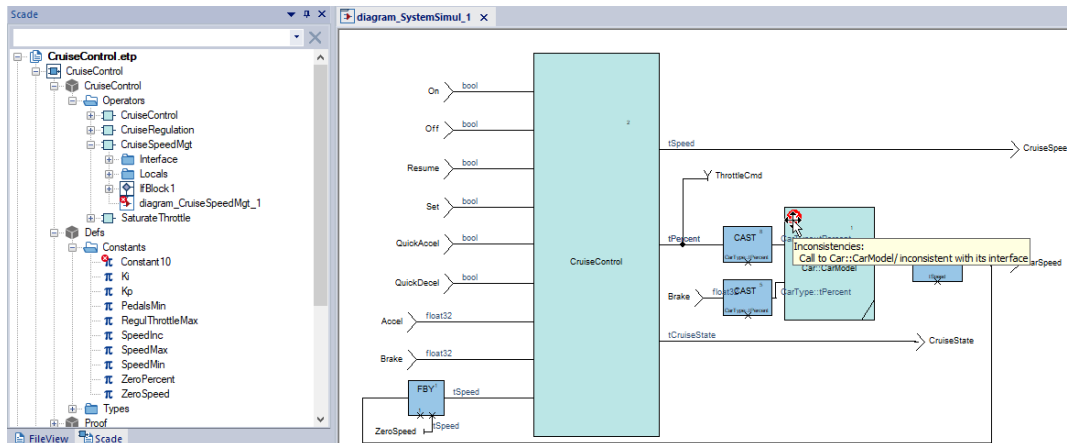


Figure 9.2: Visualizing refactoring issues in model

To access information about refactoring issues in model

- Mouse over any element with a visual cue to display a tooltip containing information about reported issues.

Applying Quick Fixes to Refactoring Issues

Refactoring issues reported in **Inconsistencies** output tab, diagrams, model browsers, and views can be resolved by applying quick fixes on them.

To apply quick fixes from Inconsistencies tab

- Right-click any reported issue or a multi-selection of reported issues and select *Quick Fix*. The fix applies to the selected issue(s) as well as to all issues under the hierarchy of the selection.

To apply quick fixes from diagrams, Scade browser, or Constants/Sensors/Types views

- Right-click any element reported with refactoring issue (element displayed with visual cue) and select *Quick Fix*.

HOW QUICK FIXES APPLY?

- When applying fixes to pending references, the reference to the element is restored based on the element OID.
- When applying fixes to user operator calls, the number and order of input pins, output pins, and parameter values are fixed according to interface definition.
- When applying fixes to make/flatten calls, the number of input pins and output pins is fixed according to type definition.
- When applying quick fix to constant value inconsistency, the constant value is modified to match the type definition of the constant.

In case a quick fix cannot be applied, an explicative message is displayed.

Applying User Fixes to Inconsistencies

SCADE Suite allows for applying user-defined fixes to model inconsistencies through a dedicated wizard.

- [“Launching Resolve Inconsistencies Wizard”](#)
- [“Resolving Missing Declarations”](#)
- [“Resolving Operator Interface Inconsistencies”](#)
- [“Resolving Type Inconsistencies”](#)
- [“Resolving Constant Value Inconsistencies”](#)
- [“Generating Inconsistencies Resolution Report”](#)

Launching Resolve Inconsistencies Wizard

You can launch the dedicated wizard to resolve inconsistencies present in models by applying some user-defined fixes.

To launch Resolve inconsistencies wizard

- 1 Select *Project > Resolve Inconsistencies* from the current project.
Depending on reported inconsistencies:
 - **Missing Declarations:** see [“Resolving Missing Declarations”](#) on page 546.
 - **Operator Interface Inconsistencies:** see [“Resolving Operator Interface Inconsistencies”](#) on page 548.
 - **Type Inconsistencies:** see [“Resolving Type Inconsistencies”](#) on page 550.
 - **Constant Value Inconsistencies:** see [“Resolving Constant Value Inconsistencies”](#) on page 552.

Resolving Missing Declarations

If the wizard reports missing declaration(s) in the model, you need to specify the new name of the element(s).

Warning

Get the new name(s) from the person who updated the content of the library that causes such model inconsistencies.

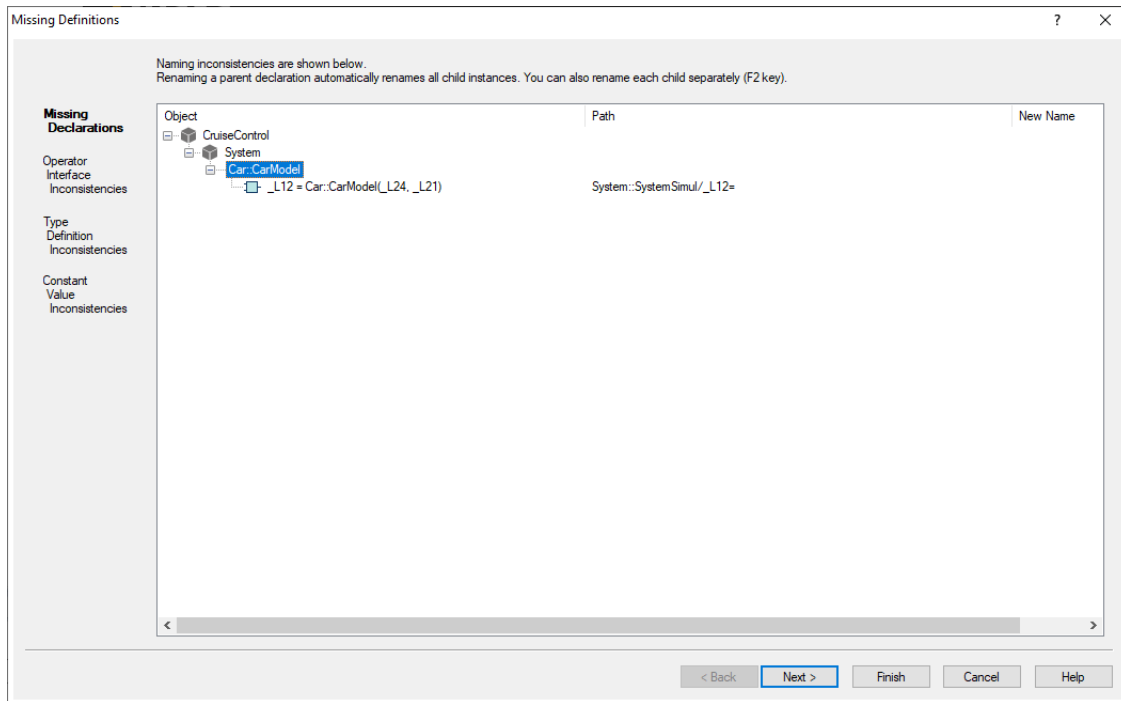


Figure 9.3: Reported missing declarations

To resolve missing declarations

- 1 In **Missing Definitions** window, right-click the missing object.
- 2 Select **New Name**, or press F2 after selecting an object in the list.

- 3 Enter the new name in active field of **New Name** column. Start typing text and press Ctrl+Space keys to use auto-completion. Select type in drop-down list. New name can be applied globally on the first line and/or for each object occurrence listed under the object name reported as missing.
 - 4 Select **Locate** of object occurrence to open the graphical view.
 - 5 Click **Next** to solve other reported inconsistencies (if any) and display a summary of resolved inconsistencies if no other inconsistency is reported (see [“Generating Inconsistencies Resolution Report”](#) on page 553).
-

Tip

Click Finish to skip other inconsistencies reported in the project and generate directly the report about resolved inconsistencies.

Resolving Operator Interface Inconsistencies

If the wizard reports inconsistent operators in the model, you need to redefine appropriately the mapping(s) between interface element(s) and call parameter(s).

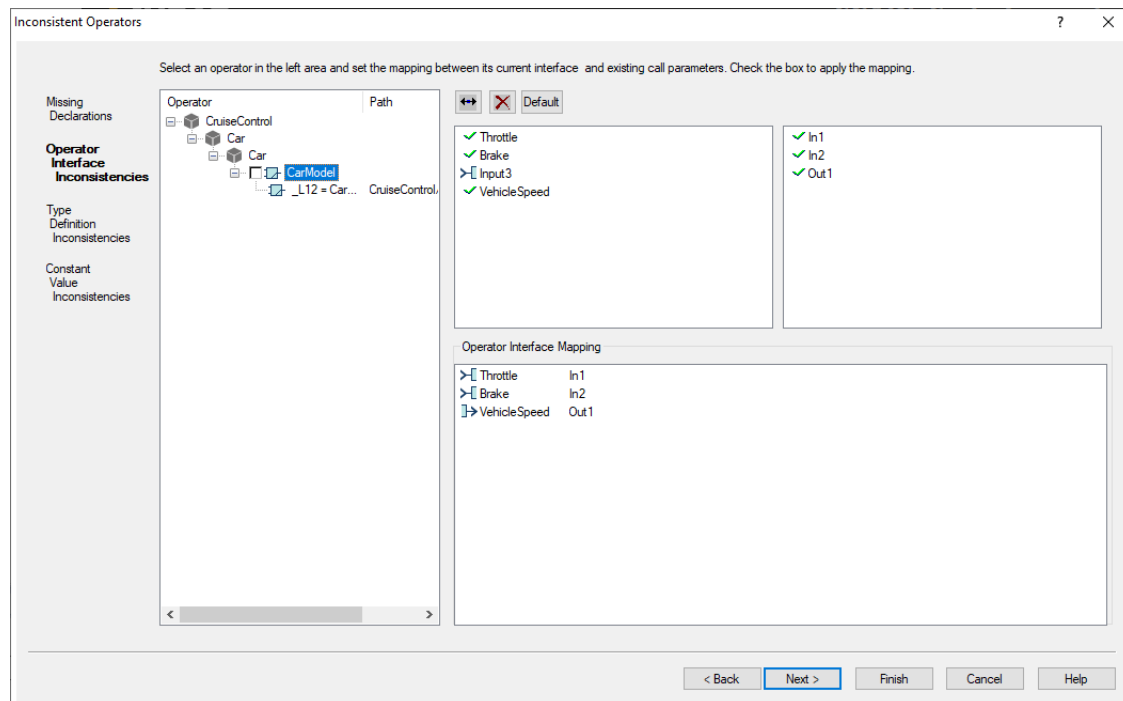


Figure 9.4: Reported operator interface inconsistencies

To resolve operator interface inconsistencies

- 1** In **Inconsistent Operators** window, select an operator on the left area.

Tip

Right-click an operator occurrence and select Locate to display it in a diagram.

- 2 Select a mapping in the **Operator Interface Mapping** area and click **Delete** to remove. Repeat for each existing mapping if necessary.

- 3 Create a new mapping (if necessary) as follows:
 - Select operator input/output or parameter in the upper left list where items are named as defined in model files for the operator definition.
 - Select a call parameter in the upper right list to specify a new mapping. In the list, items of operator calls are named `In<i>`, `Out<i>`, or `instParam<i>` based on their graphical order. Edit the selected mapping as follows:
 - Click  to set the new mapping between the selected list items. The mapping set between the first and second list areas is shown in the lower list of the **Operator Interface Mapping** area. A mapping is set by default and used as such after clicking **Next** or **Finish**. You map only free elements (*i.e.*, no green check mark)
 - Click **Default** to apply initial mappings before solving inconsistencies.
- 4 Check each operator in the Operator tree list to resolve for inconsistencies with the newly specified mapping. Leave unchecked to skip resolution of the definition mapping.
- 5 Click **Next** to solve other reported inconsistencies (if any) and display a summary of resolved inconsistencies if no other inconsistency is reported (see [“Generating Inconsistencies Resolution Report”](#) on page 553).

Tip

Click **Finish** to skip other inconsistencies reported in the project and generate directly the report about resolved inconsistencies.

Resolving Type Inconsistencies

If the wizard reports inconsistent types usage in Make or Flatten calls in the model, you need to redefine appropriately the mapping(s) between Type Definition and Make or Flatten parameters.

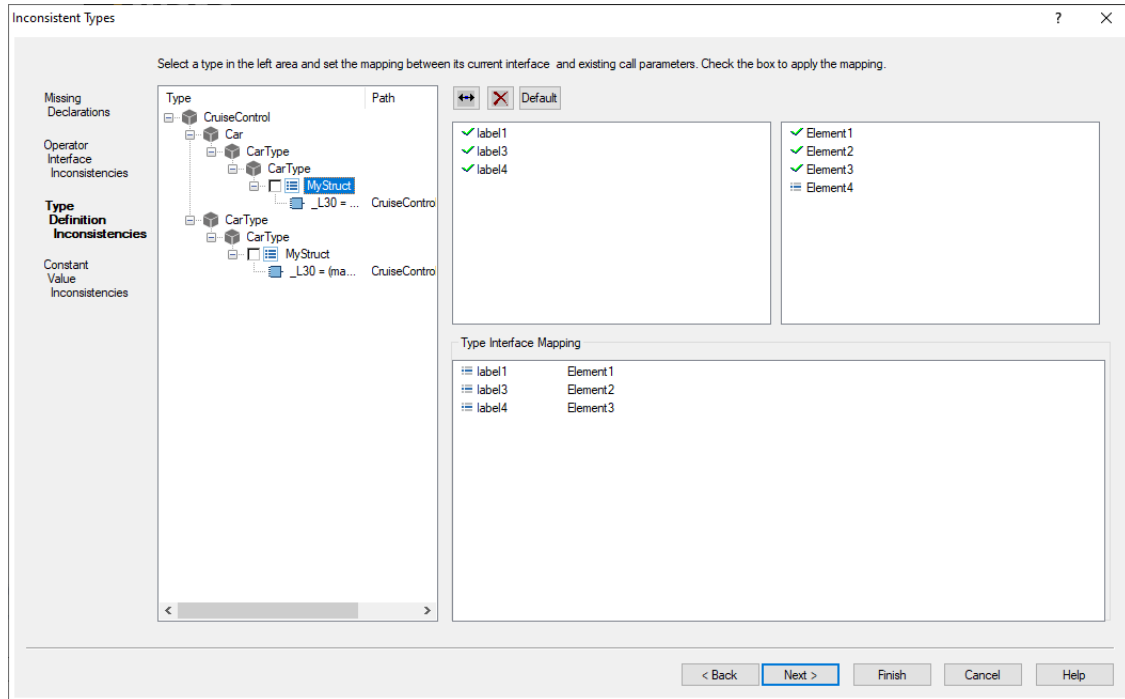


Figure 9.5: Reported type definition inconsistencies

To resolve type inconsistencies

- 1 In **Inconsistent Types** window, select a type on the left area.

Tip

Right-click a Make or Flatten call and select Locate to display it in a diagram.

- 2 Select a mapping in the **Type Definition Mapping** area and click **Delete** to remove. Repeat for each existing mapping if necessary.

- 3 Create a new mapping (if necessary) as follows:
 - Select a type element in the upper left list where items are named as defined in the model files for the type definition.
 - Select a call parameter in the upper right list to specify a new mapping. In the list, the items of Make or Flatten calls are named Element<i> according to their graphical order. Edit the selected mapping as follows:
 - Click  to set the new mapping between the selected list items. The mapping set between the first and second list areas is shown in the lower list of the **Type Definition Mapping** area. A mapping is set by default and used as such after clicking **Next** or **Finish**. You map only free elements (*i.e.*, no green check mark)
 - Click **Default** to apply initial mappings before solving inconsistencies.
- 4 Check each type in the Type tree list to resolve for inconsistencies with the newly specified mapping. Leave unchecked to skip resolution of the definition mapping.
- 5 Click **Next** to solve other reported inconsistencies (if any) and display a summary of resolved inconsistencies if no other inconsistency is detected (see [“Generating Inconsistencies Resolution Report”](#) on page 553).

Tip

Click Finish to skip other inconsistencies detected in the project and generate directly the report about resolved inconsistencies.

Resolving Constant Value Inconsistencies

Inconsistencies on constants are simply listed for information. You must correct constant value inconsistencies by matching the values of the constants with their types.

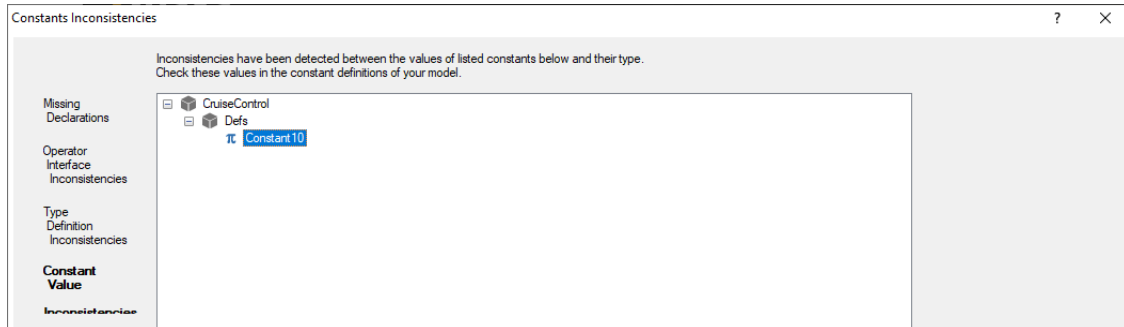


Figure 9.6: Reported constant value inconsistency

To resolve constant value inconsistencies

- 1 Resolve constant inconsistencies from corresponding Constant packages.
- 2 Click **Next** to display a summary of all resolved inconsistencies (see [“Generating Inconsistencies Resolution Report”](#) on page 553) or click **Finish** to generate directly the report about resolved inconsistencies.

Generating Inconsistencies Resolution Report

After launching the wizard, a summary of resolved references displays. It allows to generate or not a complete report with hyperlinks to resolved inconsistencies.

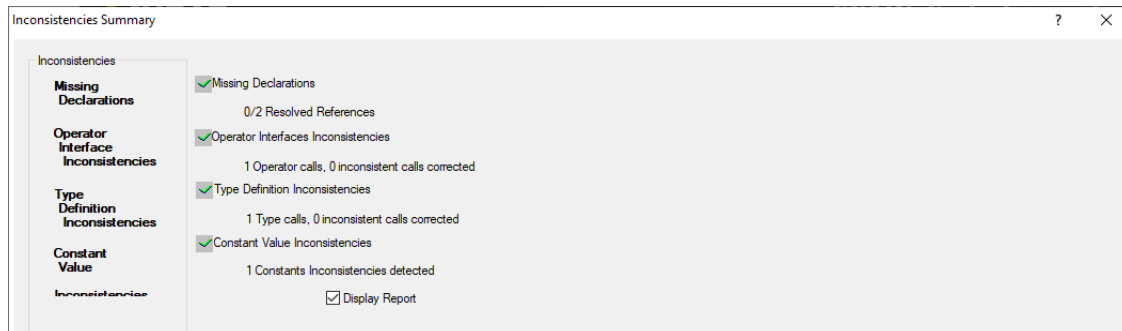


Figure 9.7: Result window summing up resolved inconsistencies

To close the wizard without report

- 1 Uncheck **Display Report** if you do not need to generate the HTML report. In the report, use hyperlinks to locate resolved entities in the graphical representation of the model.
- 2 Click **Finish** to close the wizard.

10 /Running Coverability Analysis on Design

This chapter explains how to analyze coverability of SCADE Suite models.

- [“Coverability Analysis Diagnostics and Report”](#)
- [“Setting Coverability Analysis”](#)
- [“Launching Coverability Analysis”](#)
- [“Launching Coverability Analysis Batch”](#)

Coverability Analysis Diagnostics and Report

Before testing SCADE Suite models, during design phase, a coverability analysis can be performed to detect uncoverable parts in the design. When possible, the design is reworked to remove uncoverable parts and improve model testability, or it can be justified if uncoverable parts are expected.

For each coverage point (see definition in [“Model Coverage Measurement for SCADE Suite”](#) on page 22 in *SCADE Test User Manual*) contained in the operator(s) analyzed in SCADE Suite models, coverability analysis sessions provide the following diagnostics:

- **Coverable:** the point can be reached and an input sequence (.sss file) is provided to show how the point can be observed.
- **Not Coverable:** the point cannot be reached and it is formally proved that there is no input sequence to cover it.
- **Justified:** the point is justified by the designer.
- **Undecided:** analysis could not be fulfilled for the point.

Coverability analysis relies on a proof engine. Such analysis can only be launched on semantically correct models. The results from coverability analysis are presented in a dedicated “Model Coverability Analysis” report. It contains session information, an analysis result synthesis for coverage points under analysis, detailed results for each coverage point, and details about proof engine assumptions.

Model Coverability Analysis

1. General information

Date: 2024-09-25T18:01:54
Project: C:\Users\Public\Documents\ANSYS
Inc\v251\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp
Root operator: CC::CruiseControl
Criterion: OMCDC
Justification files: [CruiseControl.mcj](#)
Detailed configuration: [config.txt](#)
Output directory: [CC_CruiseControl](#)
Generated justifications: [CC_CruiseControl.mcj](#)

Results synthesis

Coverage points under analysis	Coverable	Not coverable		Undecided
		Justified	Not justified	
190	188	1	1	0

Context of the analysis: [go to section](#)

Figure 10.1: Example of Model Coverability Analysis report

The analysis provides the list of uncoverable points of the model and input sequences leading to the observability of the coverable points.

Setting Coverability Analysis

Before running coverability analysis session on SCADE Suite models, you set coverability analysis as explained below.

To set coverability analysis for SCADE Suite projects

- 1 In loaded project, select *Project > Model Coverability Analysis > Settings* to open the **Coverability Analysis Settings** window.

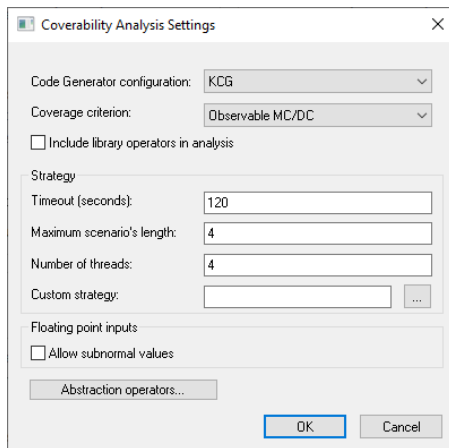


Figure 10.2: Setting coverability analysis in SCADE Suite

- 2 In **Code Generator configuration**, select a configuration of Code Generator tool. Some code generation settings have an influence on the coverability measurement.
- 3 Set **Coverage criterion** to **Influence**, **Observable DC**, or **Observable MC/DC** to define the coverability measurement. For details about coverage criteria, see [“Coverage Criteria for SCADE Suite Model Coverage”](#) on page 26 in *SCADE Test User Manual*.
- 4 Check **Include library operators in analysis** to extend coverability analysis to library operators.

5 Set **Strategy** options as follows:

- In **Timeout**, set the amount of time in seconds after which analysis may interrupt. Timeout set at 0 means that analysis never stops.
 - In **Maximum scenario's length**, set the maximum number of cycles used to produce the scenario. For details about strategy settings and analysis time, see [“Impact of Strategy on Coverability Analysis”](#) on page 139 in *SCADE Test User Manual*.
 - In **Number of threads**, set the number of threads usable during analysis (within the limit of processor cores available on host). Set at 0 to mean that analysis uses the maximum number of processor cores.
 - **Custom Strategy** is an advanced setting allowing to specify one's own strategy. For more details, see [“Defining Custom Strategies”](#) on page 765.
- 6 In **Floating point inputs**, check **Allow subnormal values** to specify that subnormal floating point values are present in covering input sequences.
- 7 Click **Abstraction operators** to define a list of libraries containing the abstraction operators to be used during analysis. Abstraction operators can be used to replace imported operators or complex operators by simpler implementation allowing faster analysis. For details, see [“Enabling Analysis Using Abstraction”](#) on page 749.

Launching Coverability Analysis

During design phase, coverability analysis can be launched operator by operator in SCADE Suite model. It is recommended to perform such model analysis as soon as operators are designed allowing to rework their design if uncoverable parts are detected. Notice that assume assertions in analyzed operator are always considered as constraints during coverability analysis.

To launch coverability analysis during design phase

- 1 Make sure the entire SCADE Suite project is semantically correct using SCADE Suite Checker.
- 2 Select the operator to analyze in **Scade View**. The model must be complete (no error detected by Check on entire model).
- 3 Right-click the operator to analyze and select *Model Coverability Analysis > Analyze*. Analysis is launched based on preset options (see [“Setting Coverability Analysis”](#) on page 557). In status bar, some information displays about analysis progress. The result report opens.

Note

In case model or settings changed, impacting model coverage, a message proposes to start a new analysis after clearing previous analysis results. If the proposal is not accepted, then analysis is canceled.

-
- 4 Analyze coverability results in the “**Model Coverability Analysis**” report that opens upon analysis completion.

Note

In case users need details about proof engine execution, it is possible to look into the `mc_stdout` . Log available from Info Log tab in Output docking window.

-
- 5 In the **Detailed results** part of the report, it is possible to switch between:
 - *full results* (full set of shortest input sequences for each coverage point)

- *compact results* (compact set of input sequences without sequence redundancies for core¹ coverage points).

Clicking on column headers sorts the table content by alphabetical order.

2. Detailed results

Switch to full results

Table content:

Compact set of input sequences without sequence redundancies for core coverage points

Number of covering sequences:

16

2.1. CruiseControl::CruiseControl

Coverage Point	Result
[A] SM1:Enabled;	sc0.sss
[I] SM1:Enabled:CruiseSpeed=	_L8 sc0.sss
[A] SM1:Off;	sc2.sss
[I] CruiseSpeed.default	ZeroSpeed sc2.sss
[A] SM1:Enabled:SM2:Interrupt;	sc0.sss
[A] SM1:Enabled:SM2:Active;	sc5.sss
[A] SM1:Enabled:SM2:Active:SM3:StandBy;	sc6.sss
[A] SM1:Enabled:SM2:Active:SM3:On;	sc5.sss
[I] SM1:Enabled:SM2:Active:SM3:On:ThrottleCmd=	_L3 sc5.sss
[I] SM1:Enabled:SM2:Interrupt:CruiseState=	_L3 sc0.sss
[I] SM1:Enabled:SM2:Active:SM3:StandBy:CruiseState=	_L3 sc6.sss
[I] SM1:Enabled:SM2:Active:SM3:On:CruiseState=	_L12 sc5.sss

Figure 10.3: Detailed results about coverability analysis on SCADE Suite model

- 6 In **Coverage Point** column, click a path to locate any coverage point in the model. Look at tooltips on each path to get more information.
- 7 In **Result** column, click any file link to open the generated input sequence corresponding to the coverable point.

To rerun analysis on undecided coverage points

- In **Scade View**, right-click a previously analyzed operator and select *Model Coverability Analysis > Analyze* to run analysis again only on undecided coverage points.

1. Core coverage points are the minimal set of coverage points that can ensure the complete coverage of the application.

To delete analysis results from analyzed operator

- In **Scade View**, right-click a previously analyzed operator and select *Model Coverability Analysis > Clear*. All analysis results are removed.

To open report from previous analysis

- In **Scade View**, right-click a previously analyzed operator and select *Model Coverability Analysis > View Results*. The last result report opens.

To interrupt analysis

- In **Scade View**, right-click analyzed operator and select *Model Coverability Analysis > Stop*. The ongoing analysis stops.

Notice it is possible to set and launch coverability analysis as described above using command lines. For details, see [“Launching Coverability Analysis Batch”](#) on page 564.

Whenever the design of a model contains uncoverable parts that cannot be reworked, it is possible during the design phase to add justifications. Note that such justifications are also taken into account by SCADE Test during the testing phase.

To add justifications for coverage points analyzed as Not Coverable

- 1 Identify in the “**Model Coverability Analysis**” report the coverage points (marked as **Not Coverable**) that may need to be justified. A mark (*) is used to distinguish points found by instrumentation from points found by coverability analysis.

2.1. Operator1

Coverage Point	Result
[T] Output2= _L7	Not Coverable [Justify]
[I] _L7= if _L4 then (_L8) else (_L9)	Not Coverable [Justify]
[A] IfBlock1:else: Output4=	Not Coverable(*) [Justify]
[I] IfBlock1:else:Output4= _L1	Not Coverable(*) [Justify]
[F] Output1= _L3	sc0.sss
[T] Output1= _L3	sc1.sss

Figure 10.4: Coverability analysis results identifying uncoverable parts

- 2 In **Result** column, click **Justify** link for each coverage point to be justified.
- 3 In **Create Justification** dialog, specify the author of the specification and provide a description of the justification.

The description indicates whether the point to be justified is found by instrumentation or by coverability analysis.

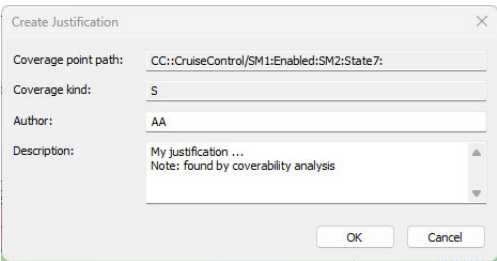


Figure 10.5: Defining justification for non coverable point

- 4 Click **OK** to add the defined justification in the analyzed operator.

Note

All justifications created for non coverable points of the operator under analysis are saved in a single .mcj file. If no file exists in the project, a new one is created and added in File View when creating the first justification.

- 5 Right-click the operator and select *Model Coverability Analysis > Analyze* to rerun analysis and take into account justification(s). In the report, any coverage point for which a justification was added, displays as **Justified** in the Result column.

Launching Coverability Analysis Batch

Use the following command to run model coverability analysis on a SCADE Suite project:

```
dv_mc [-criterion <criterion>] [-instr <instr_ops>] [-noinstr <noinstr_ops>]  
      -target HLL -root <OperatorName> -target_dir <dir>  
      [-timeout <sec>] [-threads <int>] [-no_subnormal] [-tag_file <File Path>]  
      [-bmc_depth <depth>] [-strategy <StrategyString>] [-dontcares_removal] [<KCG_Options>]  
dv_mc -mc_config <file> [-config <file>]
```

Command options and parameters:

- **-criterion** <criterion> to specify the coverability measurement. Valid arguments are: influence, ODC, or OMCDC. Default is OMCDC when not specifying the option.
- **-instr** <instr_ops> to specify a comma-separated list of operator paths included in coverability analysis if present in the call graph of the specified operator (**-root**). All other operators are not analyzed. Option incompatible with **-noinstr**.
- **-noinstr** <noinstr_ops> to specify a comma-separated list of operator paths not included in coverability analysis. All other operators in the call graph of the specified operator (**-root**) are analyzed. Option incompatible with **-instr**. If neither **-instr** nor **-noinstr** options are set, all operators in the call graph of the specified operator are included in analysis.
- **-target** HLL (mandatory) to launch coverability analysis.
- **-root** <OperatorName> (mandatory) to specify the path to the operator to be analyzed.
- **-target_dir** <dir> (mandatory) to specify the directory where the result files are generated.
- **-timeout** <sec> (optional) to specify the amount of time in seconds after which analysis may interrupt. Setting timeout to 0 means that analysis never stops because of timeout.
- **-threads** <n> (optional) to specify the number of threads that can be used during analysis (within the limit of processor cores available on host). A 0 value means the analysis run is using the maximum number of processor cores.
- **-no_subnormal** (optional) to specify that subnormal floating point values are not used in covering input sequences.
- **-tag_file** <FilePath> (optional) to restrict the analysis to a subset of coverage points specified in <FilePath>. The file describing coverage points must use the file format described below.
- **-bmc_depth** <depth> (optional) to specify the maximum value of scenario length. If option not set, depth equals 100.
- **-strategy** <StrategyString> (optional) to customize the strategy used by the proof engine. If using this option, **-timeout**, **-threads**, and **-bmc_depth** are ignored. About strategies from proof engine, contact Ansys.
- **-dontcares_removal** (optional) to generate scenarios where inputs are set only if the assigned value is relevant for the coverage objective. Without this option, the generated scenarios contain a complete valuation (*i.e.*, at each cycle, all inputs are set to a value).

- **-mc_config** <file> to read from a textual file any of the above options and arguments available for coverability analysis.
- <KCG_Options> must be those of the KCG command line used to produce C target embedded code (see [“Launching KCG Executable in Command Line”](#) on page 866). Yet, some KCG options can be used indifferently because they have no impact on coverage (see this [list](#)). Notice that if ever using -keep_asserts option in the command line, coverage analysis treats assume assertions in the root operator as constraints of the problem. Options can be passed using **-config** <file> or command line arguments.

OUTPUT FILES

Files produced in batch coverability analysis are named and stored as follows:

Analysis results for each coverage point in the model	report.json
Input sequence (for each coverage point found to be coverable)	*.sss
Traceability file (for all coverage points in the model)	<root_name>.tra
Coverability analysis session log	mc.log
All files are stored in the specified target directory.	

COVERAGE POINT FILE FORMAT

A coverage point is represented by JSON objects representing coverage points with the following fields:

- **path** key is the Scade model path of the point
- **kind** key is the kind of the coverage point (see [“List of Coverage Tags”](#) on page 158)

Example of coverage point file:

```
{
  "observations": [
    {
      "status": [
        {
          "path": "CruiseControl::CruiseControl/SM1:Enabled:SM2:Active:_L1=",
          "kind": "T"
        },
        {
          "path": "CruiseControl::CruiseControl/SM1:Enabled:SM2:Active:_L4=",
          "kind": "T"
        }
      ]
    }
  ]
}
```

Part 2

Simulation and Debugging

- Chapter 11: [“Fundamentals about SCADE Suite Simulation”](#)
- Chapter 12: [“Running Simulation in SCADE Suite”](#)
- Chapter 13: [“Running Simulation from External Tools”](#)
- Chapter 14: [“Running Advanced Simulation”](#)
- Chapter 15: [“Generating Desktop and Mobile Applications”](#)

11 / Fundamentals about SCADE Suite Simulation

This chapter contains generic information about **SCADE Suite Simulation** usage in the context of first debugging and simulation of SCADE Suite models by executing C/Ada code transparently generated by SCADE Suite Code Generator.

- [“Model-Based Simulation Principles”](#)
- [“Exploring Simulator User Interface”](#)
- [“Simulation Workflow”](#)

Model-Based Simulation Principles

Model simulation with SCADE Suite relies on a specific architecture and environment with a variety of tools allowing interactive simulation.

- [“Simulation Architecture”](#)
- [“Observable Variables and Traceability”](#)
- [“Simulation Cycles”](#)
- [“SCADE Simulation Executables”](#)
- [“Step-by-Step or Continuous Simulation”](#)
- [“Simulation Scenarios/States: Storing and Replaying”](#)
- [“Breakpoints”](#)

Simulation Architecture

As illustrated in [Figure 11.1](#), the basic architecture of simulation within SCADE Suite Editor relies on the ability to generate a simulation executable from the code generated by SCADE Suite Code Generator and to execute it for simulation purposes. The dynamic behavior of the simulation executable is identical to the generated code.

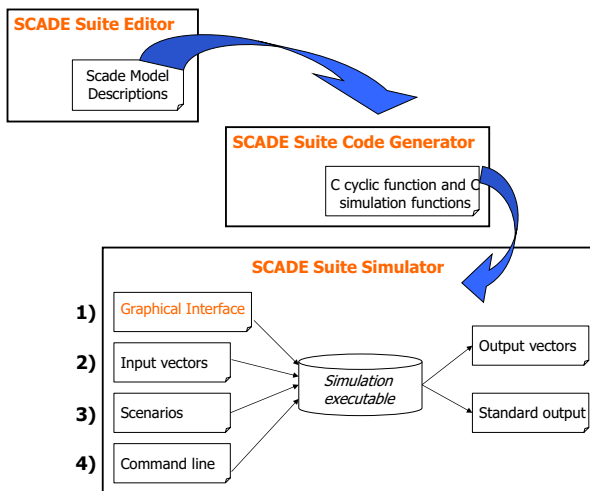


Figure 11.1: Procedures available for running simulation in SCADE Suite Editor

Several operating procedures are available using simulation executables:

- 1 Interactive simulation from SCADE Suite graphical interface
- 2 Data-driven simulation (in batch mode)
- 3 Scenario-driven simulation (in batch mode)
- 4 Interactive scenario-driven simulation (in batch mode)

About simulation batch options, see Chapter 6 about [“SCADE Suite Simulation Batch”](#) in *SCADE Suite Technical Manual*.

Observable Variables and Traceability

The variables that are observable in simulation are defined as any variable traceable between the model and the generated code. Variable traceability is impacted by code generation optimization. Maximum traceability is reached when local variables observation is enabled in simulation and when no code optimization is set.

Note

If you ever need to use embeddable code in simulation, consider that some variables cannot be traced anymore because they are optimized or local. The behavior of the generated code is independent from the quantity of traceability information requested.

About variable observability, see [“Setting Variable Visibility”](#) on page 594.

Simulation Cycles

A cycle is the logical time used by SCADE Suite when simulating C/Ada generated code. It corresponds to one execution of the built simulation executable. At each cycle, outputs are computed based on model inputs. You can trigger and observe system reactions by executing simulation cycles step by step with series of input values, or continuously with user input values set at run-time (cycles executed successively). About simulation execution, see [“Executing Simulation Cycles”](#) on page 597.

SCADE Simulation Executables

Simulation executables are compiled based on the code generated by SCADE Suite Code Generator. SCADE Suite Simulation generates the executable which is required to simulate the C/Ada code corresponding to the model design. Such executables result from compiling cyclic functions generated by SCADE Suite Code Generator and the C/Ada code that allows to monitor the code. About simulation executable generation, see [“Launching Simulation Session”](#) on page 596.

Step-by-Step or Continuous Simulation

Two simulation modes are available in SCADE Suite:

- Step-by-step simulation allows entering input values on determined model inputs and at specific simulation cycles. About running step-by-step simulation, see [“Simulating Model Step by Step”](#) on page 598.
- Continuous simulation can be run without any scenario and allows entering input values on the fly. About running continuous simulation, see [“Simulating Model in Continuous Mode”](#) on page 600.

Simulation Scenarios/States: Storing and Replaying

After running a simulation session on a model, you can store every step of the simulation in scenarios. Creating scenarios allows replaying simulation sessions with input values saved in simulation input files. In addition, application states allow restarting simulation from a given state of the application without executing all steps from the beginning.

You can replay scenarios step-by-step or in continuous mode, or you can load application states. Simulation scenarios can be described as input values (.in files) or as simulation scenario functions (.sss files). Application states can be described in binary format (.sts files) or in textual format not supporting step by step simulation (.sns files). About simulation scenarios, see [“Recording Simulation Scenarios/Application States”](#) on page 614 and [“Simulating Model in Continuous Mode”](#) on page 600.

Breakpoints

Breakpoints are debugging tools you set to interrupt simulation on particular locations in the models. You can enable breakpoints on various control and data flow elements in models.

When reaching breakpoints, SCADE Suite Simulation stops the simulation and automatically opens the **Diagram View** of the operator containing the element. About breakpoints, see [“Setting and Managing Breakpoints”](#) on page 616.

Exploring Simulator User Interface

SCADE Suite Simulation is a multi-view tool sharing with SCADE Suite the same graphical interface and the same navigation and information display principles.

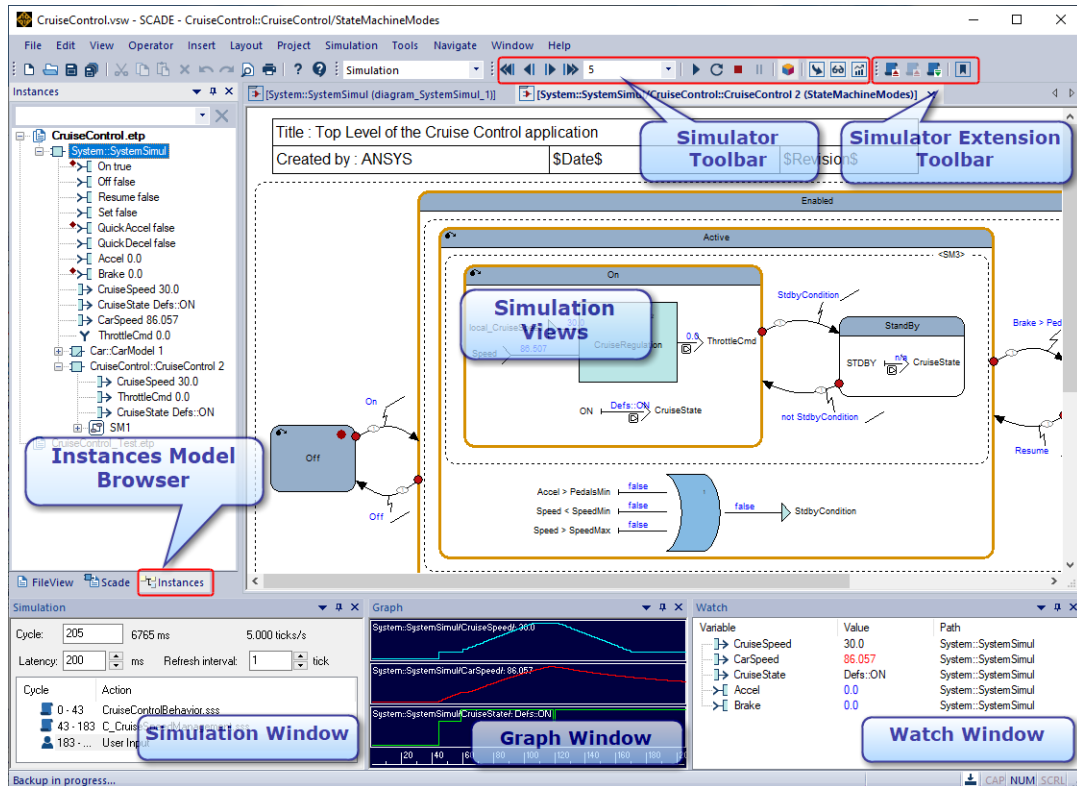


Figure 11.2: Simulator environment in simulation session

The main interface components are introduced below:

- [“Instances Model Browser”](#)
- [“Simulation Views”](#)
- [“Simulation Window”](#)
- [“Watch Window”](#)
- [“Graph Window”](#)
- [“Simulation Toolbars”](#)

Instances Model Browser, Simulation, Watch, and Graph Windows are dockable views. Their usage and layout are described in [“Organizing and Working with Docking Windows”](#) on page 154. The docking window layout is saved by [Perspectives](#).

Instances Model Browser

The **Instances Model Browser** represents a hierarchical tree view with all model instances observable during the simulation session. All observable variables and operators are displayed with their interface and information. The elements that cannot be observed appear in grey.

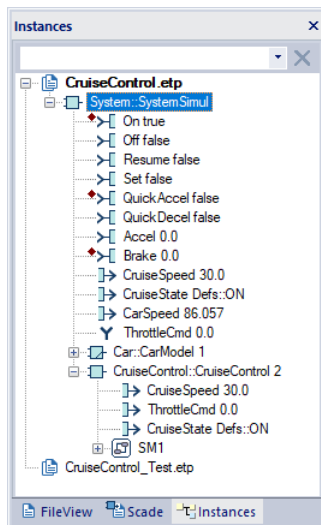


Figure 11.3: Instances Model Browser

This view highlights active items (states, transitions, branches and matches from If and When blocks) and shows if breakpoints are set on control and data flow elements. To observe the maximum of model variables, see [“Setting Variable Visibility”](#) on page 594.

Simulation Views

The **Simulation View** of an operator instance is a read-only view which displays its graphical representation in a diagram with values on observable variables instances.

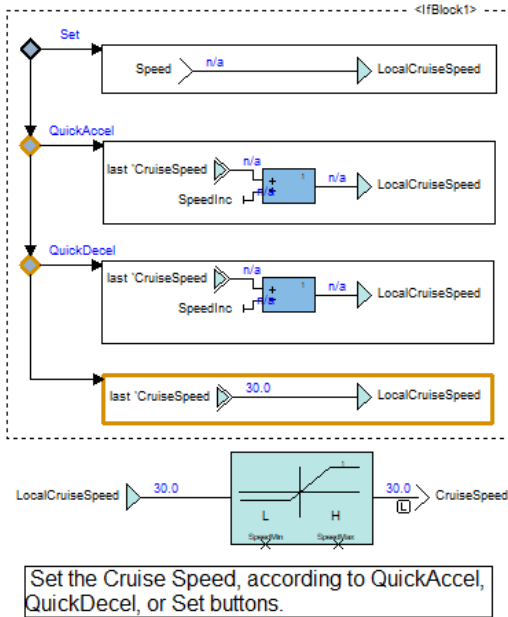


Figure 11.4: Example of observable elements highlighted in Simulation View

Navigation across operators in the design from the **Instances Model Browser** or **Simulation Views** is similar to model navigation in model editing. For navigation instructions, see Chapter 5 about [“Navigating in Models”](#).

Simulations views display as tabs like model editing views. Their usage and layout is described in [“Organizing and Working with Model Editing Views”](#) on page 144.

Simulation Window

In this window, you track information on the active simulation session, check out which step is on, or if a scenario is loaded. It also allows to move to a given simulation cycle or to set the latency between cycles when simulating in continuous mode. Read about [“Managing Simulation Execution”](#) on page 612 and [“Playing Simulation Scenarios/States”](#) on page 600.

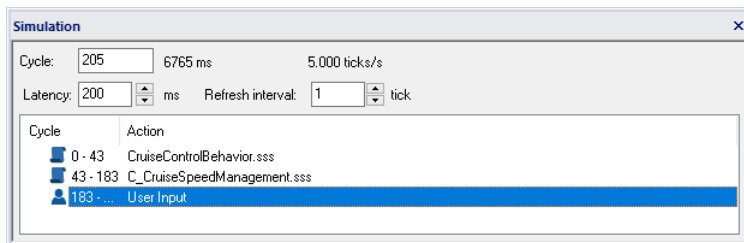


Figure 11.5: Simulation window

Several icons are displayed to highlight the simulation history:

- **User Input:** simulation is driven by input values from the user
- **Scenario input:** simulation is driven with input values from scenario
- **Bookmark** loaded or **Back** command pressed
- **External:** indicates co-simulation mode
- The black arrow indicates what action Simulator is in

Watch Window

In this window, you display observable variables selected from the **Instances Model Browser** and set their value before triggering model reaction and executing simulation steps. Read about [“Simulating Model Step by Step”](#) on page 598 and [“Displaying Variables in Watch Window”](#) on page 609.

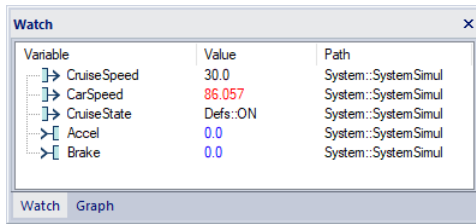


Figure 11.6: Watch window

The values of observable variables display in **Red** for values that changed during simulation cycle, or in **Blue** for values that changed before execution of next cycle.

Graph Window

In this window, you graph variables selected from the **Instances Model Browser** or the **Watch window** during simulation sessions. Only scalar variables are represented.

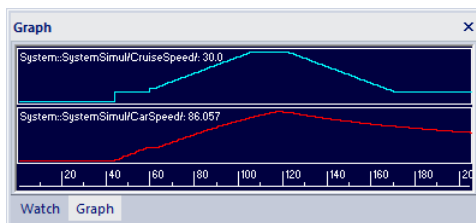


Figure 11.7: Graph window

For modifying the display in the **Graph** window, see [“Setting Simulation Environment”](#) on page 583. For instructions, see [“Displaying Variables in Graphical Charts”](#) on page 608.

Simulation Toolbars

These toolbars provide shortcuts to manage and operate simulation sessions: windows display, cycles execution, scenarios and bookmarks management. They can be docked to the top, the bottom, the left, or the right of the user interface. They can be moved in the interface by dragging their title bar to the desired location. The toolbar layout is saved by [Perspectives](#).

The following toolbars are available:

Simulator	Managing simulation cycle execution: • “Simulating Model Step by Step” on page 598 • “Simulating Model in Continuous Mode” on page 600 • Displaying dedicated windows (Simulation, Watch, Graph)
Simulator Extension	Managing scenarios and bookmarks: • “Recording Simulation Scenarios/Application States” on page 614 • “Playing Simulation Scenarios/States” on page 600 • “Setting and Using Bookmarks” on page 601

To show/hide toolbars

- Select *View > Toolbars* and select the name of the toolbar; or right-click in the toolbar area and select the name of the toolbar to show/hide from the interface. Alternatively, hide any toolbar by right-clicking on toolbar and selecting *Hide*.

Simulation Workflow

The workflow of simulation consists of successive tasks that you can iterate as you need.

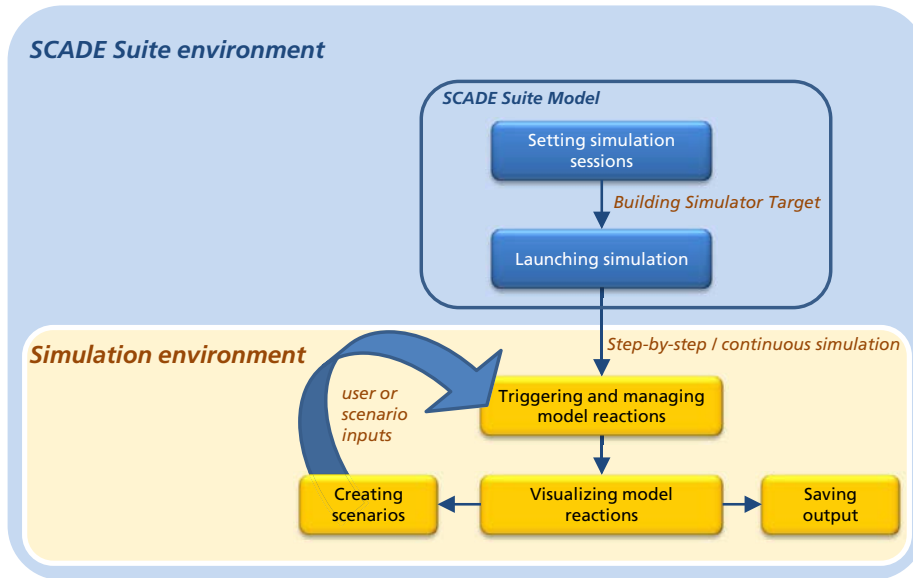


Figure 11.8: Simulation workflow

Once design specifications and system safety requirements are identified, the simulation of any SCADE Suite design can follow several phases as described below:

- 1 Setting simulation sessions: see [“Setting Simulation Environment”](#) on page 583 , [“Setting Project for Simulation Sessions”](#) on page 585, and [“Setting Simulation Code for Simulation Sessions”](#) on page 586.
- 2 Launching simulation sessions: [“Launching Simulation Session”](#) on page 596.
- 3 Triggering and managing model reactions during simulation sessions: see [“Executing Simulation Cycles”](#) on page 597.

For debugging purposes, you can set breakpoints in models to interrupt simulation execution at preset cycles or under determined conditions: see [“Setting and Managing Breakpoints”](#) on page 616.

- 4 Visualizing model reactions graphically: see [“Displaying Variables in Graphical Charts”](#) on page 608, [“Displaying Variables in Watch Window”](#) on page 609, and [“Changing Simulation Format at Runtime”](#) on page 611.
- 5 Saving sessions for replay: see [“Recording Simulation Scenarios/Application States”](#) on page 614.
- 6 Saving output values from sessions: see [“Saving Simulation Output”](#) on page 606.

For more advanced uses of SCADE Suite Simulation with external simulation tools, with external code, with imported types, or in double simulation, see Chapter 13 about [“Running Simulation from External Tools”](#) and Chapter 14 about [“Running Advanced Simulation”](#).

12 /Running Simulation in SCADE Suite

This chapter explains how to configure and run interactive simulation sessions in SCADE Suite.

- [“Setting Simulation Environment”](#)
- [“Setting Project for Simulation Sessions”](#)
- [“Setting Simulation Code for Simulation Sessions”](#)
- [“Running Model Simulation”](#)

Setting Simulation Environment

You can configure the simulation environment as explained below.

To set simulation environment

- 1 Select *Tools > Options* and display **Simulator** options.

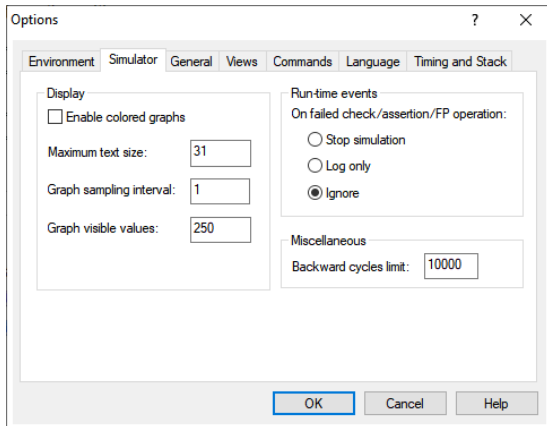


Figure 12.1: Options related to model simulation

2 Set **Display** options to modify how data display in simulation sessions:

- Check **Enable colored graphs** to display colored lines in **Graph** window.
- In **Maximum text size** field, specify a number of character for all text displayed in diagrams during simulation sessions (text longer than specified is truncated).
- In **Graph sampling interval** field, specify the sampling interval (per tick) of the graphs during continuous or multiple step simulation.
- In **Graph visible values** field, set the number of values displayable in Graph window.

3 Set **Run-time events** options to control simulation behavior when violated assertions, failed checks, or erroneous floating-point (FP) operations occur:

- **Stop simulation** on violated assertion, failed check, or erroneous FP.
- **Log only** to log an error without stopping simulation. Logs are displayed in the **Simulator** tab of the **Output** docking window.
- **Ignore** violated assertions, failed checks, or erroneous FP operations (Default).

4 In **Backward cycles limit**, specify the number of steps reachable backward.

Setting Project for Simulation Sessions

Some simulation options are set from project properties and apply only to the related project (ETP file). You can modify the precision of real value display or set connections between SCADE Suite Simulation and external applications for remote control.

To set project properties related to simulation

- 1 Right-click a project file (ETP file) in **Scade View** and select *Properties*.
- 2 Select **Simulation** on the left and set required properties on the right.

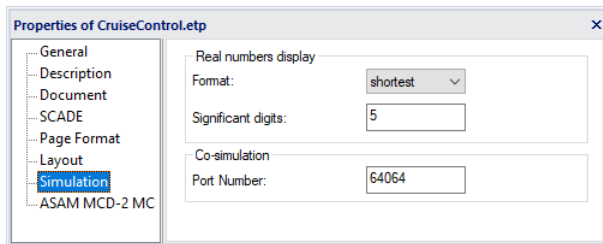


Figure 12.2: Project properties related to simulation

- 3 In **Format** list, select **scientific**, **decimal**, or **shortest** to set the real numbers display. Specify the digit precision used to display and store real values:
 - In **Digits after decimal point**, set the number of digits after decimal point *for scientific or decimal format*: an integer between 1 and 17.
 - In **Significant digits**, set the maximum number of digits *for shortest format*: an integer between 1 and 17. This option affects all reals displayed in **Instances Model Browser**, in **Diagram Simulation Views**, and in **Graph** and **Watch** windows or those dumped in simulation output files. It does not apply to SCADE Test Environment for Host or SCADE Test Model Coverage where precision is fixed. If one defines VTables,

their implementation has priority over options. For details, consult [“Conversion and Comparison Functions for Types”](#) on page 105 in *SCADE Suite Technical Manual*.

- 4 In **Port Number**, specify the port on which SCADE Suite listens for co-simulation with SCADE Suite Simulation. For more details about this option, see [“Using Simulation API for Co-Simulation”](#) on page 687.

Setting Simulation Code for Simulation Sessions

You configure the SCADE Suite KCG-generated code used by SCADE Suite Simulation. You can handle code generation for simulation purposes by applying options preset in configurations (see [“Managing Default or Custom Configurations”](#) on page 117).

- [“Setting Basic Options for Simulation Code”](#)
- [“Enabling Simulator Code Generation”](#)
- [“Setting Simulation Binary Generation”](#)
- [“Enabling Simulation with Graphical Extensions”](#)
- [“Enabling Partial Model Simulation”](#)
- [“Setting Variable Visibility”](#)

Setting Basic Options for Simulation Code

You can specify which operator in the project is used as root operator for simulation.

To select options related to simulation code

- 1 Click **Settings**  in **Code Generator** toolbar to display **General** settings.

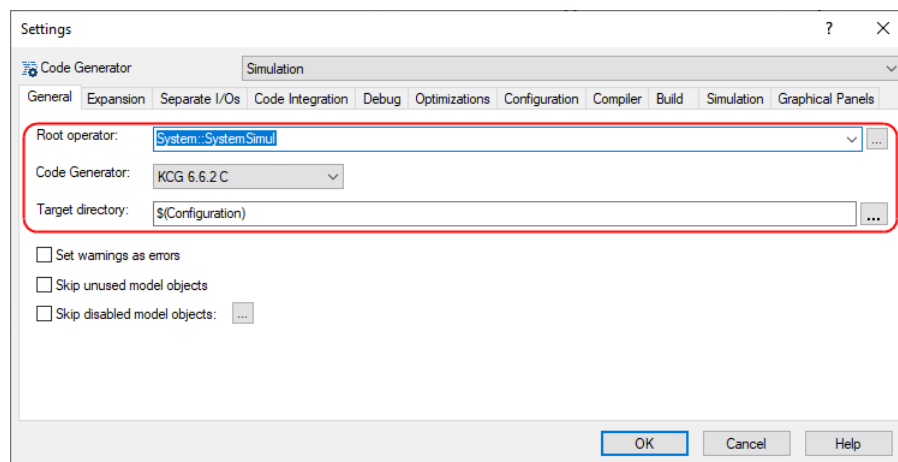


Figure 12.3: General settings for Code Generator

- 2 In **Root operator** list, set the operator to simulate. The value <selected node> generates code for the operator selected in **Scade View**.
- 3 Select the type of code to generate for simulation in **Code Generator** drop-down list. Available optimization and other code generation options are updated accordingly.

Enabling Simulator Code Generation

You must activate the **Simulator** target to produce code that enables SCAD Suite simulation. This target relies on SCAD Suite Code Generator to set the simulation executable required for launching simulation sessions.

To enable and set code generation for simulation

- 1 Click **Settings**  in **Code Generator** toolbar and display **Code Integration**.

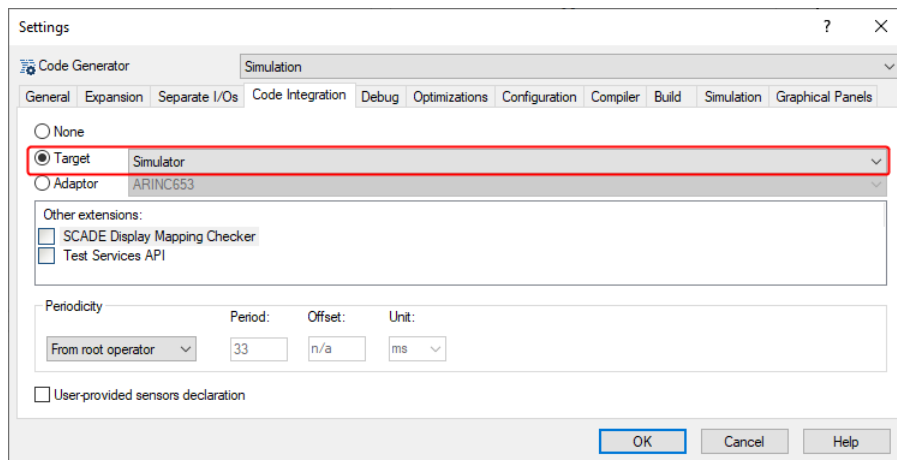


Figure 12.4: Enabling Simulator Target for running simulation sessions

- 2 Set **Target** and select **Simulator** in the corresponding drop-down list.
- 3 Specify the periodicity of code execution for simulation graphs either at root operator level or override root operator periodicity. For details, see [Periodicity](#) option.
- 4 Enable sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked). For details, see [User-provided sensors declaration](#) option.

Setting Simulation Binary Generation

You can set generation options for building simulation code with respect to the compilers in use. These options allow to select a specific compiler, to set compiler specific and compiler independent options, or to limit observable variables in the simulation code to inputs/outputs and probes. Based on compiler and build options, SCADE Suite generates a MakeFile for various Windows OS compilers. Depending on the compiler, you can customize the generation of the MakeFile associated with the simulation binary.

To set code generation for simulation executable

- 1 Click **Settings**  in **Code Generator** toolbar and display **Compiler**.

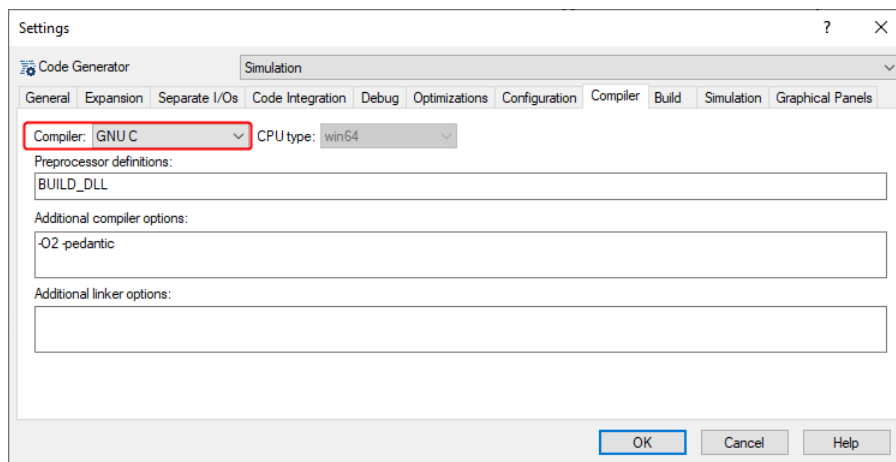


Figure 12.5: Selecting compiler for simulation binary generation

- 2 Select the compiler used to build the simulation code from the **Compiler** drop-down list. The following compilers are available by default:
 - **GNU C:** to enable the build with mingw compiler (for C and Ada code)
 - **VC<X.n>:** to enable the build with Visual C compilers present on the computer where SCADE Suite is installed.

3 CPU type is forced to win64 for simulation binary generation.

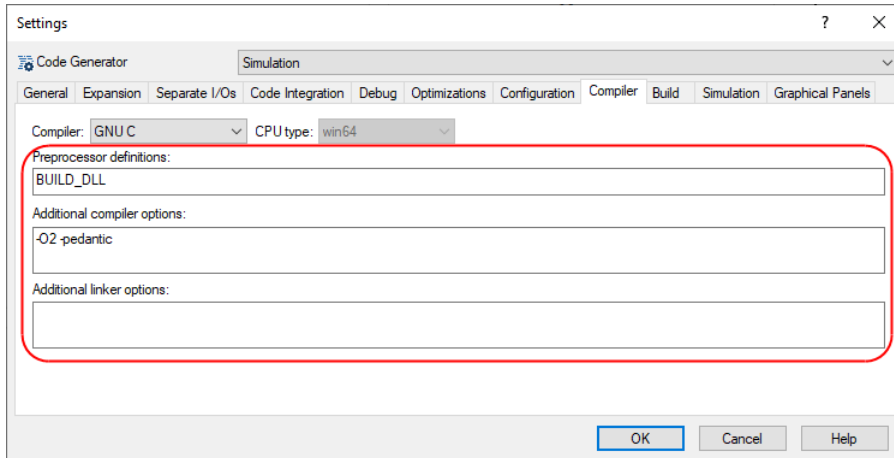


Figure 12.6: Compiler options for simulation sessions

Note

Default options are preset with respect to the selected compiler. For details about compiler and linker options, consult the documentation of the compiler in use.

The compiler and CPU information necessary to build a simulation binary can also be set based on a compiler configuration file as detailed in Appendix D about [“Compiler Configuration for SCADE Suite Code Generation”](#).

- 4** In **Preprocessor definitions**, specify the name of one or more preprocessor macro to be added to the compiler command line.
- 5** In **Additional compiler options**, specify options to customize the compiler command line.
- 6** In **Additional linker options**, specify options to customize the linker command line. Separate list items in all the above fields with a space and use quotes if an item contains spaces. Items must be compliant with the Tcl language format.

7 Display **Build** settings to set options common to all compilers.

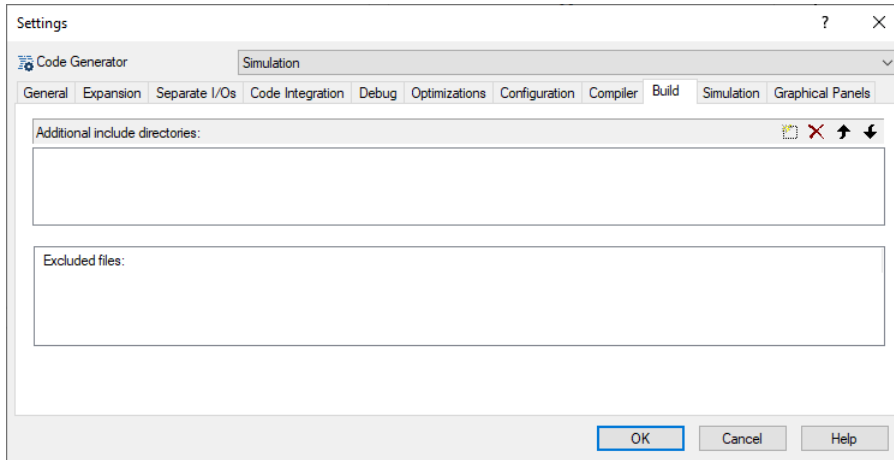


Figure 12.7: Build options for simulation makefile settings

8 In **Additional include directories**, specify one or more directories searched for include file in the simulation build.

Note

Use this option when the model contains imported code and if you need to include directories stored in various locations. The specified directories are used to produce the makefile for the simulation binary necessary to run simulation.

9 In **Excluded files**, check each file name to be ignored during makefile compilation. The list shows all project files set as **Source File for C** or **Source File for Ada** for simulation purposes (see [“To declare external C files for building simulation executable”](#) on page 674 or [“To declare external Ada files for building simulation executable”](#) on page 685).

Enabling Simulation with Graphical Extensions

You can simulate models with user interface prototypes. You set simulation sessions to display simulation reactions with graphical component extensions. Before coupling SCADE Suite variables with an external tool, you need to select the components used for simulation as extension files for the user interface.

To declare graphical component extensions for simulation

- 1 Click **Settings**  in **Code Generator** toolbar and display **Simulation**.

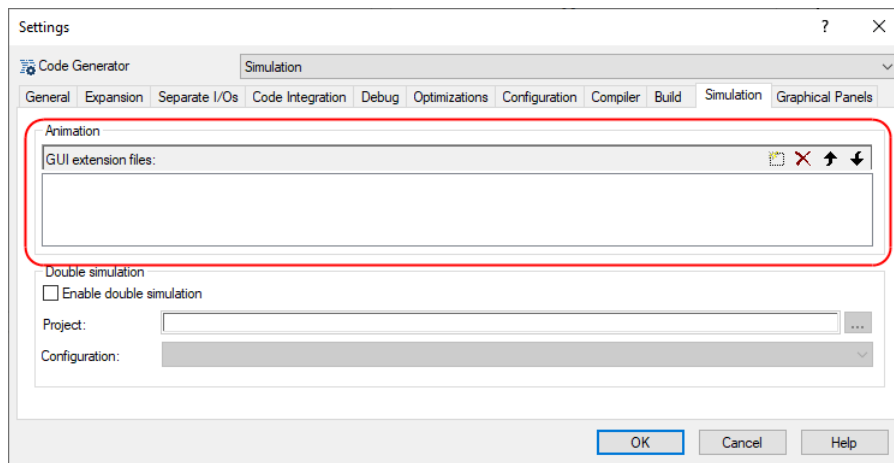


Figure 12.8: Setting graphical component extensions for simulation

- 2 Click  to add a new file entry in the list of extension files.
- 3 Click the browsing button and browse in the **GUI Extension File** window to select an extension file from XML or JAR file packaged beans in project directories.

Enabling Partial Model Simulation

You can always simulate an operator as soon as it is semantically correct (Check reports no error), even if other independent parts of the model or libraries are not correct.

To enable partial simulation of incomplete models

- 1 Click **Settings**  in **Code Generator** toolbar and display **General** settings.

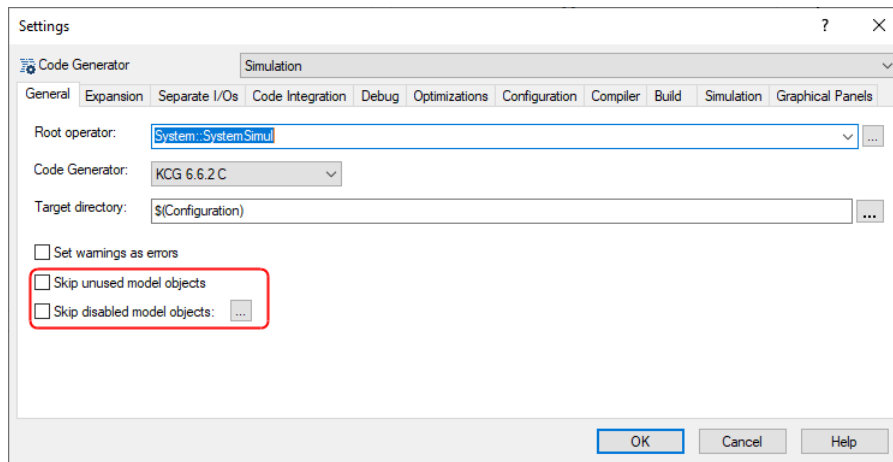


Figure 12.9: Enabling partial simulation on correct operators

- 2 Check **Skip unused model objects** to allow simulation code to consider only the call graph of the selected root operator (if semantically correct).
- 3 Check **Skip disabled model objects** to ignore packages, operators, or diagrams from simulation. Click button and check the elements to disable. If both options are unchecked, model files are passed to SCADE Suite KCG.

Warning

These options are not KCG options and cannot be used for certification purposes because the model is filtered before passing to SCADE Suite KCG Code Generator.

Setting Variable Visibility

You can achieve the highest visibility on variables by changing code generator options. Only the values of observable variables can be displayed for simulation in the **Instances Model Browser** or **Diagram Simulation Views**.

About variable visibility in simulation, see [“Observable Variables and Traceability”](#) on page 571.

To set variable visibility in simulation code

- 1 Click **Settings**  in **Code Generator** toolbar and display **Expansion**.
- 2 Make sure no operator is expanded when **Selected** is set. For details about expansion, see [“Setting Code Expansion Mode”](#) on page 847.

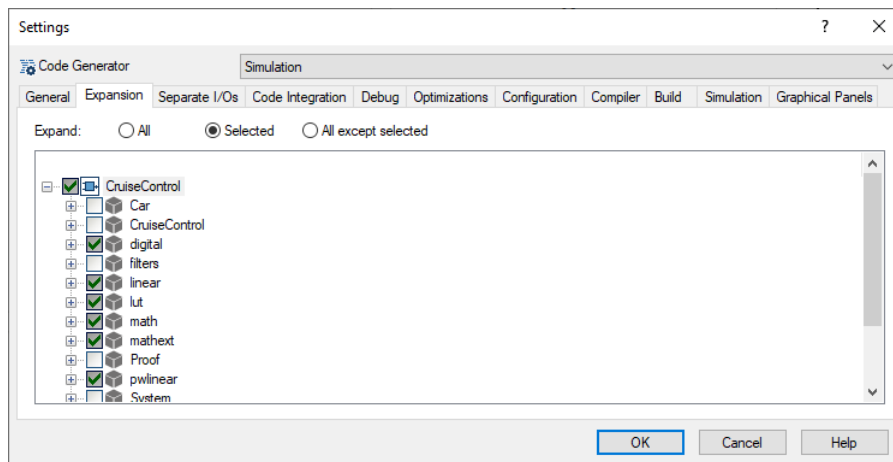


Figure 12.10: Setting no code generation expansion for enhanced variable visibility

- 3 Display **Optimizations** and make sure all options are unselected. The less optimization the more visibility you get. For details, see [“Setting Optimization Options”](#) on page 839 and [“Setting Debug Options”](#) on page 840.
- 4 Display **Configuration** and check **Keep probes in generated code**.

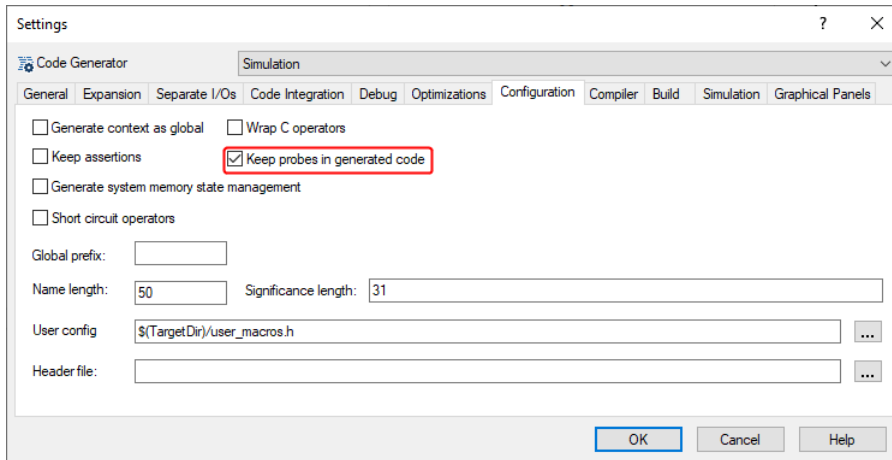


Figure 12.11: Configuration options for simulation observation

[Table 12.1](#) lists the batch commands that correspond to code optimization options that impact on variable visibility.

Table 12.1: Batch commands for variable visibility optimization

Code generation options	Batch command
Debug	-debug
Keep probes in generated code	-probes
No code expansion	-noexp

For more details about SCADE Suite Simulation in batch commands, consult Chapter 6 about [“SCADE Suite Simulation Batch”](#) in *SCADE Suite Technical Manual*.

Running Model Simulation

This section explains how to simulate models with SCADE Suite Simulation. It guides you through the execution and observation of simulation cycles, the graphical representation of system reactions, the use of scenarios, and the use of breakpoints for debugging models.

- [“Launching Simulation Session”](#)
- [“Recording Simulation Scenarios/Application States”](#)
- [“Executing Simulation Cycles”](#)
- [“Setting and Managing Breakpoints”](#)

Launching Simulation Session

Before launching interactive simulation sessions on any part of models, the appropriate configuration preset for simulation code generation must be set as active. In particular, it must specify the root operator and the target for the generation of simulation code, as described in [“Setting Simulation Code for Simulation Sessions”](#) on page 586.

If models to simulate contain imported C code/types, or imported Ada code, users must provide external files as detailed respectively in [“Simulating Models with External C Code”](#) on page 673, [“Simulating Models with Imported C Types”](#) on page 675, and [“Simulating Models with External Ada Code”](#) on page 685.

To launch simulation sessions

- 1 Open a project and select a configuration preset for simulation code generation from **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119.
- 2 If no root operator is specified in the Code Generator settings of the active configuration, select the operator to use for simulation from **Scade View**.
- 3 Click **Run**  in **Code Generator** toolbar. SCADE Suite Simulation interface opens up.

Whenever launching simulation, the **Run** command always checks whether the simulation executable is up-to-date with respect to the SCAD Suite model and its associated data. If it is out of date, a message popup asks whether to rebuild the executable or continue with the existing one despite changes.

Any input older than the simulation executable is reported in a message popup asking whether to rebuild the executable or continue with the existing one despite input file changes.

You can also generate the simulation executable without launching simulation session.

To generate simulation executable only

- In **Code Generator** toolbar, click **Build**  to generate code when necessary and build the simulation executable, or click **Rebuild All**  to clean up the target directory, generate code, and rebuild everything no matter whether model dependencies are out of date or not.

For details, see Chapter 19 about [“Setting and Launching Code Generation”](#).

Executing Simulation Cycles

SCADE Suite Simulation is idle as long as you play no simulation cycle. When simulation cycles are played consecutively (see [“Simulating Model in Continuous Mode”](#) on page 600), it is in running state. It may be interrupted when reaching a breakpoint or when encountering failed assertions, run-time exceptions, or floating-point operation errors.

Whenever breakpoints are reached, Simulator allows locating the design element that raised the breakpoint. If multiple breakpoints occur in the same cycle, the list of all breakpoints is displayed to facilitate their location. For detailed user instructions, see [“Setting and Managing Breakpoints”](#) on page 616.

The values of observable variables may not be defined during a given cycle and are highlighted as inapplicable in the simulation environment by **n/a**.

Note

Possible causes are clocked variables, variables located under non-activated states or conditional blocks, as well as variables inside any operator instantiated as an activate (see [“Designing with Operator Activation”](#) on page 402).

During simulation, you can perform the following tasks:

- [“Simulating Model Step by Step”](#)
- [“Simulating Model in Continuous Mode”](#)
- [“Playing Simulation Scenarios/States”](#)
- [“Setting and Using Bookmarks”](#)
- [“Locating Assertion Violation”](#)
- [“Identifying Check Failure”](#)
- [“Identifying Floating-Point Operation Error”](#)
- [“Analyzing and Locating Run-Time Exceptions”](#)
- [“Saving Simulation Output”](#)
- [“Displaying Variables in Graphical Charts”](#)
- [“Displaying Variables in Watch Window”](#)
- [“Changing Simulation Format at Runtime”](#)
- [“Managing Simulation Execution”](#)
- [“Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time”](#)

Simulating Model Step by Step

You can trigger the reactions of the system step by step after setting variable values on determined model inputs and at specific simulation cycles.

To simulate model step by step

- 1 In **Instances Model Browser** or **Watch window** if you dropped variables to be observed (see [“Displaying Variables in Watch Window”](#) on page 609), select a variable and press F2.
- 2 Enter the new value(s) and press Enter. Alternatively, when the selected variable is an input (of the root operator) or a sensor typed by enumeration type, right-click the variable, select *Value*, and set one of the enumeration values.

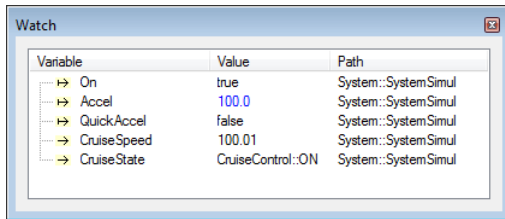


Figure 12.12: Setting variable values before executing simulation cycles

3 Use the following actions to move forward in simulation cycles:

- Click **Step** ► in the **Simulator** toolbar to execute one cycle. The code of the simulated operator is executed once.
- Click **Fast Step** ►► to execute several cycles without displaying intermediate cycles. Set the number of cycles in the drop-down list of the toolbar. The code of the simulated operator is executed n times.

Tip

Modify the information displayed about input/output values during simulation on the Diagram Views. See [“Enabling Display Options in Diagram Views”](#) on page 143.

4 Use the following actions to move back in simulation cycles:

- Click **Back** ◀ to return to the previous cycle.
- Click **Fast Back** ◀◀ to move several cycles backward without displaying intermediate cycles. Set the number of cycles from toolbar.

5 Click **Stop** ■ to stop simulation and exit SCADE Suite Simulation interface.

When saving a scenario as described in [“Recording Simulation Scenarios/ Application States”](#) on page 614, only the cycles that were executed are stored into the simulation input file, thus ignoring cycles after using **Fast** or **Fast Back**. You can set bookmarks and restore bookmarked cycles to display given cycles, see [“Setting and Using Bookmarks”](#) on page 601. You can also display given cycles using the **Simulation** window, see [“Managing Simulation Execution”](#) on page 612.

Simulating Model in Continuous Mode

You execute simulation cycles in continuous mode to observe model behavior based either on simulation scenarios (see [“Recording Simulation Scenarios/Application States”](#) on page 614) or on run-time simulation triggers.

To simulate in continuous mode

- 1 Click **Go** ► in **Simulator** toolbar.
- 2 Reduce/increase the latency value in **Latency** on **Simulation window** to accelerate/slow simulation execution. You can adjust delays between cycles as explained in [“Managing Simulation Execution”](#) on page 612.
- 3 Click **Break** || to pause simulation. Click again **Go** ► to resume simulation.
- 4 Click **Stop** ■ to stop simulation and exit SCADE Suite Simulation interface.

While simulation is running, set values using the **Watch window** (see [“Displaying Variables in Watch Window”](#) on page 609) to trigger and observe system reactions.

Playing Simulation Scenarios/States

You can replay simulation cycles saved from previous simulation sessions in scenarios or states (see [“Recording Simulation Scenarios/Application States”](#) on page 614), using continuous mode or step by step execution. Each action done in continuous simulation mode consumes values from the scenario. Loading a simulation file unload any previously loaded file. The **Simulation** window indicates you switched to scenario input and shows the directory path to the file.

To load and play simulation scenario/state

- 1 Click **Load scenario**  in **Simulator Extension** toolbar.

- 2 Select the file in the opened project directory or browse to locate the file you want to load. The following formats of scenario/state files can be used:
 - **Scenario Files** (.sss) or **Input Files** (*.in): retrieve the simulation input values starting from the selected cycle when the scenario was saved.
 - **State Files** (.sts): reset simulation and initialize simulation in the same state as when the state was saved. STS files are not loadable when simulation stops on breakpoints. Errors occur at loading when STS files are incompatible with: model context; Simulator runtime version; or current hardware (Unix vs. Windows).
 - **Snapshot Files** (.sns): retrieve the application state stored at any moment with input/output values, memories, sensors, and cycle number.
- 3 Play the scenario either by:
 - clicking **Go** ► to launch the scenario in continuous mode.
 - clicking **Step** ► or **Fast Step** ►► to examine scenario forward or **Back** ◀ and **Fast Back** ◀◀ to examine backward in the loaded scenario.

To unload simulation scenarios/states

- Click **Unload scenario**  in **Simulator Extension** toolbar to suppress the scenario from the history of the **Simulation** window and switch back to user input simulation.

Setting and Using Bookmarks

You can load and restore any simulation cycle previously bookmarked (impossible when simulation in continuous mode). Bookmarks are persistent during the active simulation session. SCADE Suite Simulation loads

bookmarked cycle and displays cycle relevant information in the **Simulation window**. Each item in the list is a bookmark and the bracketed number is the step number of the bookmarked cycle.

To bookmark any simulation cycle

- 1 Click **Bookmarks**  in **Simulator Extension** toolbar.

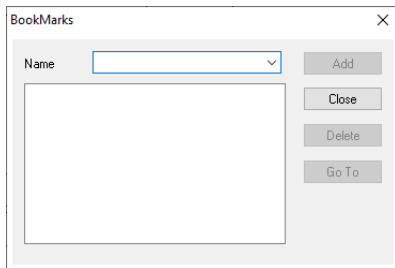


Figure 12.13: Creating bookmarks

- 2 Enter the name of the bookmark in **Name** and click **Add**. Close to validate the bookmark and continue executing simulation step by step, or bookmark other cycles.

To delete a bookmark

- 1 Click **Bookmarks**  in **Simulator Extension** toolbar.
- 2 Select a bookmark in the list and click **Delete**.

To restore bookmarked simulation cycles

- 1 Click **Bookmarks**  in **Simulator Extension** toolbar.
- 2 Select a bookmark in the list and click **Go To**.

Locating Assertion Violation

Assertions result from the use of contracts in the model design. While running simulation cycles on the model, simulation may stop when an assertion is violated. SCADE Suite Simulation checks assertions at run-time, notifies about failed ones, and locates them in the simulated model. Violated assertions can be located interactively in the model. If several assertions are detected, the list of violated assertions displays in a pop-up dialog. By default, failed assertions are ignored. You configure SCADE Suite Simulation upon detecting violated assertions from **Run-time events** options when setting simulation environment.

To locate failed assertions

- 1 Check in the list of failed assertions from the pop-up dialog.
- 2 Double click the name of the assertion to highlight it in the **Scade** tree.

RELATED TOPICS

- [“Creating Contracts: Assume/Guarantee”](#) on page 242
- [“Setting Simulation Environment”](#) on page 583
- [“Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time”](#) on page 613

Identifying Check Failure

Checks result from using SSM : : check commands in simulation scenarios. While running simulation on the model, simulation stops when a scenario check fails. SCADE Suite Simulation can detect failed checks at simulation run-time and help identify them in scenarios. The conditions of failure are identified by error messages. If several failed checks are detected, a dialog gives the list. By default, failed checks are ignored. You set the simulation environment for detecting failed checks from **Run-time events** options.

To identify failed checks

- 1 Display the **Simulator** tab from the **Output** docking window.
- 2 Browse in logged checks and identify check failure condition in error message.

RELATED TOPICS

- [“Simulator Commands”](#) in *SCADE Suite Technical Manual*
- [“Setting Simulation Environment”](#) on page 583
- [“Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time”](#) on page 613

Identifying Floating-Point Operation Error

Floating-point operations result from using arithmetic operations with real or float types. While running simulation on the model, simulation stops when a scenario leads to a floating-point operation error (*i.e.*, overflows, divisions by zero, or invalid operations). SCADE Suite Simulation can detect erroneous floating-point operations at simulation run-time and help identify them in diagrams and views. The conditions of error are identified by error messages. If several errors are detected, a dialog lists the errors. By default, floating-point errors are ignored. You set the simulation environment for detecting erroneous floating-point operation from **Run-time events** options.

To identify floating-point operation errors

- 1 Display the **Simulator** tab from the **Output** docking window.
- 2 Browse in logged errors and identify erroneous floating-point operation in message.
- 3 Check for +Inf, -Inf, or NaN output values displayed in Simulation diagrams or views.

RELATED TOPICS

- [“Setting Simulation Environment”](#) on page 583
- [“Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time”](#) on page 613

Analyzing and Locating Run-Time Exceptions

Run-time exceptions result from events like integer division by zero, stack overflows, access violation. While running simulation on the model, simulation always stops when a scenario raises an exception. SCADE Suite Simulation can help identify them in code. The conditions of exception are identified by error messages.

To analyze and locate run-time exceptions

- 1 Display the **Simulator** tab from the **Output** docking window.
- 2 Browse in logged exception and analyze stack traces.
- 3 Double click on file line displayed under the message to locate the exception in generated code. Debug information, such as function name, source file, line number where the exception is detected, are available only if using VC compiler with debug information compiler option (see [“Setting Compiler and Build Options”](#) on page 853).

RELATED TOPICS

- [“Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time”](#) on page 613

Saving Simulation Output

You can save all output values of the model as well as additional observable variables computed at each simulation cycle during simulation sessions. You need to declare which file SCADE Suite Simulation can use to store simulation output.

You save output files in formats that can be opened in SCADE Suite (.out or .csv files) or in CSV format intended for worksheet applications. Observability files (.obs) can be used to dump additional variables.

To save simulation output in output file

1 Select *Simulation > Output > Output Value File*.

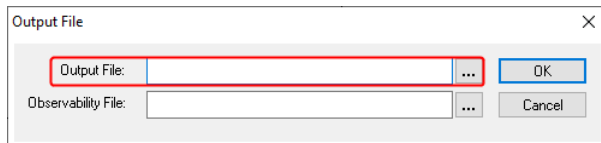


Figure 12.14: Selecting simulation output file

- 2** Click the browsing button of the **Output File** field to specify new output file or overwrite existing output file. Proceed either way:
- Specify the name and location of the output file and select the format (.out or .csv). When opening CSV output files, make sure that the regional settings are set with the same separator generated by SCADE Suite, namely a semicolon (;).
 - Select the output file from a list of existing files.

You can also generate an observability file and use it to dump more output values stored in the specified output file.

To generate observability file from Watch and use with simulation output

- 1 Select *Simulation > Watch List > Generate Observability File from Watch List* to specify the name and location of the observability file to generate.
- 2 Select *Simulation > Output > Output Value File*.

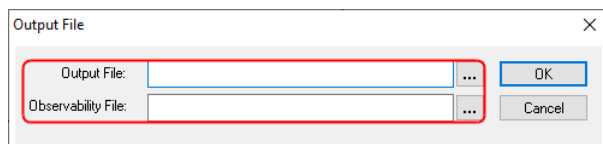


Figure 12.15: Selecting simulation output and observability files

- 3 Select the output file and generated observability file to use:
 - Click browsing button for **Output File** to select the file (.out or .csv).
 - Click browsing button for **Observability File** to select the file (.obs).

While running simulation sessions, you may want to disable the use of output files. By default, the output file you specify is automatically enabled.

To disable output value saving in specified output file

- Select *Simulation > Output > Save To File <OutputFileName>* to uncheck the file name specified for storing output values.

RELATED TOPICS

- [“Simulation Output Files”](#) on page 164 in *SCADE Suite Technical Manual*
- [“Format of OBS Files”](#) on page 166 in *SCADE Suite Technical Manual*

Displaying Variables in Graphical Charts

You can use graphical charts to represent the values of observable variables during simulation. For this purpose, you display the **Graph window** in the simulation environment. The graphs can be copied into documents or printed.

Note

You can manage and fine-tune the sampling rate of observable variables in the graphs: check for Graph sampling interval option in [“Setting Simulation Environment”](#) on page 583.

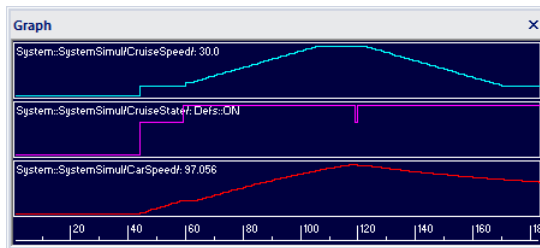


Figure 12.16: Graphs and cursor in the Graph window during simulation

- State Machine transitions and states are represented as Boolean values. The values of states are: Active or Not active. The values of transitions are: Fired or Not fired.
- If and When block branches are represented as Boolean values: Active or Not active.

To add observable variables to the Graph window

- Drag & drop a variable or any sub-element of a structured variable from **Watch window** or **Instances Model Browser** into the **Graph window**, or
- Right-click a variable in **Instances Model Browser** or **Simulation Views** and select *Graph*.

To remove observable variables from the Graph window

- Click the graph of the variable you want to remove and press **Suppr**, or
- Right-click the graph of the variable and select *Delete*.

To copy/paste graphs from the Graph window

- 1 Right-click in the **Graph** window and select *Copy*. All graphs displayed on the window are copied into the clipboard and can be pasted.
- 2 Paste the graph(s) in other application (not possible in continuous simulation).

To print graphs from the Graph window

- 1 Right-click in **Graph window** and select *Print*.
- 2 Set printing properties as needed and click **OK** to print.

To display coordinate information on graphs

- 1 Click on the graph to display a vertical cursor and hold.
- 2 Move the cursor along the graph to display coordinates about selected point.

Displaying Variables in Watch Window

You can select observable variables from the **Instances Model Browser** and display them into the **Watch window**. It allows sorting the variables to observe and trigger system reactions during simulation. The **Watch window** content is restored between sessions.

- [“To add variables into Watch window”](#)
- [“To add variables using instance path”](#)
- [“To copy instance path from variables”](#)
- [“To load variables in Watch window from observability file”](#)
- [“To remove variables from Watch window”](#)
- [“To set values from Watch window”](#)

To add variables into Watch window

- Drag & drop a variable or any sub-element of a structured variable from **Instances Model Browser** into the **Watch window**, or
- Right-click a variable in **Instances Model Browser** or **Simulation Views** and select *Watch*.

To add variables using instance path

- 1 Right-click in the **Watch window** and select *Add Instance Path* to insert an empty line with an editable Path field in the **Watch window**.
- 2 Enter the instance path with the variable name using the same path format as in observability files and validate.

To copy instance path from variables

- 1 Right-click any variable in the **Watch window** and select *Copy Instance Path*.
- 2 Paste the copied instance path wherever needed.

To load variables in Watch window from observability file

- 1 Select *Simulation > Watch List > Load Watch List from Observability File* and browse.
- 2 Select an .obs file and click **Open** to load the list of variables in the **Watch window**.

To remove variables from Watch window

- 1 Select the variable you want to remove in the **Watch window**.
- 2 Press the Delete key from the keyboard.

To set values from Watch window

- 1 Select the variable and press F2.
- 2 Enter the value to use in the next simulation cycle and press Enter. You can set values whether you simulate step by step or in continuous mode.

For simulation modes, see [“Simulating Model Step by Step”](#) on page 598 or [“Simulating Model in Continuous Mode”](#) on page 600.

Changing Simulation Format at Runtime

During simulation, you can change the display format of any variable (inputs/ outputs, hidden, probes, sensors, or local variables) of integer type (int, int8, int16, ...) or array of integer type.

To change display format of variables

- 1 Select a variable or a multi-selection of variables displayed in the **Instances Model Browser**, in the **Watch window**, or in **Simulation Views**, right-click and select *Display*
- 2 Choose format: *Default*, *Binary*, *Octal*, or *Hexadecimal*. The display format of selected variable(s) is automatically updated in all views.

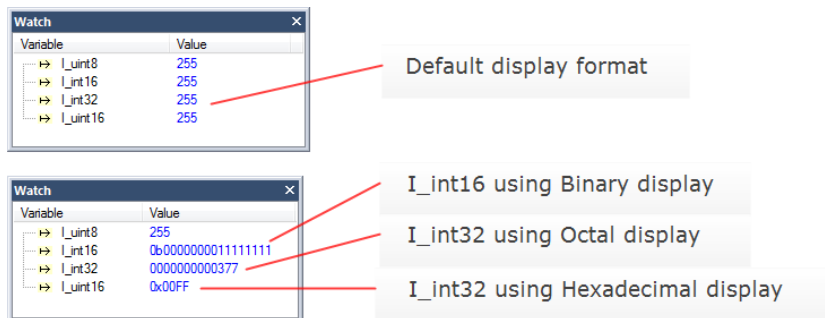


Figure 12.17: Display formats applicable to integer variables in simulation

The selected format is restored between simulation sessions.

Managing Simulation Execution

You can use the **Simulation window** either to request the display of a simulation cycle, to accelerate/slow down cycle delays when in continuous simulation, to change the refresh rate of observed variables (in continuous and a step-by-step simulation modes) or to check simulation history as illustrated in [“Simulation Window”](#) on page 577.

To go to a simulation cycle

- 1 On the **Simulation window**, click in the **Cycle** field.
- 2 Enter the number of the cycle you want to reach and press Enter.

Note

Moving back in cycles unloads any loaded scenario and graphs are updated accordingly. Beyond the last scenario cycle, the scenario is also unloaded.

To increase/decrease delays in continuous simulation execution

- 1 On the **Simulation window**, click in the **Latency** field.
- 2 Enter the delay between two cycles and press Enter, or use the arrow keys to update the delay with regular increments.

Note

The cycle delay set in the Latency field influences the sampling rate of observable variables. Therefore, you can use it to determine how quickly cycle values are updated and graphs refreshed on the Graph window.

To move up/down the refresh rate of observable variables

- 1 On the **Simulation window**, set the rate in ticks from **Refresh interval** field.
- 2 Click up/down arrows to increase/decrease the rate or enter a rate value in the field.

To reset simulation cycles

- Click **Reset**  or double click on any listed item in the **Simulation Window**.

All simulated cycles are canceled and the simulation restarts from the beginning. All changes done during simulation are forgotten and values re-initialized.

Exceptions, Assertions, Checks, FP Operations, Breakpoints at Run-Time

The behavior of SCADE Suite Simulation when encountering exceptions, assertions, checks, floating-point (FP) operations, or breakpoints at run-time is summarized below.

	Simulation Behavior
Exceptions	SCADE Suite freezes simulation and displays error messages. You can no longer enter input values or perform simulation steps. You must reset simulation.
Assertions, Checks, FP Operations	SCADE Suite sets simulation in pause and displays error messages on Simulator output tab. You can reset or continue simulation.
Breakpoints	SCADE Suite blocks simulation before the end of cycle and the column header shows a “+” sign after the cycle information. The cycle is unchanged with last displayed cycle information.

RELATED TOPICS

- [“Locating Assertion Violation”](#) on page 603
- [“Identifying Check Failure”](#) on page 604
- [“Identifying Floating-Point Operation Error”](#) on page 604
- [“Analyzing and Locating Run-Time Exceptions”](#) on page 605
- [“Setting and Managing Breakpoints”](#) on page 616

Recording Simulation Scenarios/Application States

You can record scenarios or states based on simulation steps triggered one after another by setting input values or running continuous simulation. Scenarios and states can be replayed at any time provided SCADE Suite Simulation is idle and no scenario or state is running.

Simulation scenarios can be saved into the following formats:

- **Simulator scenarios** (SSS files): This type of scenarios saves the simulation functions in a command format that is editable. For details, see [“Simulator Commands”](#) on page 81 from *SCADE Suite Technical Manual*.
- **Input scenarios** (IN files): This type of scenarios saves all input values for each cycle in a format that is easily editable.

Application states can be saved into the following formats:

- **Snapshots** (SNS files): This type of files stores application states in textual format at any given step in simulation session. Step-by-step simulation is not possible but the state is retrieved immediately. Snapshots can be reloaded independently from the compiler selection and compiler options.
- **States** (STS files): This type of files saves application states in binary format at any given step in simulation session. States cannot be used in double simulation or when simulation raised an exception. States can be replayed only with the same compiler and the same compiler options. Application states are dependent from the model and code generation options in use.

Before recording scenarios and states, it is necessary to provide VTable definitions in the following cases:

- When models contain imported types, the use of SSS, IN, and SNS files requires the definition of VTables as detailed in [“Conversion and Comparison Functions for Types”](#) on page 105 from *SCADE Suite Technical Manual*.

- When models contain imported operators, the use of SNS files requires the definition of VTables as detailed in [“Conversion Functions for Imported Operators”](#) on page 127 from *SCADE Suite Technical Manual*.

For details about scenario/state files and their format, consult [“Simulation Input Files”](#) on page 160 in *SCADE Suite Technical Manual*.

To record a scenario or state

- 1 Run a simulation session as detailed in [“Simulating Model Step by Step”](#) on page 598.
- 2 Click **Save scenario**  in **Simulator Extension** toolbar to store all input values corresponding to each simulation cycle.
- 3 In **Save As** dialog, enter the name of the file in **File name**.
- 4 Select the format of the scenario or state from **Save as type** drop-down:
 - **Scenario Files (*.sss)**
 - **Input Files (*.in)**
 - **Snapshot Files (*.sns)**
 - **State Files (*.sts)**

Note

For saving snapshot files, it is necessary to enable the Test Service API in the Code Generator settings of the configuration in use for simulation. For details, see [“Producing Code for Testing on External Platforms”](#) on page 46 in *SCADE Test User Manual*.

-
- 5 In the **Start cycle** field, set from which cycle you want to save the values of the scenario (available with scenario and input file types only).

You can also use scenario and state files in batch mode. For instructions, consult Chapter 6 about [“SCADE Suite Simulation Batch”](#) in *SCADE Suite Technical Manual*.

Setting and Managing Breakpoints

You use breakpoints while simulating models to interrupt simulation on specific design elements of the model or under given conditions. You define and manage [breakpoints](#) during simulation sessions.

- [“Setting Breakpoints on Control Flow Elements”](#)
- [“Setting Breakpoints on Data Flow Elements”](#)
- [“Defining Advanced Breakpoints”](#)
- [“Modifying Breakpoint Definition”](#)
- [“Enabling/Disabling Breakpoints”](#)
- [“Suppressing Breakpoints”](#)

Setting Breakpoints on Control Flow Elements

While in simulation, you can set breakpoints on control flow elements like states, transitions, if conditional block branches, and match.

To set breakpoints on control flow elements

- Right-click the control element on which to set the breakpoint from the **Instances Model Browser** or **Diagram Simulation Views** and select *Insert Breakpoint* (F9 in **Instances Model Browser** only). The breakpoint is represented as red dot on the selected element.

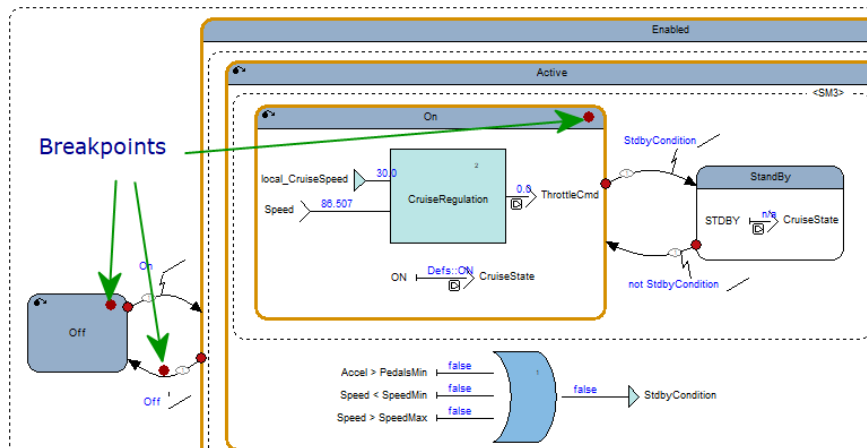


Figure 12.18: Breakpoint set on control flow elements

Setting Breakpoints on Data Flow Elements

While in simulation, you can set breakpoints on any observable variables as defined in [“Observable Variables and Traceability”](#) on page 571.

To set breakpoints on data flow elements

- 1 Right-click the data flow element on which to set the breakpoint from **Instances Model Browser** or **Diagram Simulation Views** and select *Insert Breakpoint* (F9 in **Instances Model Browser** only).

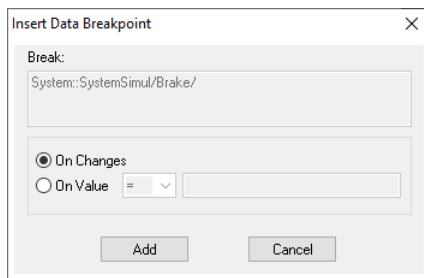


Figure 12.19: Inserting data breakpoint

- 2 In **Insert Data Breakpoint** window, set the breakpoint by selecting one of the following radio buttons:
 - **On Changes:** to enable breakpoint whenever the value of the selected variable changes.
 - **On Value:** to enable the breakpoint whenever the selected variable must be compared with a given value. Select the comparison formula to apply from the drop-down list.
- 3 Click **Add** to create the breakpoint. The definition of the breakpoint is added in the **Breakpoints** list below.

Defining Advanced Breakpoints

You can define advanced breakpoints based on multiple criteria with the help of Tcl expressions. One can write an expression to stop simulation when a state is reached at cycle 64 and when a given variable is higher than 100.

To define advanced breakpoints

- 1 Select *Simulation > Breakpoints* (Alt+F9) to open the **Breakpoints** window and display **Advanced** options. Name the breakpoint in **Breakpoint Name**.

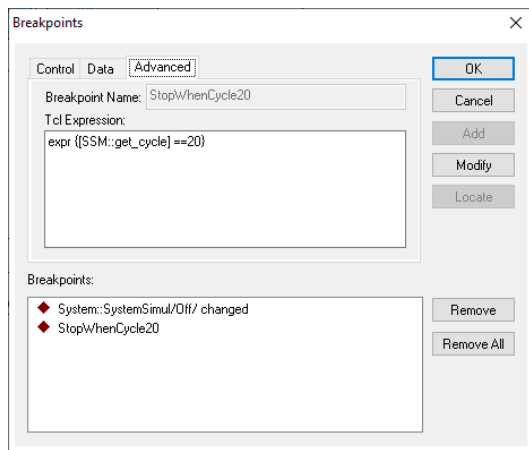


Figure 12.20: Defining advanced breakpoint for stop conditions

- 2 Express the criteria/conditions of the breakpoint in **Tcl Expression** using Tcl language. Stop conditions are expressions that return a value comparable to a boolean (0, 1, "0", "1"). Typical definitions for stop conditions are:
 - Stop on value: syntax is `expr {[SSM::get Output1] > 10}` to express that simulation stops when Output1 is higher than 10.
 - Stop on cycle: syntax is `expr {[SSM::get_cycle] == 20}` to express that simulation stops when it reaches the 20th cycle.
- 3 Click **Add** to create the breakpoint. The definition of the breakpoint is added in the **Breakpoints** list below.

Modifying Breakpoint Definition

You can modify the definition of the breakpoints you set on data flow elements or the expression of any advanced breakpoint from the **Breakpoints** window.

Breakpoints set on control flows can only be toggled between enabled and disabled or simply removed.

To modify breakpoint definition for dataflow

- 1 Select *Simulation > Breakpoints* (Alt+F9) to open the **Breakpoints** window.
- 2 Select a breakpoint from the bottom list to display the definition of the selected breakpoint in **Data** options.

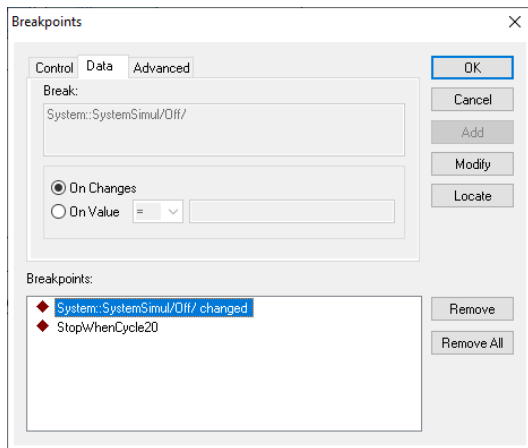


Figure 12.21: Selecting data flow breakpoint to be modified

- 3 Make all necessary changes to the definition and click **Modify** to record the changes. The definition of the breakpoint is edited accordingly in the **Breakpoints** list below.

Enabling/Disabling Breakpoints

You can toggle breakpoints on control flow items as enabled or disabled during simulation sessions directly from the **Instances Model Browser** or **Diagram Simulation Views**. Breakpoints on data flow items can only be enabled/disabled from the **Instances Model Browser**. You may also use the **Breakpoints** window to manage easily the breakpoint enabling/disabling.

Tips

You can toggle breakpoints from various interface contexts:

- Select any control or data item with breakpoint icon in the Instances Model Browser and press **Ctr l+F9**.
- Right-click any control or data item with breakpoint icon in the Diagram Simulation Views and select *Enable/Disable Breakpoint* contextual menu.
- From the list of set breakpoints in the Breakpoints window: click the icon in front of breakpoint definitions to switch breakpoint status.

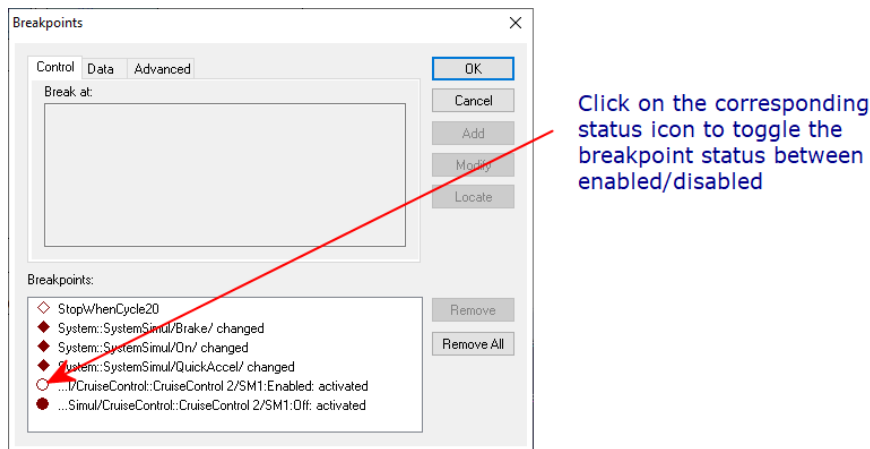


Figure 12.22: Enabling/disabling breakpoints in Breakpoints window

[Table 12.2](#) presents the breakpoint representation when enabled or disabled.

Table 12.2: Icons representing the enabled/disabled status of breakpoints

Icon	States
●	Control flow breakpoint enabled
○	Control flow breakpoint is disabled
◆	Data flow breakpoint enabled
◇	Data flow breakpoint is disabled

Advanced breakpoints are represented with the same icons as data flow breakpoints.

To enable/disable breakpoints

- In the **Instances Model Browser** or **Diagram Simulation Views**, right-click the element with the breakpoint to edit and select:
 - *Enable Breakpoint* (Ctrl+F9 in **Instances Model Browser** only)
 - *Disable Breakpoint* (Ctrl+F9 in **Instances Model Browser** only)

Suppressing Breakpoints

Breakpoints can be suppressed from the models any time during simulation using the breakpoints management window or contextual menus.

To suppress breakpoints

- 1 Select *Simulation > Breakpoints* (Alt+F9) to open the **Breakpoints** window.
- 2 Select the breakpoint you want to suppress from the list and click **Remove** or Click **Remove All** to suppress all breakpoints set in the model.

To suppress breakpoints contextually

- Proceed either way:

- Right-click the element in **Instances Model Browser** or **Diagram Simulation View** and select *Remove Breakpoint*.
 - Select a control or data flow element in **Instances Model Browser** and press F9 to remove the breakpoint
-

Tip

On control flow elements, press F9 to toggle breakpoints insertion/removal.

13 /Running Simulation from External Tools

This chapter explains how you configure and set the projects for co-simulation sessions from external tools.

- [“Co-Simulation Principles”](#)
- [“Simulating Models with Graphical Panels”](#)
- [“Simulating Models in FMI-Compliant Tools”](#)
- [“Simulating Models in Twin Builder Environment”](#)
- [“Simulating Models in Simulink Environment”](#)
- [“Simulating Models in VeriStand Environment”](#)

Note

For configuring SCADE Suite co-simulation capabilities with other external applications, you have access to the SCADE Suite Simulation API functions. For details, see [“Using Simulation API for Co-Simulation”](#) on page 687.

Co-Simulation Principles

SCADE Suite co-simulation capabilities enable the simulation of SCADE Suite models from external simulation applications.

You have access to built-in co-simulation with various development tools and environments as described hereafter in this chapter.

You can also extend other external applications to take remote control over SCADE Suite Simulation using a set of simulation API C functions.

Note

For API usage, see [“Using Simulation API for Co-Simulation”](#) on page 687.

In SCADE Suite, co-simulation can operate in either of the following modes:

- *Black-box co-simulation*: the aim is to encapsulate the code generated from SCADE Suite models into dedicated interface layers required by the external application and execute the SCADE Suite-generated code directly within the external application.
- *White-box co-simulation*: the aim is to generate SCADE Suite simulation models (DLL) and a separate adaptation layer for the external application. The adaptation layer forwards model inputs to SCADE Suite Simulation and sends model outputs to the external application through simulation API functions.

Note

Such co-simulation modes have specifics which are illustrated and detailed for each tool and environment in their dedicated documentation.

Simulating Models with Graphical Panels

SCADE Suite allows simulating models with graphical panels to control and display simulation reactions graphically. Graphical panels are designed either in Rapid Prototyper or SCADE Display®. The principles of simulation with graphical panels rely on the inputs/output (I/O) connections between the root operator of the SCADE Suite model and graphical panels inserted in the project as explained in [“Developing Control and Display Software”](#) on page 302.

- [“Workflow for Simulation from Graphical Panels”](#)
- [“Managing Model Simulation from Graphical Panels”](#)
- [“Enabling Simulation with Graphical Panels”](#)

Workflow for Simulation from Graphical Panels

The workflow for developing control and display software using SCADE Suite and SCADE Display is described in the following figure:

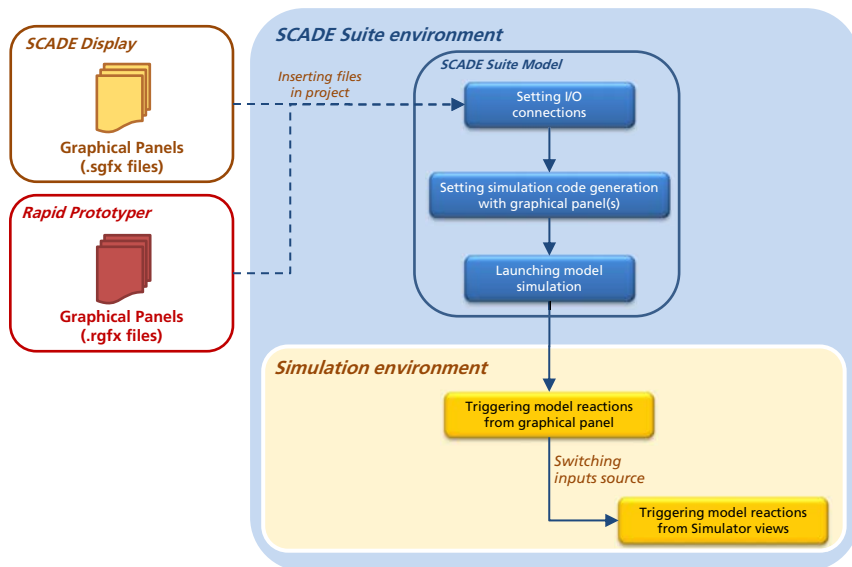


Figure 13.1: General workflow for developing control and display software

This workflow consists of successive tasks that you can iterate as you need. It can follow several phases as described below:

- 1 Inserting graphical panels in SCADE Suite project: see [“Inserting Graphical Panels in Project”](#) on page 305.
- 2 Connecting the SCADE Suite I/Os to the graphical panels I/Os: see [“Connecting Operator and Graphical Panel Interfaces”](#) on page 306.
- 3 Setting simulation code generation with graphical panels integration: see [“Enabling Simulation with Graphical Panels”](#) on page 626.
- 4 Launching sessions and stimulating model reactions from graphical panels: see [“Running Model Simulation”](#) on page 596 and [“Managing Model Simulation from Graphical Panels”](#) on page 628.

For instructions on setting code generation for display applications embedding logic and graphic components for target integration, see [“Setting Code Integration Capabilities”](#) on page 849.

For instructions on reporting about both SCADE Suite model and SCADE Display/ARINC 661 specification: see [“Reporting Suite and Display/ARINC 661 Joint Design”](#) on page 45 in *SCADE LifeCycle User Manual*.

Enabling Simulation with Graphical Panels

You set simulation sessions to control and display simulation reactions in graphical panels.

To enable simulation with graphical panels

- 1 Select a configuration preset for simulation code generation from **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119.
- 2 Click **Settings**  in **Code Generator** toolbar and display **Graphical Panels**. These settings list referenced projects containing graphical panels.

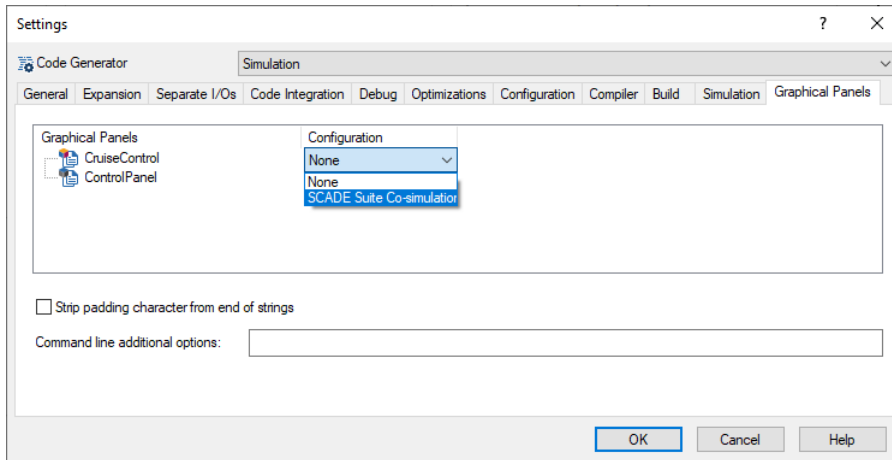


Figure 13.2: Selecting configuration for simulation code generation

- 3 For each SCADE Display or Rapid Prototyper graphical panel included in the project, select which configuration to use for generation. Select **None** if you do not want to use the graphical panel in co-simulation.

Note

For information on SCADE Display configurations, see [“Setting SCADE Suite Co-Simulation Configuration”](#) on page 466 in *SCADE Display User Manual*. For information on Rapid Prototyper configurations, consult *Rapid Prototyper User Manual*.

- 4 Check **Strip padding character from end of strings** to prevent copying padding characters added at the end of strings in the generated code. Padding characters need to be removed for proper display of arrays in SCADE Suite Simulation interface.

Note

This option applies for co-simulation, co-generation or standalone executable generation. For all the other options related to the configuration and the optimization of the simulation environment and the simulation executable you generate, see [“Setting Simulation Environment”](#) on page 583.

Managing Model Simulation from Graphical Panels

During simulation sessions, you can change graphical panels resolution and switch into full-screen mode. Moreover, by default, connected inputs can be changed only from graphical panels. If needed, you can switch input source to set connected inputs values from the SCADE Suite Simulation views.

- [“To switch Full Screen mode”](#)
- [“To switch inputs source during simulation with graphical panels”](#)

To switch Full Screen mode

- 1 Click a graphical panel to set it as the active window.
- 2 Press F11 alternately to switch to:
 - full screen without distortion
 - complete full screen with distortion
 - exit full screen

To switch inputs source during simulation with graphical panels

- Click alternately **Use Graphical Panels Inputs**  from **Simulator** toolbar.

When disabling graphical panels as inputs source, interaction between model and all graphical panels is stopped. You can set value for all inputs listed in the **Instances Model Browser** from the Simulator interface as explained in [“Running Model Simulation”](#) on page 596.

Simulating Models in FMI-Compliant Tools

You can use SCADA Suite Code Integration Target for FMU to generate simulation unit for SCADA Suite model simulation in FMI-compliant tools. This allows to execute SCADA Suite models within external tools using FMI (Functional Mock-up Interface) standard.

Note

FMI is a standard for coupling two or more simulation tools in a simulation environment. Models implementing the FMI interface are called *Functional Mock up Units* and can be run in any simulation environment supporting the FMI interface. For detailed information, visit [FMI standard site](#).

SCADA Suite solution for FMI-compliant tools enables black-box and white-box simulation capabilities.

- [“Code Generation Workflow for FMU Simulation”](#)
- [“Exporting SCADA Suite Model as FMU”](#)
- [“Setting Sensors as Calibration Parameters”](#)
- [“Importing Generated Files in FMI-Compliant Tools”](#)
- [“Setting Code Generator for FMU Simulation”](#)
- [“Running Simulation with Distributed FMUs”](#)

Code Generation Workflow for FMU Simulation

You can export SCADA Suite models as Functional Mock up Units (FMU). After developing SCADA Suite models, you need to specify various FMU-related properties to identify the FMU to be created from the SCADA Suite model.

Note

As the FMI interface accepts only scalar variables and arrays of scalar variables (only FMI 3.0), each compound (array or structure) inputs/outputs of the SCADA Suite model is split automatically during FMU generation into multiple scalar variables or arrays of scalar variables (only FMI 3.0) corresponding to their components.

As illustrated below, you use SCADE Suite Code Integration Target for FMU to generate all FMI-related files and SCADE Suite KCG code necessary to run the simulation of SCADE Suite models within FMI-compliant tools.

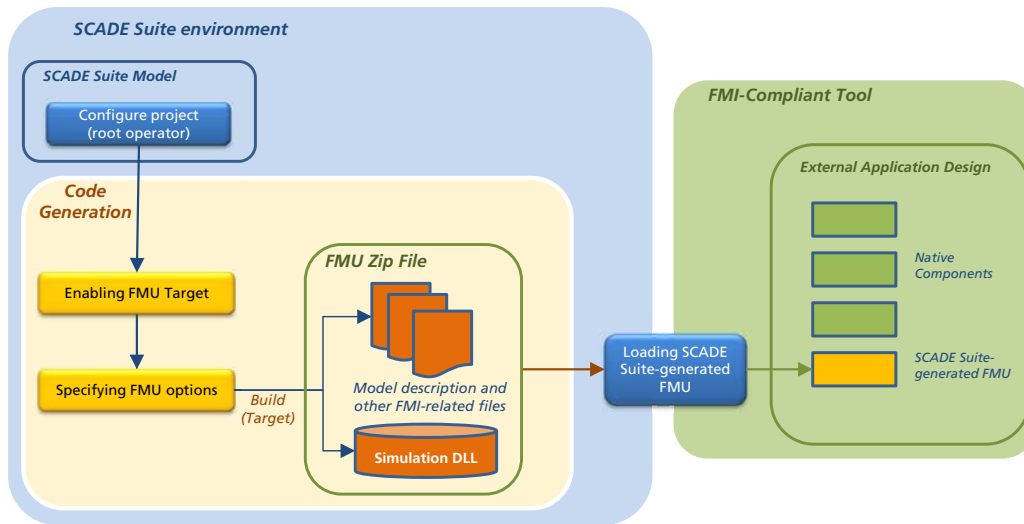


Figure 13.3: Code generation workflow for FMU simulation

In *black-box simulation*, the C code from SCADE Suite model is directly embedded in the generated FMU.

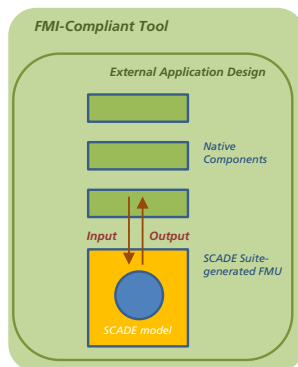


Figure 13.4: FMU black-box simulation

In *white-box simulation*, the SCADE Suite model is executed in SCADE Suite Simulation; and the FMU forwards the I/Os to SCADE Suite Simulation through a network connection. This mode enables model debugging with the full power of SCADE Suite simulation where you can observe model reactions graphically at each simulation cycle or set breakpoints and stop conditions.

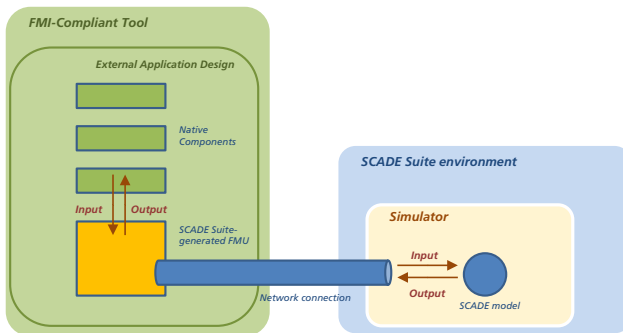


Figure 13.5: FMU white-box simulation

Notice that in both white-box and black-box simulation, the SCADE Suite cyclic function of the operator is called periodically at the user-specified period.

Setting Sensors as Calibration Parameters

The sensors you define in SCADE Suite projects can be set as parameters for FMU simulation. You can set the code integration properties of a sensor from **Scade View**.

To edit the code integration properties of a sensor

- 1 Right-click a sensor in **Scade View** and select *Properties*.
- 2 Select **Code Integration** and set required properties on the right.

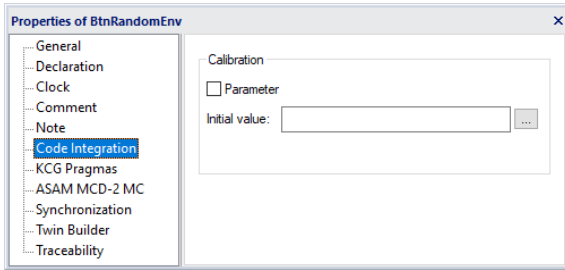


Figure 13.6: Declaration properties of a sensor for FMU simulation

- 3 Check **Parameter** to set the selected sensor as a parameter.
- 4 Specify an **Initial value** for the parameter in the edit box or click  to open the **Initial Value for Calibration** window then click **Compute Default** to set a default value according to the type of the selected sensor as follows:

Type	Value
kcg_[u]intN signed/unsigned<<N>>, imported type with constraint integer/signed/unsigned	0
kcg_floatN, imported type with constraint float	0.0
kcg_char	' '
kcg_bool	false
enum	1st enumeration element
array ¹	[v1, v2, ...]
struct ¹	{elem1 : v1, elem2 : v2, ...}
generic, imported type with no constraint or constraint numeric	?

1. Structure and array inputs/outputs/sensors are decomposed into scalar elements in FMI 1.0 and 2.0, and into scalar elements or array of scalar elements in FMI 3.0.

Setting Code Generator for FMU Simulation

For FMU Simulation, you enable code generation to export SCADA Suite model as FMU and set options from SCADA Suite Code Generator **Settings**.

Note

It is recommended to create a custom configuration with FMU specific options. For more information about managing configurations in SCADA Suite, see [“Managing Default or Custom Configurations”](#) on page 117 in *SCADA Suite User Manual*.

To set code generation for FMU simulation

- 1 Click **Settings**  in **Code Generator** toolbar. Select a configuration set for FMU from the drop-down list in the upper right corner.
- 2 Display **Code Integration**, set **Target**, and select **FMU** in the list.

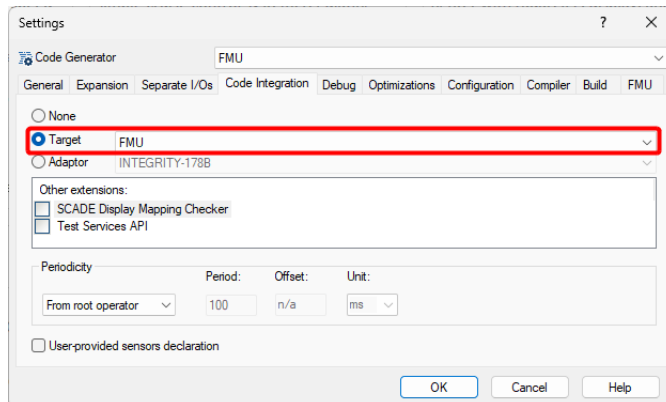


Figure 13.7: Enabling code generation for FMU generation

- 3 Specify the periodicity at which the FMU calls the SCADA operator cyclic function: set [Periodicity](#) either **From root operator** level or **Override** root operator periodicity.
- 4 Set sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked) - see [User-provided sensors declaration](#).

5 Display **FMU** to set target-specific options.

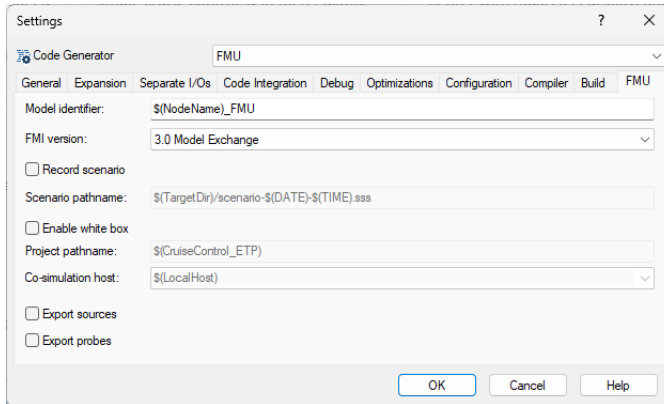


Figure 13.8: Specifying FMU export options

6 Specify the name of the FMU in the **Model identifier** field.

7 Specify the FMI standard to use from the **FMI version** drop-down list:

- 1.0 Model Exchange
- 2.0 Model Exchange
- 3.0 Model Exchange
- 1.0 Co-Simulation
- 2.0 Co-Simulation
- 3.0 Co-Simulation

8 Check **Record scenario** for recording a scenario (. sss file) during simulation. Specify the location of the file in **Scenario pathname** with a relative pathname and/or a pathname containing environment variables, like “ . . /scn1 . sss”, “\$(MYVAR)/scn1 . sss”. Pathname resolution varies as detailed below:

- *On the machine where the FMU is generated:* \$(ProjectDir) and \$(TargetDir) variables are resolved respectively as absolute paths of project directory and target directory; \$(Configuration) variable is resolved as configuration name. Relative paths and environment variables are not resolved.
- *On the machine where the FMU is executed:* \$(DATE) and \$(TIME) variables are resolved before environment variables. If the resulting path is relative, it is made absolute using the DLL path as base. In Twin

Builder, for a project at C:\temp\Project1.aedt, the DLL is at C:\temp\Project1.aedtresults\TwinBuilder\temp\49524C~1\binaries\win64\<model>.dll. To save the scenario in the same directory as the Twin Builder project, the relative path to use is ../../../../scenario.sss.

- 9 Check **Enable white box** to run the exported model in white-box simulation or uncheck this option to run black-box simulation. Specify related options:
- Specify the location of the SCADE Suite project to use in white-box simulation in **Project pathname**. Use a relative project pathname and/or a pathname containing environment variables, like “../myProject.etp”, “\$(MYVAR)/myProject.etp”, “\$(myProject.etp)”.
-

Note

- *For launching SCADE Suite*, the generated FMU uses the %SCADE% environment variable if existing or the most recent SCADE version found in Windows registry.
 - *For opening the project*, the generated FMU uses the specified Project pathname. Environment variables in the pathname are resolved first. If a variable does not exist, the FMU opens the project with a local path. If the resulting path is relative, it is made absolute using the DLL path as base. DLL location depends on the tool. In Twin Builder, for a project at C:\temp\Project1.aedt, the DLL is at C:\temp\Project1.aedtresults\TwinBuilder\temp\49524C~1\binaries\win64\<model>.dll. To use the SCADE Suite project in the same directory as the Twin Builder project, the relative path to use is ../../../../myProject.etp.
-
- Specify the IP address of the host running SCADE Suite in **Co-simulation host**. Do not change the default value of the host if running co-simulation with SCADE Suite and the FMI-compliant tool on the same local host. Simulation with FMI-compliant tools can be launched from another host able to connect to the specified IP.

- 10 Check **Export sources** to produce an FMU file with all model sources (.h and .c files) in a separate folder. Leave unchecked to produce an FMU file without model sources.
- 11 Check **Export probes** to connect all the probes of the model to FMU outputs. Check also the **Keep probes in generated code** option on **Configuration** to ensure all probes are part of the output context.
- 12 Display **Configuration** and check **Generate system memory state management** to activate memory state management in FMUs. Since the option activates the -state_vector KCG option, users have to define the kcg_load_SV_* and kcg_save_SV_* functions for imported operators having memory.
- 13 Display **Compiler** to set **CPU type** for generating 32-bits or 64-bits FMU and set compiler-specific options. Display **Build** for specific build options. For details, see [“Setting Compiler and Build Options”](#) on page 853.

Exporting SCADE Suite Model as FMU

Once code generation is set, you can use it to export the SCADE Suite model into a DLL contained into an FMU file.

To export SCADE Suite model as FMU

- 1 Select the configuration set for FMU export from **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119.
- 2 Click **Build**  from **Code Generator** toolbar.

For both black-box and white-box simulation modes, SCADE Suite generates in the default folder (same name as configuration name) a zip file named <FMU_Name>.fmu and containing all required FMI-compliant files.

The FMU zip file contains in particular:

- the DLL of the SCADE Suite model **for FMI 1.0 and 2.0** in:
 - `binaries/win32/<FMU_Name>.dll` for 32-bits DLLs
 - `binaries/win64/<FMU_Name>.dll` for 64-bits DLLs
- the DLL of the SCADE Suite model **for FMI 3.0** in:
 - `binaries/x86-windows/<FMU_Name>.dll` for 32-bits DLLs
 - `binaries/x86_x64-windows/<FMU_Name>.dll` for 64-bits DLLs
- the interface description of the FMU in `modelDescription.xml` copied from `<Model_Identifier>.xml`.
- the model sources (.c and .h files used to compile the DLL) in a separate folder if [Export sources](#) option is set.

The FMU component description contains:

- FMI input variables corresponding to inputs and sensors in SCADE Suite model
- FMI output variables corresponding to outputs in SCADE Suite model
- A `_period_` parameter for changing the period of the component during the initialization phase in FMI-compliant tools and overriding the period initialization set in SCADE Suite.

Read more details below:

- [“About FMU Component Periodicity”](#)
- [“About Imported Operators for FMU Components”](#)
- [“About FMU Components and their Type”](#)

ABOUT FMU COMPONENT PERIODICITY

FMU components exported from SCADE Suite allow to change the period of the component during the initialization phase of the FMU. The `_period_` parameter can override the period set in SCADE Suite Code Integration settings (see [“Setting Periodicity of Code Execution”](#) on page 852). Users or FMU master modify this period value by calling the following functions: `fmiSetReal()` | `fmi2SetReal()` | `fmi3SetFloat64()`.

ABOUT IMPORTED OPERATORS FOR FMU COMPONENTS

When generating FMU components in FMU 3.0 or 2.0, the FMU components support the ability to save and restore memory state. When the **Save and restore memory state** option is activated, for imported node operators, in addition to defining the `init`, `reset`, and `cycle` functions, users must define the following functions: `kcg_save_SV_<ImportedOp>` and `kcg_load_SV_<ImportedOp>`.

ABOUT FMU COMPONENTS AND THEIR TYPE

FMU components are exported from SCADE Suite inputs/outputs/sensors depending on their types. The correspondence of scalar types, array types, and structure types between SCADE Suite and FMU is described below:

FMI Version	SCADE Suite Variable Types	Scalar FMI Variables
1.0 and 2.0	InputA of type int8 InputA of type uint8 InputA of type uint16 InputA of type int32 InputA of type uint32 InputA of type int64 InputA of type uint64	InputA of type Integer
	InputB of type bool	InputB of type Boolean
	InputC of type float32	InputC of type Real
	InputC of type float64	
	InputA of type MyArray:float32^3	InputA[0] of type Float32 InputA[1] of type Float32 InputA[2] of type Float32
	OutputB of type MyStruct {field1:int32, field2:float32, field3:int32^3}	OutputB.field1 of type Int32 OutputB.field2 of type Float32 OutputB.field3[0] of type Int32 OutputB.field3[1] of type Int32 OutputB.field3[2] of type Int32

FMI Version	SCADE Suite Variable Types	Scalar FMI Variables
3.0	InputA of type float32	InputA of type Float32
	InputA of type float64	InputA of type Float64
	InputA of type int8	InputA of type Int8
	InputA of type uint8	InputA of type UInt8
	InputA of type int16	InputA of type Int16
	InputA of type uint16	InputA of type UInt16
	InputA of type int32	InputA of type Int32
	InputA of type uint32	InputA of type UInt32
	InputA of type int64	InputA of type Int64
	InputA of type uint64	InputA of type UInt64
	InputA of type bool	InputA of type Boolean
	InputA of type MyArray: float32^3	InputA of type Float32[3]
	OutputB of type MyStruct {field1:int32, field2:float32, field3:int32^3}	OutputB.field1 of type Int32 OutputB.field2 of type Float32 OutputB.field3 of type Int32^3

Importing Generated Files in FMI-Compliant Tools

After importing SCADE Suite-generated FMU in FMI-compliant tool, use SCADE Suite operators as any other components and run [black-box](#) or [white-box](#) simulation. Based on [FMU settings](#), FMI-compliant tools start black-box or white-box simulation.

- *In both modes*, you import the SCADE Suite-generated FMU into the design and run the simulation capabilities of the FMI-compliant tool.
- *In white-box simulation*, starting simulation with FMI-compliant tools launches simulation. If no SCADE Suite session is running, a new one is opened; otherwise, simulation runs in the open session. Stopping simulation from the FMI-compliant tool closes the SCADE Suite session.

Note

Refer to the documentation of the FMI-compliant tool for more instructions about simulation operations. Consult Chapter 12 about [“Running Simulation in SCADE Suite”](#) in *SCADE Suite User Manual* for more instructions about SCADE Suite Simulation.

Running Simulation with Distributed FMUs

You can use the FMUs generated from SCADE Suite models to run (black-box) simulation with FMI-compatible tools such as Ansys Twin Builder, Dassault Systèmes Dymola, LMS AMESim, etc.

- [“Distributed FMU Simulation Overview”](#)
- [“Distributed FMU Simulation Workflow”](#)
- [“Generating FMU Proxy Using FMU Proxy Builder”](#)
- [“Editing Configuration File Used by FMU Proxy Builder”](#)
- [“Setting Connection over Network via FMU Loader”](#)

Distributed FMU Simulation Overview

The following figure provides a general overview of the simulation of SCADE components in FMI-compatible tools.

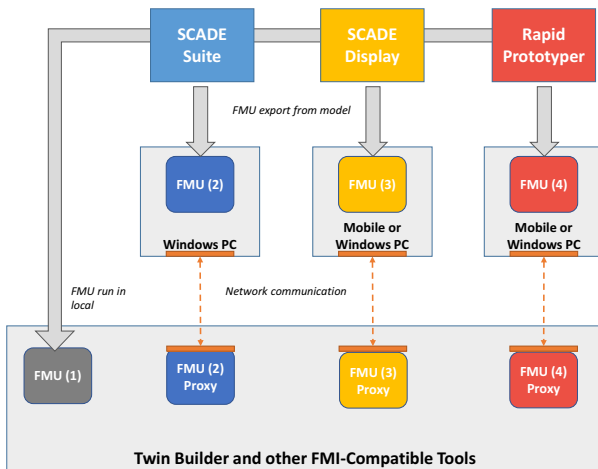


Figure 13.9: Simulation of SCADE components in Twin Builder and FMI-compatible tools

The FMUs exported from SCADE Suite models and used for simulation can be either run in local in the simulation environment, or distributed on different machines. In distributed simulation, an FMU proxy is generated out of each

exported FMU and imported in the simulation environment. Regarding network communication, the performances of the connectivity are directly linked to the network performances.

Distributed FMU Simulation Workflow

The workflow for running distributed FMU simulation is described below:

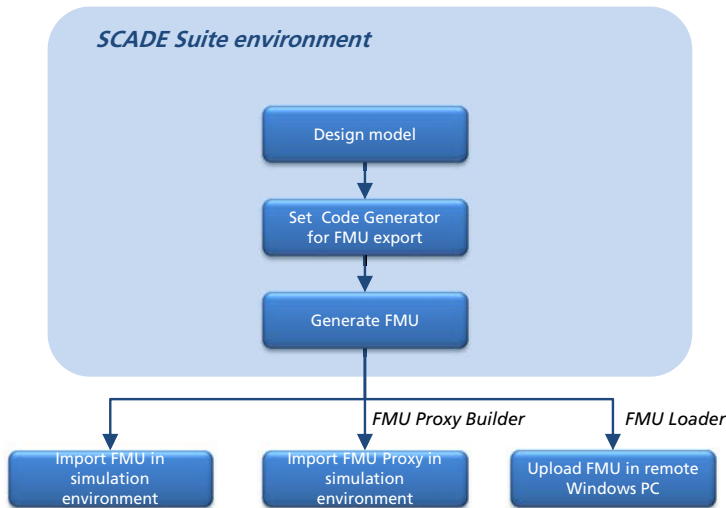


Figure 13.10: Distributed FMU simulation workflow

The workflow for running distributed FMU simulation consists of the following steps:

- 1 Setting FMU export and exporting one or several SCADE Suite models as FMUs: see [“Setting Code Generator for FMU Simulation”](#) on page 633 and [“Exporting SCADE Suite Model as FMU”](#) on page 636.
- 2 For each component that is run on a remote machine, converting the exported FMU into an FMU proxy using the FMU Proxy Builder tool: see [“Generating FMU Proxy Using FMU Proxy Builder”](#) on page 643.

- 3 In the FMI-compliant tool, importing the various FMU proxies (and other local FMUs) and connecting them according to the application's architecture: see [“Importing Generated Files in FMI-Compliant Tools”](#) on page 640.
- 4 For each component selected in step 2, uploading the original FMU (not the proxy) on the suitable remote machine and running it using the FMU Loader: see [“Setting Connection over Network via FMU Loader”](#) on page 646.
- 5 In the FMI-compliant tool, setting simulation parameters and running the simulation.

Generating FMU Proxy Using FMU Proxy Builder

The FMU Proxy Builder is a batch tool that takes as input an arbitrary FMU, and an optional configuration file containing parameters for the network communication to be established (see [“Editing Configuration File Used by FMU Proxy Builder”](#) on page 644). It produces an FMU proxy having the same FMI interface as the input FMU, but whose code is able to communicate with a remote FMU Loader.

You can run the FMU Proxy Builder executable on Windows in batch as follows:

```
fmu-proxy-builder [-config <CONFIG_FILENAME>] [-model_id <ID>] <FMU_FILENAME>
```

Possible command options and parameters:

- **-config** <CONFIG_FILENAME> to specify the path to a configuration file. For information on this file, see [“Editing Configuration File Used by FMU Proxy Builder”](#).
- **-model_id** <ID> to specify a valid FMI model identifier for the output FMU proxy.
- <FMU_FILENAME> (*mandatory*) to specify the path to an existing FMU file.

Editing Configuration File Used by FMU Proxy Builder

A configuration file for the FMU Proxy Builder tool contains a set of parameters of the form <PARAM_NAME>=<PARAM_VALUE>, each one written on a new single line.

Each parameter is optional and can appear at most once. The order of parameters in the file is not meaningful. Parameter names and values are case sensitive, but blank lines and additional white spaces are ignored. The table below specifies the set of parameters that may appear in the configuration file:

Table 13.1: Configuration file parameters

<PARAM_NAME>	<PARAM_VALUE>	Description
address	Either: • <ADDR>[:<PORT>] where <ADDR> is a valid IP address or host name, and <PORT> a valid port number • discover	The address parameter specifies how the FMU proxy attempts to connect to the remote component: • either by explicitly providing a network address <ADDR>:<PORT> (if no port set, 7777 is the default) • or by triggering a discovery protocol during the initialization phase If the parameter is not present, this is equivalent to address=discover.
timeout	A positive integer	When timeout=<N> is set, every network request performed by the FMU proxy (except for the discovery phase, if any) is canceled if it does not end normally within <N> milliseconds. By default, network requests last until they end normally (timeout=∞).

Table 13.1: Configuration file parameters (Continued)

<PARAM_NAME>	<PARAM_VALUE>	Description
discovery_time	A positive integer	When the discovery protocol is enabled, the FMU proxy tries (at initialization) to find a matching remote FMU on the network. The duration of this discovery phase is bounded by the value of the <code>discovery_time</code> parameter, expressed in milliseconds. By default, <code>discovery_time=1000</code>. Note: When several matching remote FMUs are launched simultaneously over the network, the FMU proxy connects to one of them: the particular instance to which it connects is unspecified. If several FMU Loaders are launched on the same machine, only one of them is able to perform the discovery phase.
fail_on_error	yes or no	If <code>fail_on_error=no</code> is set, the FMU proxy attempts to resume execution when a network error or timeout occurs. In such case, the consistency of results is not guaranteed. Otherwise, the FMU proxy fails (<i>i.e.</i>, it returns an FMI error) as soon as a network error or timeout occurs. This is also the default behavior when <code>fail_on_error</code> is not present.
log_file	A valid filename. It may be a path to an existing file or to a new file to be created.	This parameter specifies a file to trace network information and errors. This is intended for debugging purposes and to allow measuring network traffic. If it is not present, no log file is produced.

The FMU Proxy Builder creates an FMU proxy in the same directory as the input FMU. By default, when option `-mode l_id` is not set, the model identifier of the FMU proxy is “FmuProxy” and the output file is thus `FmuProxy.fmu`. A generic DLL (`FmuProxy.dll`) is used, no compilation step has to be done.

When `-model_id <ID>` is set, the model identifier of the FMU proxy is “<ID>” and the output file `<ID>.fmu`. In this case, the FMU Proxy Builder attempts to build a DLL (`<ID>.dll`), using GCC to compile and link the code. The following environment variables are expected: GCC32 to build a 32-bit DLL, and GCC64 to build a 64-bit DLL.

Setting Connection over Network via FMU Loader

The FMU Loader is the component which is able to answer requests of a matching FMU proxy (*i.e.*, having the same FMI interface) through network communication.

You can run the FMU Loader executable on Windows in batch as follows:

```
fmu-loader [-port <PORT>] <FMU_FILENAME>
```

Possible command options and parameters:

- **-port** <PORT> to specify a valid port number (if no port set, 7777 is used by default).
- <FMU_FILENAME> (*mandatory*) to specify the path to an existing FMU file.

To terminate FMU Loader execution

- Press **Ctr l+C** on the keyboard.

If an error occurs (invalid FMU file or network error), the FMU Loader stops with an exit code of 1.

The tool also prints all FMI logs (of the connected FMU proxy) on the standard output, and prints all network-related messages on the standard error.

Simulating Models in Twin Builder Environment

You can use SCADE Suite Code Integration Target for Twin Builder to generate a dynamic link library (DLL) for integrating SCADE Suite models into ANSYS® Twin Builder® simulation models. This allows co-simulating generated SCADE Suite models with entire complex systems in the Twin Builder environment.

You can also run simulation in Twin Builder with SCADE Suite-generated FMUs that are potentially distributed on different Windows PC machines. For more information, see [“Running Simulation with Distributed FMUs”](#) on page 641.

SCADE Suite solution for Twin Builder enables black-box and white-box co-simulation capabilities.

- [“Code Generation Workflow for Twin Builder Simulation”](#)
- [“Exporting SCADE Suite for Twin Builder Simulation”](#)
- [“Editing Operator Properties for Twin Builder”](#)
- [“Importing Generated Files in Twin Builder”](#)
- [“Setting Code Generator for Twin Builder Simulation”](#)
- [“Importing FMU for Parametric Analysis”](#)

Code Generation Workflow for Twin Builder Simulation

You can export SCADE Suite models as a DLL embedding the SCADE Suite KCG-generated code and based on the Software Development Kit (SDK) provided by Twin Builder for simulation within Twin Builder. The DLL publishes as many Twin Builder C-models as there are root operators declared for the code generation.

Note

SCADE Suite KCG allows specifying several root operators. For information, see [“Specifying Code Generation”](#) on page 836.

As illustrated below, you use SCADE Suite Code Integration Target for Twin Builder to generate, compile, and link SCADE Suite KCG code with all Twin Builder files necessary to run the co-simulation of SCADE Suite models within Twin Builder. You may also use FMU code generation to run Optimetrics Analysis from Twin Builder simulation environment.

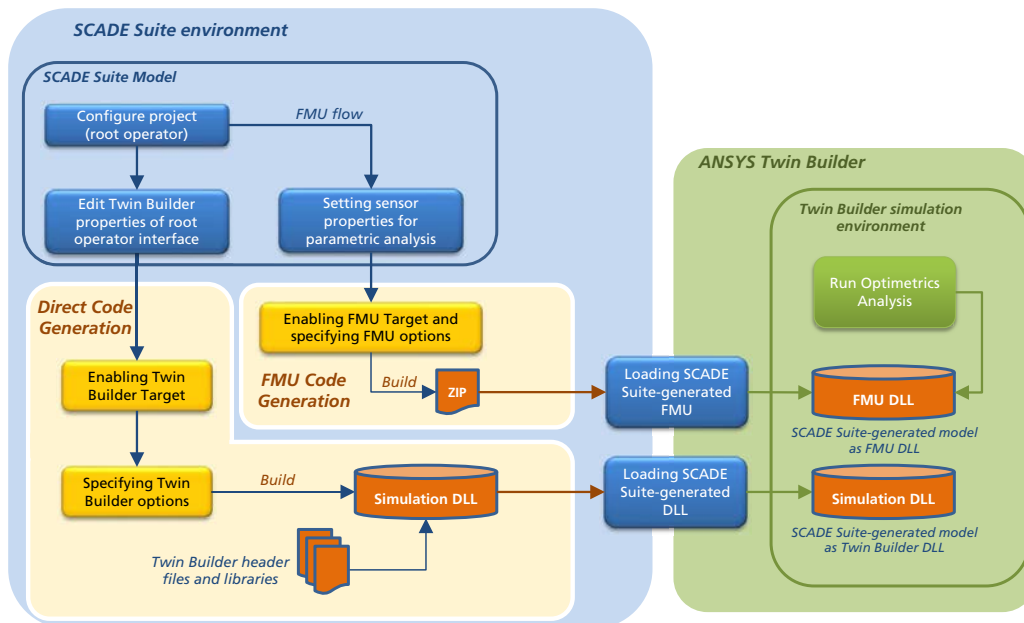


Figure 13.11: Code generation workflow for Twin Builder simulation

In *black-box co-simulation*, the C code from the SCADE Suite model is directly embedded in the generated DLL for Twin Builder.

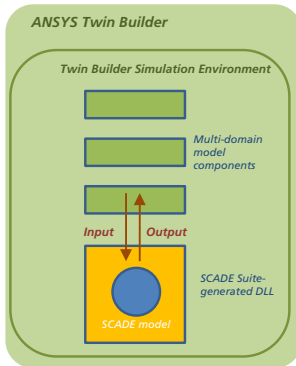


Figure 13.12: Twin Builder black-box co-simulation

In *white-box co-simulation*, the model is executed in SCADE Suite Simulation; and the Twin Builder forwards the inputs to SCADE Suite Simulation through a network connection. This mode enables model debugging with the full power of SCADE Suite simulation where you can observe model reactions graphically or set breakpoints and stop conditions.

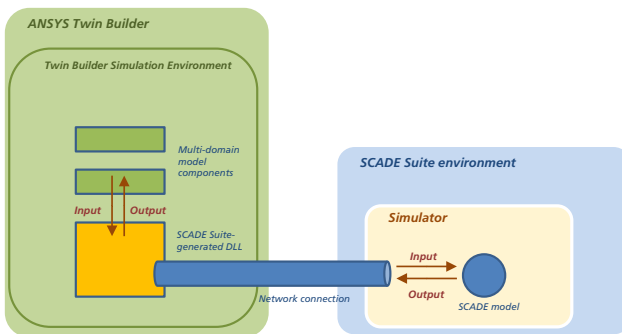


Figure 13.13: Twin Builder white-box co-simulation

SCADE Suite Code Integration Target for Twin Builder also allows recording the input scenarios during simulation. In both white-box and black-box co-simulation, the SCADE Suite cyclic function of the operator is called periodically at user-specified period.

Editing Operator Properties for Twin Builder

You can set Twin Builder properties for the inputs/outputs of a root operator, or for sensors. These properties are string values and may remain empty.

To set the Twin Builder properties of a SCADE Suite operator

- 1 Right-click an input/output or a sensor in **Scade View** and select *Properties*.
- 2 Select **Twin Builder** and set required properties on the right.

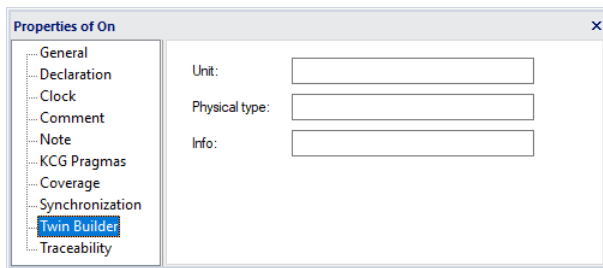


Figure 13.14: Twin Builder properties of input/output, or sensor

- 3 In **Unit** field, specify the unit of the quantity in or out of the model.
- 4 In **Physical type** field, specify the physical type of the quantity in or out of the model.
- 5 In **Info** field, type the test to display in Twin Builder Schematic and Model Agent.

Setting Code Generator for Twin Builder Simulation

You must enable code generation to generate SCADE Suite model for Twin Builder simulation and specify specific options from SCADE Suite Code Generator **Settings**.

Note

It is recommended to create a custom configuration with Twin Builder specific options. For details about managing configurations in SCADE Suite, see [“Managing Default or Custom Configurations”](#) on page 117 in *SCADE Suite User Manual*.

To set code generation for Twin Builder simulation

- 1 Click **Settings**  in **Code Generator** toolbar. Select a configuration from the drop-down list in the upper right corner.
- 2 Display **Code Integration**, set **Target**, and select **Twin Builder** in list.

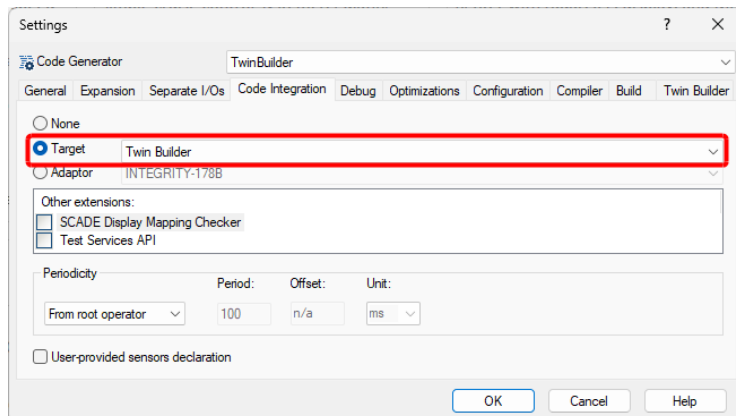


Figure 13.15: Enabling code generation for Twin Builder generation

- 3 Specify the periodicity at which Twin Builder calls the SCADE operator cyclic function: set [Periodicity](#) either at root operator level or override root operator periodicity.

- 4 Enable sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked). For details, see [User-provided sensors declaration](#).
- 5 Display **Twin Builder** to set target-specific options.

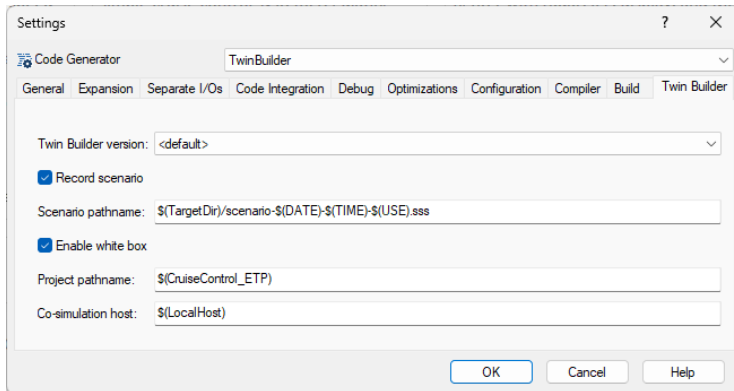


Figure 13.16: Specifying Twin Builder export options

- 6 Select the installed version of Twin Builder to use from **Twin Builder version** drop-down list. <default> indicates the most recent version of Twin Builder is used.
- 7 Check **Record scenario** for recording a scenario (.sss file) during simulation. Specify the location of the file in **Scenario pathname** with a relative pathname and/or a pathname containing environment variables, like “./scn1.sss”, “\$(MYVAR)/scn1.sss”. Pathname resolution varies as detailed below:
 - *On the machine where the DLL is generated:* \$(ProjectDir) and \$(TargetDir) variables are resolved respectively as absolute paths of project directory and target directory; \$(Configuration) is resolved as configuration name. Relative paths and environment variables are not resolved.

- *On the machine where the DLL is executed:* `$(USE)`, `$(DATE)`, and `$(TIME)` variables are resolved first, then environment variables. `$(USE)` is resolved as name of operator instance in Twin Builder. If the resulting path is relative, it is made absolute using the DLL path as base.
- 8 Check **Enable white box** to run the exported model in white-box co-simulation or uncheck this option to run black-box co-simulation. The generated DLL is portable on a machine where the project and/or SCADE installation are not in the same directory as on the local machine. Specify the related options:
- Specify the location of the SCADE Suite project to use in white-box co-simulation from **Project pathname**. Use a relative project pathname and/or a pathname containing environment variables, like “`../myProject.etp`”, “`$(MYVAR)/myProject.etp`”, “`$(myProject.etp)`”.

Note

- *For launching SCADE Suite*, the generated DLL uses the `%SCADE%` environment variable if existing or the most recent SCADE version found in Windows registry.
 - *For opening the project*, the generated DLL uses the specified Project pathname. Environment variables in the pathname are first resolved. If a variable does not exist, the DLL opens the project with the local path. If the resulting path is relative, it is made absolute using the DLL path as base.
 - White-box co-simulation is only possible for one instance of an operator in a given DLL. Multi-root operators are not allowed with this option.
-
- Specify the IP address of the host running SCADE Suite in **Co-simulation host**. Do not change the default value if running co-simulation with SCADE Suite and Twin Builder on the same local host. The `$(LocalHost)` macro can be used. Co-simulation with Twin Builder can be launched from another host able to connect to the specified IP. This option does not support multiple root operators.

- 9 Display **Compiler** to set compiler-specific options and **Build** to set specific build options. **CPU type** for compiler is automatically set to **win64**. For details, see [“Setting Compiler and Build Options”](#) on page 853.

Exporting SCADE Suite for Twin Builder Simulation

Once code generation is set, you can use it to export the C code of the SCADE Suite model contained into a DLL for co-simulation in Twin Builder.

To export SCADE Suite model for Twin Builder simulation

- 1 Select the configuration set for Twin Builder export from **Code Generator** toolbar.
- 2 Click **Build**  from **Code Generator** toolbar. SCADE Suite generates the DLL file in the target directory. The DLL is named Smp<ProjectName>.dll and contains the SCADE Suite model-generated code.

Importing Generated Files in Twin Builder

After importing the SCADE Suite-generated DLL in Twin Builder, use SCADE Suite operators as any other components and run [black-box](#) or [white-box](#) co-simulation. Based on [Twin Builder settings](#), Twin Builder starts black-box or white-box co-simulation.

- *In both modes*, import the SCADE Suite-generated DLL into the design and run the simulation capabilities of Twin Builder.

- *In white-box co-simulation*, starting simulation with Twin Builder launches SCADE Suite Simulation in any open session. If no SCADE Suite session is running, a new one is opened. Stopping simulation from Twin Builder closes SCADE Suite.

Note

Refer to Twin Builder documentation for instructions about simulation. For information about SCADE Suite Simulation, see [“Running Simulation in SCADE Suite”](#) on page 583.

Importing FMU for Parametric Analysis

The sensors you define in SCADE Suite projects can be set as variables for parametric analysis and imported in Twin Builder by means of SCADE Suite-generated FMU. Calibrated sensors allow to run Optimetrics analysis through Twin Builder simulation. Optimetrics analysis supports multiple simulation by varying parameters in Twin Builder.

To import FMU with calibration parameters in Twin Builder

- 1 Set sensors for parametric analysis as explained in [“Setting Sensors as Calibration Parameters”](#) on page 631.
- 2 Generate FMI-compliant code for Twin Builder simulation with calibrated parameters using FMU Target as explained in [“Setting Code Generator for FMU Simulation”](#) on page 633 and [“Exporting SCADE Suite Model as FMU”](#) on page 636.
- 3 Import FMU into Twin Builder and run simulation sessions for Optimetrics analysis.

Simulating Models in Simulink Environment

You can use SCADE Suite Code Integration Target for Simulink to generate an S-Function dynamic library and a Simulink S-Function Wrapping library model (SFW) necessary to co-simulate SCADE Suite models within Simulink/MATLAB environment.

SCADE Suite solution for Simulink enables black-box and white-box co-simulation capabilities. It is possible to co-simulate SCADE Suite models whether or not translated from Simulink models. In case of models translated from Simulink, a hybrid Simulink model is generated automatically.

- ["Code Generation Workflow for Simulink Co-Simulation"](#)
- ["Setting Simulink Code Wrapping"](#)
- ["Creating S-Function Dynamic Library/SFW Library Model"](#)
- ["Launching Co-Simulation with Simulink"](#)
- ["Executing MATLAB Files"](#)
- ["Advanced Tuning"](#)

Code Generation Workflow for Simulink Co-Simulation

The workflow for Simulink code generation and co-simulation between SCADE Suite and Simulink models consists of various tasks that you perform from SCADE Suite or Simulink environments.

SCADE Suite provides the ability to perform co-simulation of SCADE Suite models with Simulink environment. SCADE Suite Simulink Wrapper enables co-simulation as follows:

- *Black-box co-simulation*: Simulink Wrapper generates C code that encapsulates code generated by SCADE Suite KCG Code Generator to provide the expected interface to the Simulink S-Function.
- *White-box co-simulation*: Simulink Wrapper generates the S-Function code, which interacts with SCADE Suite Simulation via Simulation API functions.

Note

An S-Function Wrapping (SFW) model is required for black-box and white-box co-simulation capabilities.

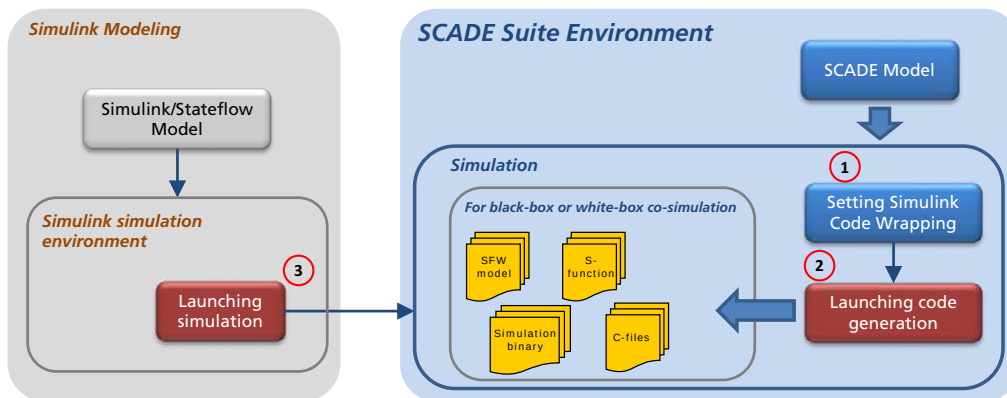


Figure 13.17: Simulink code generation workflow for co-simulation

To enable co-simulation, a Simulink S-Function Wrapping library model is necessary. This model contains an S-Function with proper wrapping for model input/outputs of the SCADE model as illustrated below.

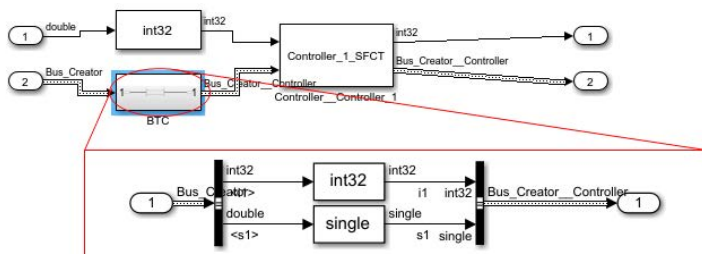


Figure 13.18: Content of Simulink S-function with Input/output wrapping

The Simulink S-Function can perform: data type conversions¹; folding/unfolding of structured data¹; connection of SCADE Suite probes to Scope/Display²; connection of SCADE Suite sensors to Constant. The following activities are necessary for using Simulink code wrapping and co-simulation:

- 1 Setting code wrapping: [“Setting Simulink Code Wrapping”](#) on page 659
- 2 Generating code and files required for co-simulation: [“Creating S-Function Dynamic Library/SFW Library Model”](#) on page 663
- 3 Launching simulation (black-box or white-box) from Simulink models: [“Launching Co-Simulation with Simulink”](#) on page 665
- 4 Tuning sample time, data types, sensor values, or Bus object names may be necessary: [“Advanced Tuning”](#) on page 667

-
1. Generated by default but not mandatory.
 2. The value of SCADE Suite probes connected to S-Function outputs is considered as always valid. This may be wrong because validity depends, amongst other criteria, on the states and control block hierarchy.

Setting Simulink Code Wrapping

Simulink code wrapping is an extension of SCADE Suite Code Generator and is required for black-box and white-box simulation modes. Code wrapping generates the S-Function dynamic library (a wrapper around SCADE Suite-generated code), the SFW library model (a capsule of the S-Function), and a hybrid model (a copy of the Simulink source model where the controller is replaced by a capsule of S-function). The SFW library model can be used in black-box simulation from Simulink or in white box co-simulation from Simulink driving SCADE Suite Simulation.

Note

Simulink code wrapping supports multiple instances of the S-function with some limitations: no white box co-simulation, no scenario, no `global_root_context` KCG option. All SCADE Suite KCG options are supported by Simulink code wrapping.

To set Simulink code wrapping options

- 1 Click **Settings**  in **Code Generator** toolbar.
- 2 Display **Code Integration**, set **Target**, and select **Simulink** in the list.

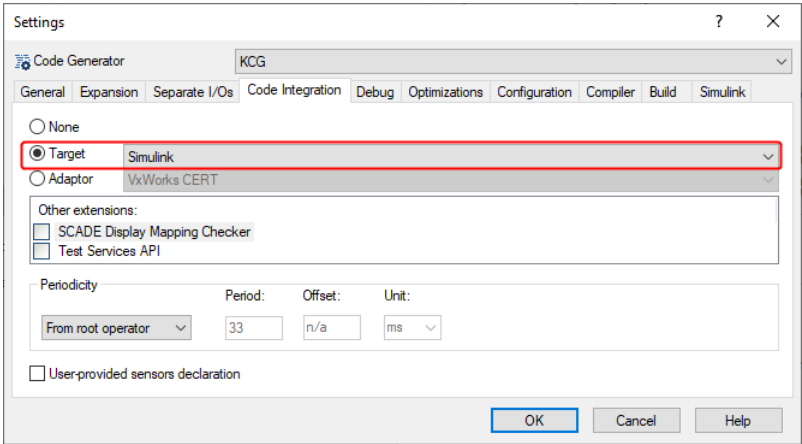


Figure 13.19: Selecting Simulink as code extension

- 3 Set **Periodicity** to modify the periodic execution of the S-function.
 - When selecting **Override**, the SampleTime of the S-Function will read from the specified **Period** and **Unit** converted into seconds.
 - When selecting **From root operator**, the SampleTime of the S-Function will read from the **Period** and **Unit** specified in the **Code integration** properties of the root operator and converted into seconds.
 - Do not select **No periodicity** since Simulink code wrapping cannot generate the S-function without periodicity. For details about periodicity, see [“Setting Periodicity of Code Execution”](#) on page 852.
- 4 Check **User-provided sensors declaration** to prevent adding the sensors as inputs of the S-function block when the C definition of sensors is provided in a user C source file. With this option, Simulink code wrapping discards the generation of the corresponding blocks in the S-Function wrapper (`<GenerationName>_SFW.slx` or `<GenerationName>_SFW.mdl`).
- 5 Display **Simulink** to set target specific options.

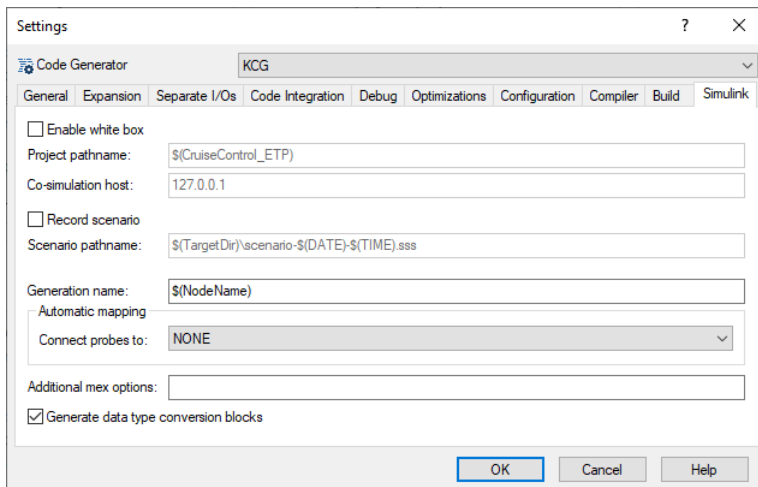


Figure 13.20: Setting Simulink code wrapping generation options

- 6 Check **Enable white box** to generate an S-Function supporting Simulink co-simulation with SCADE Suite code running as a white box. If unchecked, the generated code runs as black box during Simulink simulation. Specify the location of the SCADE Suite project to use for white-box co-simulation in **Project pathname**. Specify a relative project pathname and/or a pathname containing environment variables, like “../myProject.etp”, “\$(MYVAR)/myProject.etp”, “\$(myProject.etp)”.
-

Note

- *For launching SCADE Suite*, the generated S-function uses %SCADE% environment variable if any or the most recent SCADE version found in Windows registry.
 - *For opening the project*, the generated S-function uses the specified Project pathname. Environment variables in pathname are resolved first. If a variable does not exist, the S-function opens the project from the local path. If the resulting path is relative, it is made absolute using the S-Function path as base.
-

- 7 In **Co-simulation host**, set the IP address of the host running SCADE Suite. You must create the M-file, the S-Function dynamic library, and the simulation executable on this host (see [“Creating S-Function Dynamic Library/SFW Library Model”](#) on page 663). Simulation with Simulink can be launched from any host able to connect to the specified IP.
-

Warning

Do not change the default value of the co-simulation host if you run co-simulation with SCADE Suite and Simulink on the same local host.

- 8 Check **Record scenario** to enable the generated code of the S-function to produce a SCADE Suite scenario (.sss file) at each simulation session running from Simulink environment. Specify the location of the file in **Scenario pathname** with a relative pathname and/or a pathname containing environment variables, like “../scn1.sss”, “\$(MYVAR)/scn1.sss”. Pathname resolution varies as detailed below:

- *On the machine where generating S-Function:* `$(ProjectDir)` and `$(TargetDir)` variables are resolved respectively as absolute paths of project directory and target directory; `$(Configuration)` is resolved as configuration name. Relative paths and environment variables are not resolved.
 - *On the machine where executing S-Function:* `$(DATE)`, `$(TIME)`, and `$(INSTANCE)` variables are resolved before environment variables. `$(INSTANCE)` is replaced at runtime by the instance path of the S-function, allowing to instantiate the same S-Function several times and producing a scenario for each instance. If the resulting path is relative, it is made absolute using the S-Function DLL path as base.
 - By default, the scenario file is in the same location as the S-Function DLL. Scenario files are named according to the session date and time.
- 9 In **Generation name**, set the S-Function file and function name. Default is `$(NodeName)` where `NodeName` is the root operator set for code generation.

Warning

The Sample time displayed after translation is computed based on the Greatest Common Divisor of all clock data in the model. It ensures coherency between the SCADE Suite model and the Simulink S-Function generated for co-simulation. Change the value with caution. Read more about [“Defining Sample Time”](#) on page 669.

10 In **Automatic mapping** options:

- From **Connect probes to** drop-down, select whether or not to connect all probe instances of the SCADE Suite model through the S-Function: connect to *Display*, *Scope*, *Terminator*, *To File*, or *To Workspace* blocks.

11 Specify options for mex compiler (MATLAB) in **Additional mex options**.

- 12** Check **Generate data type conversion blocks** to enable the generation of data type conversion blocks in the S-function capsule. If conversion blocks cause performance issues, the option can be unchecked, but users must ensure that the S-Function input types are correct.
- 13** Display **Compiler** to set **CPU type** as **win32** or **win64** to generate 32-bits or 64-bits S-Function dynamic library, and set compiler-specific options. Display **Build** for specific build options. For details, see [“Setting Compiler and Build Options”](#) on page 853.

After setting code wrapping, it is necessary to create of S-Function dynamic library and the SFW library model as explained in [“Creating S-Function Dynamic Library/SFW Library Model”](#) on page 663.

Creating S-Function Dynamic Library/SFW Library Model

SCADE Suite generates the C code and an M-file that facilitates the creation of the S-Function dynamic library used by Simulink for co-simulation with SCADE Suite. Executing the M-file launches the S-Function dynamic library creation with MATLAB. Building this library is required for co-simulation:

- MATLAB current directory is set to the directory where the S-Function dynamic library is located
- Simulink code wrapping supports Microsoft Visual compiler (not Simulink lcc compiler)

To create S-Function dynamic library and SFW library model

- Click **Build**  in **Code Generator** toolbar to generate all code necessary with the SCADE Suite simulation executable. The following files are generated:
 - <GenerationName>_SFCT_mk.m (MATLAB script file) and C files
 - <GenerationName>_SFW.mk.m
 - <GenerationName>_SFW.mdl or <GenerationName>_SFW.slx
 - <GenerationName>_Bus.mat (if any Simulink Bus Objects)
 - <GenerationName>_SFCT.mexw64

All files are stored in the output directory set for code generation of the project.

At this stage, you can launch co-simulation session with Simulink:

- [“Launching Black-Box Co-Simulation with Simulink”](#)
- [“Launching White-Box Co-Simulation with Simulink”](#)

If the input SCADE model is produced from a Simulink import, a copy of the original Simulink model is also created. In this copy, the original Simulink controller is replaced by the newly created S-Function.

The <GenerationName>_HYB_mk.m script produces in the output directory the following files: <GenerationName>__HYB.slx or <GenerationName>__HYB.mdl. When the SCADE Suite model does not come from a Simulink import, the above script file is not generated and the following warning message pops up: “Root operator does not have SLW_Controller annotation, hybrid model will not be generated.” The root operator annotation SLW_Controller contains the path of the original Simulink controller if any.

Launching Co-Simulation with Simulink

Depending on how code wrapping for Simulink was set in Simulink target settings, you launch co-simulation in white-box or black-box as explained below.

- [“Launching Black-Box Co-Simulation with Simulink”](#)
- [“Launching White-Box Co-Simulation with Simulink”](#)

Launching Black-Box Co-Simulation with Simulink

Black-box co-simulation consists in Simulink simulation sessions where you simulate the SCADE Suite model imported as a Simulink S-Function.

- If the input SCADE model is produced from a Simulink import, `<GenerationName>_HYB.slx` or `<GenerationName>_HYB.mdl` must be open in Simulink, with the MATLAB current directory correctly set.
- If the input SCADE model is directly developed from a specification, `<GenerationName>_SFW.slx` or `<GenerationName>_SFW.mdl` must be used as component of a Simulink simulation model. This model should call the generated S-Function Wrapper library (`<GenerationName>_SFW.mdl`) and provide its inputs.

To launch black-box co-simulation

- 1 In Simulink, click **Start simulation** button.
- 2 Execute simulation operations using Simulink commands. Please consult Simulink documentation for more information and instructions about simulation operations.

Launching White-Box Co-Simulation with Simulink

In white-box co-simulation, running simulation from Simulink launches SCAD Suite slave simulator. If SCAD Suite is closed, a new session starts. If open, simulation starts in the active session. At launch, SCAD Suite Simulation is in pause allowing to set breakpoints or perform several simulation steps.

- If the input SCAD model is produced from a Simulink import, `<GenerationName>_HYB.slx` or `<GenerationName>_HYB.mdl` must be open in Simulink, with the MATLAB current directory correctly set.
- If the input SCAD model is directly developed from a specification, `<GenerationName>_SFW.slx` or `<GenerationName>_SFW.mdl` must be used as component of a Simulink simulation model. This model should call the generated S-Function Wrapper library (`<GenerationName>_SFW.mdl`) and provide its inputs.

To launch white-box co-simulation

- 1 In Simulink, click **Start simulation** button.
- 2 Execute simulation operations using SCAD Suite or Simulink commands.

Refer to Simulink documentation for details about simulation operations in Simulink. Consult Chapter 12 about [“Running Simulation in SCAD Suite”](#) for detailed instructions about SCAD Suite Simulation.

Executing MATLAB Files

It is possible to execute M-files from SCAD Suite projects and interact with MATLAB environment.

To execute MATLAB M-files from SCAD Suite

- Right-click any M-file in **File View** and select *Execute MATLAB M-File* to initialize the MATLAB workspace.

Advanced Tuning

Users may want to adapt SCADE models to support S-Function code generation by defining one’s own data types, sensor values, or sample time. In some cases, they may also need to rename bus objects.

- [“Defining Data Types Wrapping”](#)
- [“Defining Sample Time”](#)
- [“Defining Sensor Values in MATLAB”](#)
- [“Renaming Simulink Bus Objects by Annotations”](#)
- [“Defining Sensor Values in User-Provided C Code”](#)

DEFINING DATA TYPES WRAPPING

The SCADE Suite operator to wrap can be specified with input/output of any Scade type using basic Scade types except char (raising an error). The S-Function built from the code generated for that operator requires that the type of the signals map with Scade types.

Simulink Type Conversion operators can be used to provide the S-Function with signals of correct types.

Table 13.2: Mapping for basic types

Scade	Simulink	Scade	Simulink
[u]int8 16 32	[u]int8 16 32	float64	double
float32	single	bool	boolean

Inputs/outputs of SCADE Suite operators can be nested structures, tuples, and/or arrays of different types:

[int32, int32, int32]

DEFINING SENSOR VALUES IN MATLAB

Sensor values can be defined in MATLAB. Disable the [User-provided sensor declaration](#) option when setting Simulink code wrapping. When defining sensor values in MATLAB:

- The S-function has one additional input, in addition to the root operator inputs, for each sensor used in the model.
- The inputs are fed with Constant blocks in the S-Function capsule: the Constant block value is a reference to a MATLAB variable with the same name as the sensor.
- The sensor value is set using the MATLAB variables as illustrated below.

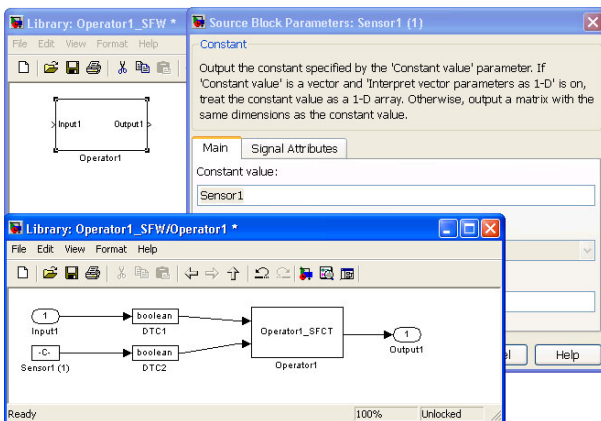


Figure 13.21: Setting sensor value with MATLAB variables

DEFINING SENSOR VALUES IN USER-PROVIDED C CODE

Sensor values can be defined as user C code in SCAD Suite. Enable the [User-provided sensor declaration](#) option when setting Simulink code wrapping.

When defining sensor values in external C code:

- The S-function has the same inputs as the root operator.
- The user-provided C code gives the value for sensors. You need to add the file in the project and set it as Source File for C in Simulation properties as detailed in [“Editing Project Files Properties”](#) on page 431).

DEFINING SAMPLE TIME

The sample time is a clock set in Simulink models. As documented for the S-Function by The MathWorks, Inc., it implies the following:

- By default, the sample time of the S-Function is set to 1.
- Real numbers are greater or equal to 0.
- -1 indicates an inherited sample time.
- -2 indicates variable sample time for the S-Function block.

A sample time of 0 means the S-Function is a continuous block. If the sample time is less than 0, a warning appears and a sample time of 1 is used for the S-Function. When translating Simulink models into SCADE Suite, a basic clock value is computed and its value is stored as a pragma of the root operator.

Note

You can define a sample time value for the S-Function when setting Simulink code wrapping. When From root operator periodicity is set in Code Integration settings, the sample time of the S-Function is read from the pragma. Otherwise, it is set by the Override periodicity option and the value fields. For details, see [“Setting Simulink Code Wrapping”](#) on page 659.

The basic clock value can be set in the translation wizard; ensuring that this clock value, which can be used in SCADE Suite model (e.g., for Discrete Time integrators), is consistent with the one set for the Simulink S-Function.

Note

An S-function samples all its inputs with the same clock. If in the original Simulink model, the inputs of the subsystem replaced by the S-function do not have the same clock, the behavior of the S-function might be different from the original Simulink model because of the sampling performed by the S-function.

RENAMING SIMULINK BUS OBJECTS BY ANNOTATIONS

Whenever users need to resolve name conflicts or facilitate identification of bus objects with easier names, it is possible to rename Bus objects in SCAD Suite models by using a dedicated annotation¹.

To rename Bus objects

- 1 In **Scade View**, select any Bus object imported from Simulink model.
- 2 Right-click to select *Properties* and display **Note** properties.
- 3 Click **New +** to open the **New Note** window.
- 4 In **Note types** select `SLW_BusObject` and enter new name in text field. Click **OK** to validate and display selection in properties.

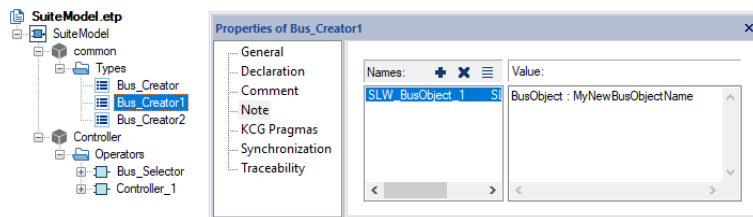


Figure 13.22: Using annotations to rename Simulink Bus objects

1. It is in the `SimulinkWrapper.aty` file located at `SCADE\lib` in installation.

Simulating Models in VeriStand Environment

SCADE Suite Gateway for VeriStand extends SCADE Suite environment with the ability to perform co-simulation of SCADE Suite models in NI VeriStand environment. It allows to validate interactively SCADE Suite models using SCADE Suite Simulation and all VeriStand and LabVIEW instrumentation capabilities to reproduce the application environment including physical data acquisition capabilities. Co-simulating SCADE Suite models with VeriStand enables to verify complex algorithms (like signal processing).

Note

For guidelines, consult Chapter 2 about [“Configuring Co-Simulation with VeriStand”](#) in *Gateway Guidelines for VeriStand*.

14 /Running Advanced Simulation

This chapter explains how you configure and set the projects for advanced simulation sessions. Simulating models with external code or imported types requires specific attention. You may also simulate SCADE Suite models in remote control or in double simulation mode.

- [“Simulating Models with External C Code”](#)
- [“Simulating Models with Imported C Types”](#)
- [“Implementing C Macro User Definitions”](#)
- [“Simulating Models with User C-Macros”](#)
- [“Simulating Models with External Ada Code”](#)
- [“Using Simulation API for Co-Simulation”](#)
- [“Running Double Simulation Sessions”](#)

Simulating Models with External C Code

As SCADE Suite Simulation is based on generated code, C files are required when you simulate a model using imported operators. SCADE Suite Simulation needs to know that corresponding C files are required to build a simulation executable. Such external files must be listed in **File View** and declared as source file. For detailed instructions about imported operators, see [“Declaring and Editing Imported Operators”](#) on page 194.

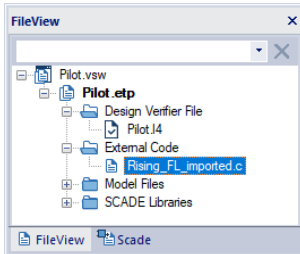


Figure 14.1: C-files of imported operator stored in SCADE Suite File View

To declare external C files for building simulation executable

- 1 In **File View**, right-click a C file needed for simulation and select *Properties*.
- 2 Select **Simulation** on the left and set required properties on the right.
- 3 Set the type of file to **Source File for C** from the **File type** list.

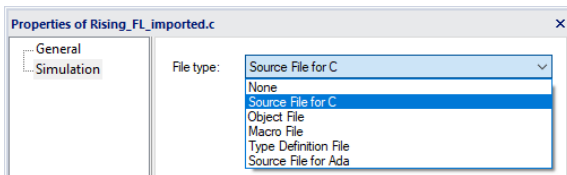


Figure 14.2: Declaring external C file for building simulation binary

Declaring a C source file directly from the **Properties** of the imported operator automatically inserts the file in **File View** under **External Code** folder and set as **Source File for C**.

The building process of the simulation executable compiles C code only when files are created or modified. External code must be compiled only once for each project. In the context of Simulation, it is possible to add a call to a debug trace function in user-provided code. The function is:

```
void SsmOutputMessage(int level, const char* msg)
```

where **level** is 1 for none/info, 2 for warning, 3 for error. The debug trace is displayed in the **Simulation** tab of the **Output Window**.

Simulating Models with Imported C Types

Imported types can be added or set without interface changes in SCADE Suite Simulation. Imported types are implemented in models as detailed in Chapter 20 about [“Implementing Models with Imported Code”](#). For simulation, you need to provide C functions allowing values to be compared and converted to/from strings.

To set external files for simulation with imported types

- 1 Define two files: an imported type definition (.h) file and a conversion/comparison function (.c) file. For details and examples about conversion/comparison functions, consult Chapter 5 about [“Dedicated Simulation Functions”](#) in *SCADE Suite Technical Manual*.
- 2 Insert both files in **File View**.

Tip

Create a new folder called `External Type` to insert new files.

- 3 In **File View**, right-click each file and select *Properties*
- 4 Select **Simulation** on the left and set required properties on the right.
- 5 Set each file from the **File type** list as follows:
 - For imported type definition file (. h), select **Type Definition File**.

Note

When declaring the definition file directly from the Properties of the imported type, the H-file is automatically inserted in the project File View under `External Type Definitions` folder and set as “Type Definition File”. For details, see [“Setting Imported Types”](#) on page 214.

- For conversion/comparison function file (. c), select **Source File for C**.

Implementing C Macro User Definitions

All default macros generated by KCG can be defined to customize generated code for specific user needs and cases. Such customizations may have an impact on SCADE tools like SCADE Suite Simulation, SCADE Test Model Coverage, and SCADE Test Environment for Host.

- [“Implementing Supported Type Definition”](#)
- [“Implementing Unsupported Type Definition”](#)
- [“Implementing Imported Type Definition”](#)
- [“Implementing Unsupported Macro Definition”](#)
- [“Declaring Type/Macro Definition in Projects”](#)

Depending on customization needs, implementing macro user definitions involves defining given macros and simulation functions (VTables) in specific files to be declared in projects.

Supported macro definitions can be used by SCADE Suite Simulation, SCADE Test Model Coverage, and SCADE Test Environment for Host without user intervention. *Unsupported macro definitions* require users to provide specific VTables to adapt the behavior of the tools. The following table summarizes the different supported/unsupported customization cases.

Table 14.1: Macro user definition cases, simulation functions, expected files

Macros for:	Types	Boolean Constants	Other ¹	C Operators	Imported Types
Supported definitions	Default and custom (see Table 14.2)	kcg_true = 1 kcg_false = 0	n/a	All default when Wrap C Operators option used	n/a
Simulation functions²	Use of built-in functions		n/a	Use of built-in functions	n/a

Table 14.1: Macro user definition cases, simulation functions, expected files

Macros for:	Types	Boolean Constants	Other ¹	C Operators	Imported Types
Expected definition files	Default in kgc_types.h				n/a
	Custom definition in user file (.h) declared as macro file	n/a	n/a	n/a	
Unsupported definitions	Custom definitions other than supported definitions		Custom definitions		External user definitions
Simulation functions	VTable definition likely required	VTable definition not necessary	n/a	VTable definition required	
Expected definition files	Definitions in user file (.h) declared as macro file				Definitions in user file (.h) declared as type definition file
	VTable definitions in user file (.c) declared as C source file		n/a	VTable definitions in user file (.c) declared as C source file	

1. Includes all kgc_assign macros as well as kgc_assume and kgc_guarantee macros.
2. For details, see [“Conversion and Comparison Functions for Types”](#) on page 105.

Implementing Supported Type Definition

Some definitions for predefined types (bool, char, (u)int8|16|32|64, float32|64) are natively supported in SCADE Suite. Based on the supported definitions of predefined types, SCADE Test Model Coverage, Simulation, and

SCADE Test Environment for Host can work without any specific customization. Users can modify the corresponding macros within a given scope and take care of the declaration of defined macros in projects.

Note

For certification activities of Model Coverage and Test Execution, the supported type definitions do not require any additional activities. For details about the respective certification data, contact SCADE Products Technical Support.

Table 14.2: Supported macro definitions for predefined types

Type Macros	Supported definitions
kcg_bool	unsigned char (default), unsigned long
kcg_char	char (default), unsigned char
kcg_int8	signed char
kcg_uint8	unsigned char
kcg_int16	signed short
kcg_uint16	unsigned short
kcg_int32	signed long
kcg_uint32	unsigned long
kcg_int64	signed long long
kcg_uint64	unsigned long long
kcg_float32	float
kcg_float64	double
kcg_size	ptrdiff_t (internal)

The following type macros can be defined with the following syntax in KCG C:

Default Type Definitions

```
#ifndef kcg_bool
#define kcg_bool kcg_bool
typedef unsigned char kcg_bool;
#endif /* kcg_bool */

#ifndef kcg_char
#define kcg_char kcg_char
typedef char kcg_char;
#endif /* kcg_char */
```

Supported Custom Definitions

```
#define kcg_bool kcg_bool
typedef unsigned long kcg_bool;

#define kcg_char kcg_char
typedef unsigned char kcg_char;
```

The following type macros can be defined with the following syntax in KCG:

Default Type Definitions

```
#ifndef kcg_int8
#define kcg_int8 kcg_int8
typedef signed char kcg_int8;
#endif /* kcg_int8 */

#ifndef kcg_int16
#define kcg_int16 kcg_int16
typedef signed short kcg_int16;
#endif /* kcg_int16 */

#ifndef kcg_int32
#define kcg_int32 kcg_int32
typedef signed long kcg_int32;
#endif /* kcg_int32 */

#ifndef kcg_int64
#define kcg_int64 kcg_int64
typedef signed long long kcg_int64;
#endif /* kcg_int64 */

#ifndef kcg_float32
#define kcg_float32 kcg_float32
typedef float kcg_float32;
#endif /* kcg_float32 */

#ifndef kcg_size
#define kcg_size kcg_size
typedef ptrdiff_t kcg_size;
#endif /* kcg_size */
```

```
#ifndef kcg_uint8
#define kcg_uint8 kcg_uint8
typedef unsigned char kcg_uint8;
#endif /* kcg_uint8 */

#ifndef kcg_uint16
#define kcg_uint16 kcg_uint16
typedef unsigned short kcg_uint16;
#endif /* kcg_uint16 */

#ifndef kcg_uint32
#define kcg_uint32 kcg_uint32
typedef unsigned long kcg_uint32;
#endif /* kcg_uint32 */

#ifndef kcg_uint64
#define kcg_uint64 kcg_uint64
typedef unsigned long long kcg_uint64;
#endif /* kcg_uint64 */

#ifndef kcg_float64
#define kcg_float64 kcg_float64
typedef double kcg_float64;
#endif /* kcg_float64 */
```

After defining types in H-file, users must ensure that the new definitions are included in the project as macro file and taken into account in code generation configuration. For guidelines, see [“Declaring Type/Macro Definition in Projects”](#) on page 681.

Implementing Unsupported Type Definition

Users can provide their own custom implementation of Scade predefined types. Yet, any definition outside the scope of [Table 14.2](#) may require additional activities.

In addition to defining the macros of the predefined types themselves, users may have to define the macros used for the arithmetics of user-defined types, as well as the conversion/comparison functions described in VTables and required for adapting simulation behavior.

After customizing types as needed in an H-file, one must ensure that the new definitions are included in the project as macro file and taken into account for code generation configuration. For guidelines, see [“Declaring Type/Macro Definition in Projects”](#) on page 681.

Implementing Unsupported Macro Definition

Users can provide their own custom implementation of all other default macros. Additional activities depend on the macros users customize.

CASE OF MACROS FOR BOOLEAN CONSTANTS

Users define the macros for Boolean constants (`kcg_true` and `kcg_false`) as needed. Changing the definitions of these macros does not impact the simulation behavior, but it is possible to combine them with user definition of Boolean arithmetics in macros for C operators. In this case, the simulation behavior is indirectly impacted .

If needed, one can define independently from the macro definition the conversion/comparison functions described in VTables and required for adapting simulation behavior.

CASE OF MACROS FOR C OPERATORS

In addition to defining the macros of the C operators themselves, one must define the comparison functions described in VTables and required for adapting simulation behavior.

CASE OF OTHER MACROS

Users define the `kcg_assign`, `kcg_assume`, or `kcg_guarantee` macros as needed. The conversion/comparison functions are not related to such macro definitions. Changing the definition of these macros affects only the behavior of the generated code and not simulation behavior.

In all above cases, after defining macros as needed in an H-file, you must ensure that the new definitions are included in the project as macro file and taken into account for code generation configuration. For guidelines, see [“Declaring Type/Macro Definition in Projects”](#) on page 681.

Implementing Imported Type Definition

Users can provide their own custom implementation with external type definitions. For guidelines, see [“Setting Imported Types”](#) on page 214.

For simulation purposes, users additionally need to declare the type definitions in external files with the definitions of the required conversion/comparison functions in the project. For user guidelines, see [“Simulating Models with Imported C Types”](#) on page 675.

Declaring Type/Macro Definition in Projects

Whenever customizing projects with type and macro definitions, users need to configure code generation for the new definitions. This does not apply to imported type definitions.

To declare type/macro definition for code generation

- 1 Define the macros in an H-file. By default, this file is named `user_macro.h` and included in `kcg_types.h` (`#include "user_macros.h"`).

Note

When defining types outside the scope of natively supported definitions (see [Table 14.2](#)), you must also define all the macros associated with the types.

- 2 Click **Settings**  in **Code Generator** toolbar and display **Configuration**.
- 3 In **User config**, browse for files and specify the H-file that contains the new type and/or macro definitions.

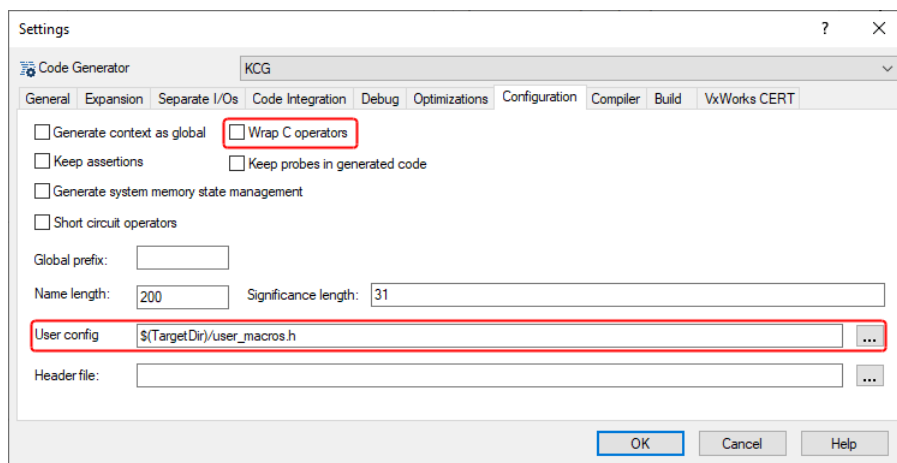


Figure 14.3: Setting code generation with new type/macro definitions

Note

In batch mode, use the `-user_config` and `-wrap_C_ops` options. For details, see [“Description of Configuration Options”](#) on page 309 in *SCADE Suite Technical Manual*.

- 4 If needed, check **Wrap C operators** to generate the defined macros instead of the native C operators. All arithmetic, binary, and relational operators are wrapped into C protected macro definitions. This option is not required when customizing types within the scope of supported definitions (see [Table 14.2](#)).

For simulation purposes, you additionally need to declare the new type/macro definitions with the definitions of the required conversion/comparison functions in the project. For guidelines, see [“Simulating Models with User C-Macros”](#) on page 684.

Note

Conversion/comparison functions are mandatory for processing redefined types and associated macros in simulation when the definition of the types is not natively supported in SCADE Suite (see [Table 14.2](#)). Conversion/comparison functions are also mandatory when changing any of the default macros without modifying type definitions.

Simulating Models with User C-Macros

The default macro definitions provided in SCADE Suite can be customized by users. User macros are implemented in models as detailed in [“Implementing C Macro User Definitions”](#) on page 676. For simulation, you need to provide the macro definitions and the corresponding C functions allowing values to be compared and converted to/from strings.

Note

For type definitions natively supported, you do not need to provide conversion/comparison functions as explained below.

To set external files for simulation with user macros

- 1 Define two files: a user macro definition (.h) file and a conversion/comparison function (.c) file.

Note

For details and examples about conversion/comparison functions, consult Chapter 5 about [“Dedicated Simulation Functions”](#) in *SCADE Suite Technical Manual*.

- 6 Insert both files in **File View**.
- 7 In **File View**, right-click each file and select *Properties*.
- 8 Select **Simulation** on the left and set required properties on the right.
- 9 Set each file from the **File type** list as follows:
 - For conversion/comparison function file (.c), select **Source File for C**.
 - For user macro definition file (.h), select **Macro File**.

Simulating Models with External Ada Code

SCADE Suite Simulation is based on generated code and requires Ada files when you simulate a model using imported operators or types. SCADE Suite Simulation needs to know which corresponding Ada files are required to build a simulation executable. Such external files must be listed in **File View** and declared as source file. For detailed instructions about imported operators, see [“Declaring and Editing Imported Operators”](#) on page 194.

Note

For details and examples about conversion/comparison functions, consult Chapter 5 about [“Dedicated Simulation Functions”](#) in *SCADE Suite Technical Manual*.

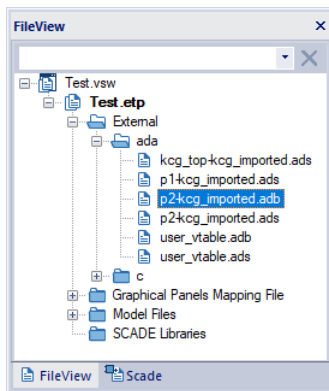


Figure 14.4: Ada files of imported operator stored in SCADE Suite File View

To declare external Ada files for building simulation executable

- 1 In **File View**, select an Ada file needed for the simulation executable.
- 2 Right-click on the file and select *Properties*.
- 3 Select **Simulation** on the left of Properties window and set the type of file to **Source File for Ada** from the **File type** drop-down list on the right.

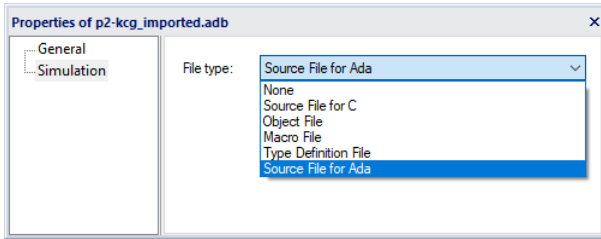


Figure 14.5: Declaring external Ada file for building simulation binary

Tip

When declaring an Ada source file directly from the Properties of the imported operator, the file is automatically inserted in the project File View under External Code folder and set as “Source File for Ada”.

The building process of the simulation executable compiles Ada code only when files are created or modified.

By default, an implementation of `Kcg_config` package is provided and used by simulation. This package is required by SCADE Suite KCG code to define all predefined types. The default definitions are contained in `kcg_config.ads` located under `SCADE\include\Ada` and in `kcg_config.adb` located under `SCADE\lib\Ada`.

It is possible to override the default definitions by providing any user implementation of types in files with the same name and inserting them in the File View of the project. The files must be declared as Ada external files as described above.

Using Simulation API for Co-Simulation

Co-simulation capabilities allow any external application written in C to take remote control over SCADE Suite Simulation through the public SCADE Suite Simulation API functions. It is possible to simulate in parallel the code of an external C application and any SCADE Suite model that completes the external application: you can run simultaneously a SCADE Suite Simulation session and a VC++ simulation session.

The simulated SCADE Suite model can contain C or Ada imported code with no impact on co-simulation. For details about co-simulation capabilities supported by SCADE Suite Simulation from external applications, consult Chapter 13 about [“Running Simulation from External Tools”](#).

When SCADE Suite Simulation works in co-simulation mode with an external application, the following Simulation features are not active anymore:

- **Back, Step, Fast Back, Fast Step, Go, Reset** buttons are disabled.
- Setting input values in **Instances Model Browser** or **Watch** window, scenarios, bookmarks and external GUI extensions are disabled.

You can use **Break**  to pause simulation from SCADE Suite, outside an external application. Use **Go**  to resume simulation.

Co-simulation mode is indicated in the **Simulation window** by tagging the ongoing Action as “External”.

- [“Setting Co-Simulation Compilation Resources”](#)
- [“Instrumenting Application for Co-Simulation”](#)

Setting Co-Simulation Compilation Resources

Before running SCADE Suite Simulation in co-simulation with an external application, you must ensure the adequate resources are accessible from this application. For successful compilation, the declaration of the API must be accessible from the external application.

To set the application code with the API gateway

- 1 Include `SsmSlaveLib.h` located in <SCADE installation>/include in the C files that need to access SCADE Suite Slave Simulator API.
- 2 Add to the project `SsmSlaveLib.c` located in <SCADE installation>/lib.

Instrumenting Application for Co-Simulation

Run SCADE Suite Simulation and the external C simulation by instrumenting the external application with calls to the SCADE Simulation API functions:

Table 14.3: List of SCADE Suite Simulation API main functions

Function	Basic Description
<code>SsmOpenScadeSimulator</code>	Opens SCADE Suite Simulation
<code>SsmCloseScadeSimulator</code>	Closes SCADE Suite Simulation
<code>SsmGuiActivate</code>	Brings SCADE Suite Simulation interface into foreground
<code>SsmGuiRefresh</code>	Refreshes simulation values in SCADE Suite Simulation interface
<code>SsmSetInput</code>	Sets a particular input of the simulated model
<code>SsmSetInputVector</code>	Sets the vector of inputs of the simulated model
<code>SsmStep</code>	Performs simulation steps
<code>SsmGetOutput</code>	Reads a particular output of the simulated node
<code>SsmGetOutputVector</code>	Reads the vector of outputs of the simulated node
<code>SsmGetLastError</code>	Retrieves the error message from the last caught error

You declare the API functions and modify the main function as described in:

- [“Starting SCADE Suite Simulation”](#)
- [“Retrieving Outputs”](#)
- [“Setting SCADE Suite Simulation Inputs”](#)
- [“Closing SCADE Suite Simulation”](#)
- [“Running Simulation Steps”](#)

Note

About functions, error codes, and an example of a main function enriched by API functions, see [“Public API Functions for Co-Simulation Mode”](#) on page 136 in *SCADE Suite Technical Manual*. About simulation values and paths declaration, see [“Simulation Value Format”](#) on page 154, and [“Simulation Path Format”](#) on page 157 in *SCADE Suite Technical Manual*.

Starting SCADE Suite Simulation

You use the **SsmOpenScadeSimulator** function declared as follows:

```
extern int SsmOpenScadeSimulator (const char* pszProjectPathName, const char*
pszConfName, const char* pszRootNode, int nSocketPort);
```

The function arguments are:

const char* PszProjectPathName	path string to project workspace (e.g., “C:/tmp/myproject.vsw”)
const char* pszConfName	name string of the project configuration with simulation parameters (e.g., default configuration “Simulation”)
const char* pszRootNode	name string of the root operator to simulation (e.g., “Package::MainNode”)
int nSocketPort	socket port number (e.g. 64064)

Note

Communication is socket-based. The default port, set to 64064, on which SCADE Suite is listening the gateway can be configured from project Properties. For guidelines, see [“Setting Project for Simulation Sessions”](#) on page 585.

Example:

```
int returnValue;
const char* pszReturnValue = NULL;
/* Open the SCADE Suite Simulator of node from specified project */
returnValue = SsmOpenScadeSimulator("C:/tmp/myproject.vsw", "Simulation",
"Package::MainNode", 64064);
if (returnValue) {
    SsmGetLastError(&pszReturnValue);
    printf("%s\n", pszReturnValue);
    return -1;
}
/* Make SCADE active window */
SsmGuiActivate();
```

See also [“SsmOpenScadeSimulator API function”](#) on page 137 in *SCADE Suite Technical Manual*.

Setting SCADE Suite Simulation Inputs

You can use the following functions to set inputs for simulation:

To set a value for a specific input

- Use the **SsmSetInput** function declared as follows:

```
extern int SsmSetInput(char* pszPath, char* pszValue);
```

The function arguments are:

char* pszPath	name of the input. The syntax is “<root_node_name>/<input_name>”
char* pszValue	string value to assign to the input

Example:

```
/* Set integer Boolean BoolInput */
SsmSetInput("Package::MainNode/BoolInput", "true");
/* Set integer input IntInput */
SsmSetInput("Package::MainNode/IntInput", "100");
/* Set structured inputs BoolIntStruct with two fields, first Boolean, second integer */
SsmSetInput("Package::MainNode/BoolIntStruct", "(true, 5)");
```

See also [“SsmSetInput API function”](#) on page 138 in *SCADE Suite Technical Manual*.

To set the values for the vector of all inputs

- Use the **SsmSetInputVector** function declared as follows:

```
extern int SsmSetInputVector(const char* pszInputVector);
```

The function arguments are:

const char* pszInputVector	string vector of values in their declaration order to assign to the inputs

Example:

```
/* Set the input vector BoolInput, IntInput, BoolIntStruct */  
SsmSetInputVector("true, 100, (true, 5)");
```

See also [“SsmSetInputVector API function”](#) on page 139 in *SCADE Suite Technical Manual*.

Running Simulation Steps

For running simulation steps on previously set inputs, use the **SsmStep** function declared as follows:

```
extern int SsmStep(int nCycle, int bRefresh);
```

The function arguments are:

int nCycle	number of execution cycles to perform
int bRefresh	flag that, if true, refreshes SCADE Suite Simulation interface

Example:

```
/* Play one step and refresh values */  
nResult = SsmStep(1, true);  
if (nResult) {  
    SsmGetLastError(&pszReturnValue);  
    printf("%s\n", pszReturnValue);  
    return -1;  
}
```

See also [“SsmStep API function”](#) on page 139 in *SCADE Suite Technical Manual*.

Retrieving Outputs

You can use the following functions for retrieving outputs from the last execution step:

To retrieve the value of a specific output

- Use the **SsmGetOutput** function declared as follows:

```
extern int SsmGetOutput(char* pszPath, const char** ppszValue);
```

Note

The buffer used by SsmGetOutput is static.

The function arguments are:

char* pszPath	name of the output. The syntax is "<root_node_name>/<output_name>"
char** ppszValue	pointer to the string value to store the string value of the output

Example:

```
char stringValue[100];

/* Retrieve integer Boolean output BoolOutput */
smGetOutput("Package::MainNode/BoolOutput", & stringValue);

/* Retrieve integer output IntOutput */
SsmGetOutput ("Package::MainNode/IntlOutput", & stringValue);

/* Retrieve structured outputs BoolIntStruct with two fields, first Boolean, second integer */
SsmGetOutput ("Package::MainNode/BoolIntStruct", & stringValue);
```

See also ["SsmGetOutput API function"](#) on page 139 in *SCADE Suite Technical Manual*.

To retrieve the values for the vector of all outputs

- Use the **SsmGetOutputVector** function declared as follows:

```
extern int SsmGetOutputVector(const char** ppszValue);
```

The function arguments are:

<code>char** ppszValue</code>	pointer to the string value to store the string values of the outputs in their declaration order

Example:

```
char* stringValue[100];

/* Retrieve the output vector BoolOutput, IntOutput, BoolIntStruct */
SsmGetOutputVector(&stringValue);
```

See also [“SsmGetOutputVector API function”](#) on page 140 in *SCADE Suite Technical Manual*.

Closing SCADE Suite Simulation

On simulation completion, the function **SsmCloseScadeSimulator** closes SCADE Suite Simulation. You use the function declared as follows:

```
extern int SsmCloseScadeSimulator();
```

See also [“SsmCloseScadeSimulator API function”](#) on page 137 in *SCADE Suite Technical Manual*.

Running Double Simulation Sessions

You can run a simulation session on two similar models at the same time. SCADE Suite is able to run simultaneously a main simulation session on the opened project and a background session whose model can be set from simulation settings. User scenario inputs are set from the main session and propagated to the background session. Both simulated models must share the same interface. It is possible to watch the background simulation values by values or graphs only.

- [“Enabling/Launching Double Simulation Mode”](#)
- [“Using Graph in Double Simulation Mode”](#)
- [“Using Watch in Double Simulation Mode”](#)
- [“Handling Breakpoints, Assertions and Exceptions”](#)

Enabling/Launching Double Simulation Mode

You can compare models in double simulation mode. You first have to create the simulation library corresponding to the model simulated in background. Then you activate double simulation in the simulation executable of the main model before launching simulation.

- [“To create the simulation library of the background model”](#)
- [“To activate and launch double simulation from the main model”](#)

To create the simulation library of the background model

- 1 Open the model to be used in background.
- 2 Select a configuration in **Code Generator** toolbar (see [“Managing Default or Custom Configurations”](#) on page 117) or set configuration options from **Code Generator Settings** window. See [“Setting Basic Options for Simulation Code”](#) on page 587 or [“Setting Simulation Binary Generation”](#) on page 589.
- 3 Click **Build**  to create the simulation library of the model.
- 4 Close the project and open the main model in SCADE Suite.

To activate and launch double simulation from the main model

- 1 Click **Settings**  in **Code Generator** toolbar and display **Simulation**.

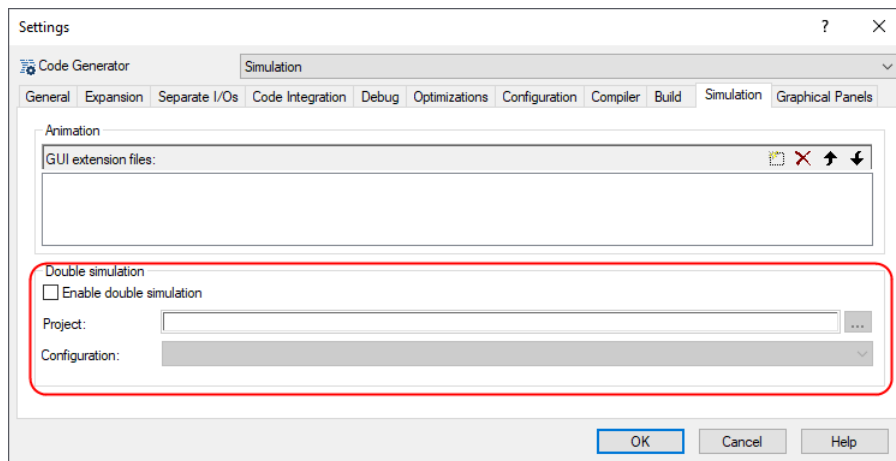


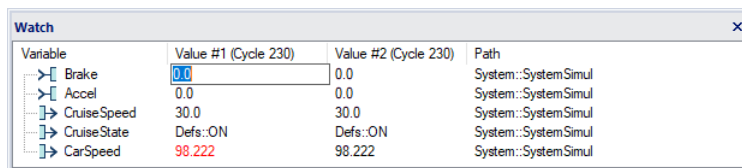
Figure 14.6: Enabling double simulation

- 2 Check **Enable double simulation** and click the browsing button to select the background project for which you built a simulation executable and set the file location in the **Project** field.
- 3 In **Configuration** list, select the configuration set when building the simulation executable of the background model.
- 4 Click **OK** to validate the settings.
- 5 Select a Configuration in **Code Generator** toolbar (see [“Managing Default or Custom Configurations”](#) on page 117).
- 6 Click **Build**  to create the simulation library of the model.
- 7 Click **Run**  to launch double simulation.

Using Watch in Double Simulation Mode

When observing the values of variables while in double simulation mode, the **Watch** window displays a new column containing the values of the background model.

The **Value #1** column corresponds to the values of the main model. The second **Value #2** column corresponds to those of the background model (not editable). Set observable variables in the first column as shown below.



Variable	Value #1 (Cycle 230)	Value #2 (Cycle 230)	Path
Brake	0.0	0.0	System::SystemSimul
Accel	0.0	0.0	System::SystemSimul
CruiseSpeed	30.0	30.0	System::SystemSimul
CruiseState	Defs::ON	Defs::ON	System::SystemSimul
CarSpeed	98.222	98.222	System::SystemSimul

Figure 14.7: Watch window in double simulation mode

For observing selected variables, see [“Displaying Variables in Watch Window”](#) on page 609.

Using Graph in Double Simulation Mode

You can observe simulation graphs using one of the following modes:

- **Standard display** (default): Both graphs are displayed. The graph corresponding to the main model is in bright color while the graph of the background model is in grey.
- **Delta display**: For each point, a graph representing the difference between both values is computed (available only for numeric types).

To switch graphs to delta display

- 1 Select the graph of the variable where you want to observe differences between values computed on both models running in double simulation.
- 2 Right-click and select *Delta Display*.

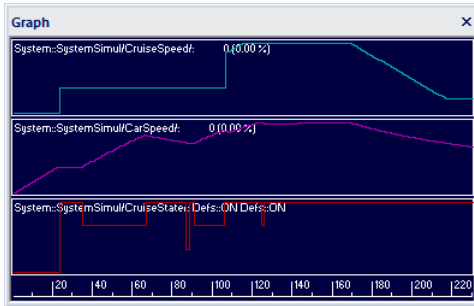


Figure 14.8: Graph window in double simulation mode

For representing selected variables with graphs, see [“Displaying Variables in Graphical Charts”](#) on page 608.

Handling Breakpoints, Assertions and Exceptions

The behavior of SCADE Suite Simulation when encountering breakpoints, assertions, or exceptions in double simulation varies between both main and background models.

Table 14.4: Behavior with main and background models

	Main Model	Background Model
At Breakpoints	Usual behavior: Since SCADE Suite blocks simulation before the end of cycle, the column header corresponding to the main model shows a + sign after the cycle information. The cycle of the background model is unchanged with lastly displayed cycle information.	Cannot be set on such models
On Assertions	Usual behavior: SCADE Suite interrupts simulation and displays a warning. You can reset the simulation.	Displayed on Simulator output tab in same format as normal mode. No simulation pause.
On Exceptions	Usual behavior: SCADE Suite interrupts simulation and displays a warning.	Simulation is blocked and cannot progress any further.

15 /Generating Desktop and Mobile Applications

This chapter explains how to configure and generate desktop or mobile applications for demonstration, test, or validation purposes.

- [“Application Generation Principles”](#)
- [“Generating Desktop Applications”](#)
- [“Generating Mobile Applications”](#)

Application Generation Principles

SCADE Suite Code Generator provides code generation capabilities for the automatic generation of Windows desktop or mobile device applications (tablet or smartphone). The applications are built out of a combination of SCADE Suite models and graphical panels designed in SCADE Display® and Rapid Prototyper.

SCADE applications can be generated for the following target platforms:

- *Windows*: generated desktop applications (. exe) can run on any Windows® PC
- *Android*: generated mobile applications (. apk) can run on any Android™ compatible mobile device
- *iOS*: generated mobile applications can run on any Apple® iOS compatible mobile device (including iPod®, iPhone® and iPad®)

The generation of SCADE applications relies on the connections of SCADE Suite I/Os to the graphical panels I/Os, as explained in [“Developing Control and Display Software”](#) on page 302.

SCADE applications can then be shared with other users for demonstration, test, or validation.

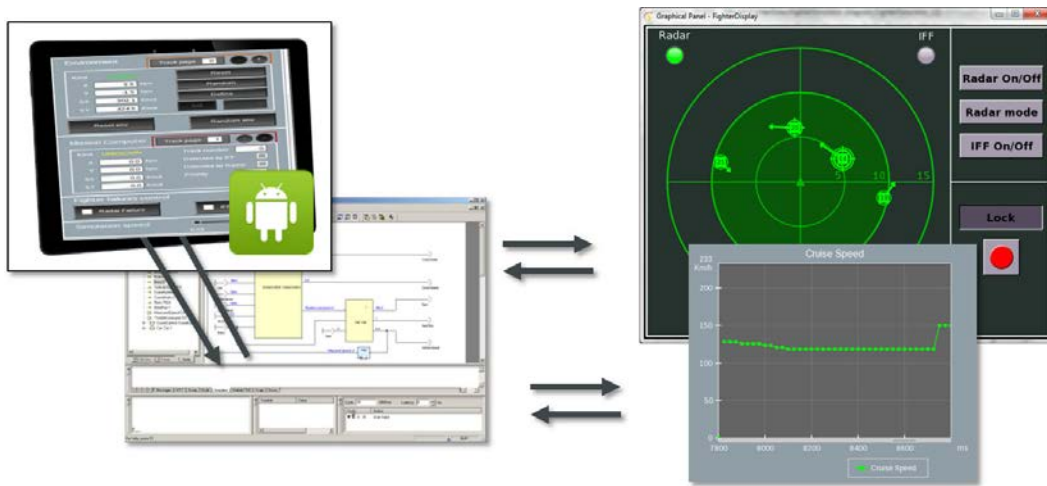


Figure 15.1: Example of desktop applications generated out of SCADE models

Generating Desktop Applications

You can set and launch automatic generation of Windows executable applications.

- [“Setting Windows Executable Generation”](#)
- [“Running Generated Windows Executable”](#)
- [“Generating Windows Executable”](#)
- [“Customizing Windows Executable”](#)

Setting Windows Executable Generation

You can configure SCADE Suite Code Generator for the automatic generation of Windows executables. You can conveniently handle application generation by applying options preset in configurations. For instructions, see [“Managing Default or Custom Configurations”](#) on page 117.

To set code generator for Windows executable generation

- 1 Reference the graphical panels in the project. For instructions, see [“Inserting Graphical Panels in Project”](#) on page 305.

- 2 Connect the I/Os between the SCADE Suite model and the graphical panels. For instructions, see [“Connecting Operator and Graphical Panel Interfaces”](#) on page 306.
- 3 Click **Settings**  in **Code Generator** toolbar and display **Code Integration**.
- 4 Set **Target** and select **Standalone Executable** in the corresponding drop-down list.

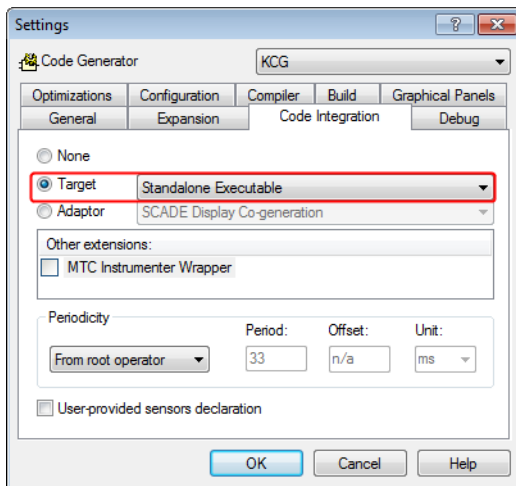


Figure 15.2: Selecting standalone executable target from code integration settings

- 5 Specify the periodicity of the code execution either at root operator level or override root operator periodicity. For details, see [Periodicity](#) option.

Note

The defined periodicity is used as minimum time (converted to milliseconds) between each beginning of cycle of the standalone executable.

- 6 Enable sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked). For details, see [User-provided sensors declaration](#).

- 7 Display **Compiler** to set compiler-specific options and **Build** to set specific build options. **CPU type** for compiler is automatically set to **win64**. For details, see [“Setting Compiler and Build Options”](#) on page 853.
- 8 Display **Graphical Panels**. For each SCADE Display or Rapid Prototyper graphical panel in the project, select the preset configuration to be used for generation, or select **None** if you do not want to include the graphical panel in the generation.

Note

The available configurations are those defined in the associated SCADE Display project or Rapid Prototyper project.

- For information on SCADE Display configurations, see [“Setting Windows Configuration”](#) on page 496 in *SCADE Display User Manual*.
 - For information on Rapid Prototyper configurations, consult *Rapid Prototyper User Manual*.
-

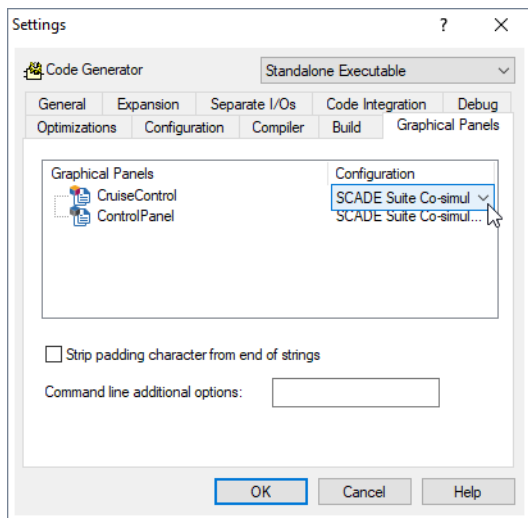


Figure 15.3: Setting graphical panel configuration for standalone executable generation

- 9 Check **Strip padding character from end of strings** to prevent copying padding characters added at the end of strings in the generated code.

Generating Windows Executable

You can launch automatic generation of the standalone executable.

To generate Windows executable

- 1 Select the configuration preset for Windows executable generation from **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119.
- 2 If no specific root operator is specified in the code generator settings, select the operator to be used as root operator from the **Scade View**.
- 3 From **Code Generator** toolbar:
 - Click **Build**  to generate code and build the Windows executable
 - Click **Rebuild All**  to clean the target directory, generate code, and rebuild everything.

For details about code generation, see Chapter 19 about [“Setting and Launching Code Generation”](#).

STANDALONE EXECUTABLE GENERATED FILES

The files are generated and stored in the directory specified from the **Target directory** field in **General** settings of Code Generator. You share the standalone executable by copying the following files:

- <PackageName_RootOperatorName>.exe
- Fonts.dll
- <SDYi_GraphicalPanelName>.dll where *i* is an incremental number depending on the number of graphical panels specified in the code generator settings.

To share the Windows application

- 1 Copy the files listed in the previous section in the same folder into the PC on which you want to run the SCADE application.
- 2 Double-click the copied EXE file to launch the application.

Running Generated Windows Executable

You can run the generated application on any Windows PC provided the executable files are copied in the same folder (as described in previous section). This enables to share quickly and easily the application with other users for demonstration, test, or the validation of the application.

To run the generated application on Windows

- 1 Browse you PC to the folder specified in the **Target directory** field of **General** settings of Code Generator.
- 2 Double-click the EXE file to launch the application.

At run-time, you can switch into full-screen mode.

To switch Full Screen mode

- 1 Press F11 to switch to Full Screen without distortion.
- 2 Press again F11 to switch to complete Full Screen with distortion.
- 3 Press again F11 to exit Full Screen.

Customizing Windows Executable

You can customize the standalone executable you created and integrate it to the simulation environment so that, e.g. it takes the inputs from Simulator. The **main** file as well as the `Makefile.bat` are available in the Windows executable target directory, specified by the SCADE Suite project.

Warning

Before modifying the standalone executable generated files, it is highly recommended that you make a copy of the content of the target directory so that you do not lose all changes if the standalone executable is generated again from SCADE Suite.

To customize the standalone executable

- 1 Create a new project in the C compiler.
- 2 Add all the standalone executable files contained in the directory, as well as other files required for the customizing.
- 3 Change the main file as you want and compile the standalone executable with the modified main and the new files.

Generating Mobile Applications

You can set and launch automatic generation of mobile applications.

- [“External Tools Requirements for Mobile Generation”](#)
- [“Launching Mobile Application Generation”](#)
- [“Setting Mobile Application Generation”](#)
- [“Building Mobile Application”](#)

External Tools Requirements for Mobile Generation

Various tools can be used for Android and iOS application generation from SCADE Suite models. Mobile application generation supports the following Android and IOS tools:

Android Tools	Version	iOS Tools	Version
Java JDK	7u40	iOS	8.x
Android SDK	API 17	Xcode SDK	6.x
Android NDK	r8e		
Apache Ant	1.9.2		

For **Android** generation:

The following environment variables must also be set:

- SDK_ROOT=C:\ . . . \PATH_TO_SDK_DIRECTORY
- NDKROOT=C:\ . . . \PATH_TO_NDK_DIRECTORY
- ANT_HOME=C:\ . . . \PATH_TO_APACHE_ANT_DIRECTORY

If you want generated Android applications to be automatically installed on an Android-powered device at the end of the build process, the following conditions must be fulfilled:

- Google USB driver is installed on the PC
- “USB Debugging” mode is enabled on the Android-powered device
- the Android-powered device is connected to the PC

For **iOS** generation:

You need to be enrolled to the iOS Developer Enterprise Program to be able to develop proprietary applications for in-house distribution.

Setting Mobile Application Generation

You can configure SCAD Suite Code Generator for the automatic generation of mobile applications. You set application generation by applying options preset in configurations. For instructions, see [“Managing Default or Custom Configurations”](#) on page 117.

Warning

If the SCAD Suite project includes external files, you must add these files by hand into your Xcode project (on Mac station) to generate an application for an iOS target.

To set code generator for mobile application generation

- 1 Reference the graphical panels in the project. For instructions, see [“Inserting Graphical Panels in Project”](#) on page 305.
- 2 Connect the SCAD Suite I/Os to the graphical panels I/Os. For instructions, see [“Connecting Operator and Graphical Panel Interfaces”](#) on page 306.
- 3 Click **Settings**  in **Code Generator** toolbar and display **Configuration**.

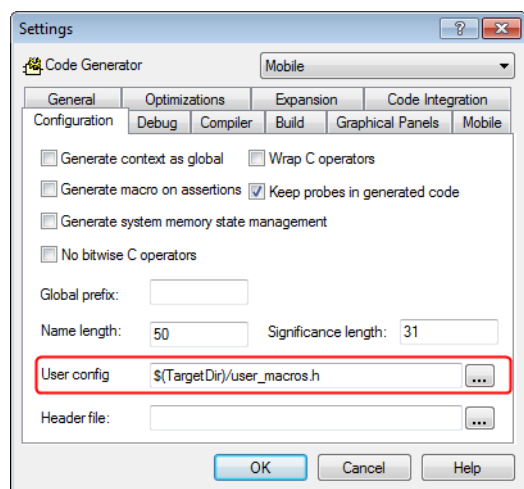


Figure 15.4: Specifying user configuration file for mobile generation

- 4 In the **User config** field, set a user configuration file by typing `$(TargetDir)/user_macros.h`, or browse for files and specify an H-file to be included at the beginning of the `kcg_types.h` file. `$(TargetDir)` is the internal environment variable that references the name of the directory where the generated code is stored.
-

Warning

This option is used by SCADE Display for macros redefinitions. If no user configuration file is set, the generated code cannot compile correctly.

5 Display **Code Integration**.

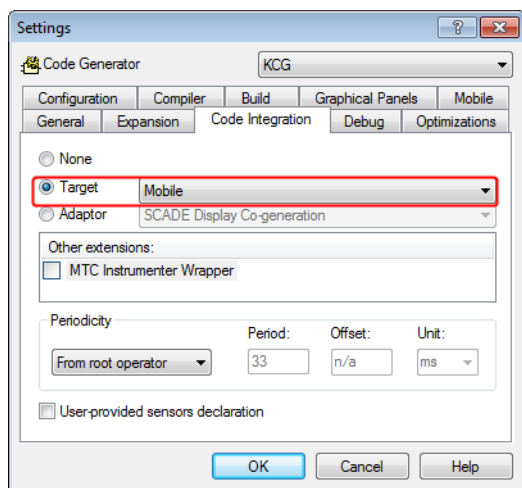


Figure 15.5: Selecting mobile target from code integration settings

- 6 Set **Target** and select **Mobile** in the corresponding drop-down list.
- 7 Specify the periodicity of the code execution either at root operator level or override root operator periodicity. For details, see [Periodicity](#) option.
-

Note

The defined periodicity is used as minimum time (converted to milliseconds) between each beginning of cycle of the standalone executable.

- 8 Enable sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked). For details, see [User-provided sensors declaration](#).
- 9 Display **Mobile**.

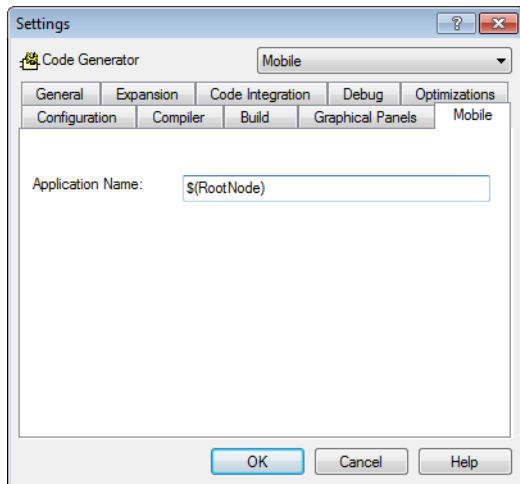


Figure 15.6: Setting application name for mobile application generation

- 10 Define application name in the **Application Name** field (root operator name by default).
- 11 Display **Compiler** to set **CPU type** for generating 32-bits or 64-bits applications and set compiler-specific options. Display **Build** for specific build options (see [“Setting Compiler and Build Options”](#) on page 853).
- 12 Display **Graphical Panels**.

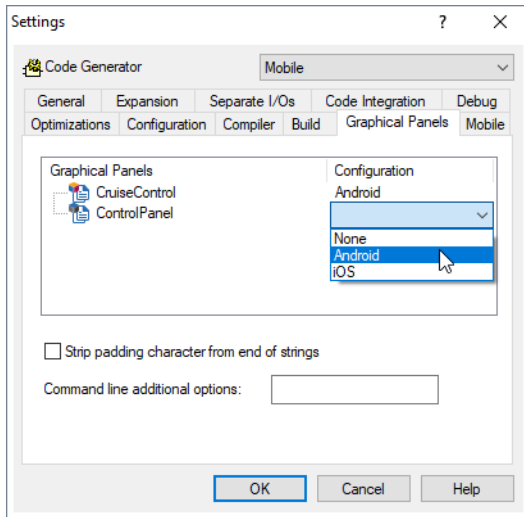


Figure 15.7: Setting graphical panel configuration for mobile application generation

13 For each SCADE Display or Rapid Prototyper graphical panel included in the project, select the preset configuration to be used for generation, or select **None** if you do not want to include the graphical panel in the generation.

Note

The available configurations are those defined in the associated SCADE Display project or Rapid Prototyper project.

- For information on SCADE Display configurations, see [“Setting Android Configuration”](#) on page 501 in *SCADE Display User Manual*.
- For information on Rapid Prototyper configurations, consult *Rapid Prototyper User Manual*.

Launching Mobile Application Generation

You can launch automatic generation of the mobile application from SCAD Suite user interface using SCAD Suite KCG Code Generator.

To generate mobile application code

- 1 Select the configuration preset for mobile application generation from **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119.
- 2 If no specific root operator is specified in the code generator settings, select the operator to be used as root operator from the **Scade View**.
- 3 From **Code Generator** toolbar:
 - Click **Build**  to generate code and build the Windows executable
 - Click **Rebuild All**  to clean the target directory, generate code, and rebuild everything.

For details about code generation, see Chapter 19 about [“Setting and Launching Code Generation”](#).

GENERATED FILES

The files are generated and stored in the directory specified from the **Target directory** field in the **General** settings of Code Generator. You share the mobile application by copying the following file:

- <PackageName_RootOperatorName>.apk for the Android target
- <PackageName_RootOperatorName>.xcodeproj for the iOS target

Building Mobile Application

Applications for Android mobile platforms can be built right from SCADE Suite Editor. For Apple iOS, the applications must be built from a Mac station.

Note

Android application generation was tested with the Android framework API Level 8.

- [“Building Android application”](#)
- [“Building iOS application”](#)

Building Android application

You can build the Android application automatically from SCADE Suite Editor. At the end of the build process, the built application can be automatically installed and run on an Android-powered device under the following conditions:

- Google USB driver is installed on the PC
- “USB Debugging” mode is enabled on your Android-powered device
- the target device is connected to the PC

To build Android application

- From **Code Generator** toolbar:
 - Click **Build**  in **Code Generator** toolbar to build the generated code.
 - Click **Rebuild All**  to clean the target directory, generate code, and rebuild everything.

For details about code generation, see Chapter 19 about [“Setting and Launching Code Generation”](#).

Building iOS application

You can build an iOS application from a Mac station.

To build an iOS application from a generated XCode project

- 1 Copy the output directory to a Mac station.
- 2 Open generated XCode project (.xcodeproj) in XCode®¹ and select *Product > Build*. Click **Run** on the simulator or target.

1. XCode is Apple's Integrated Development Environment for creating iOS compatible applications.

Part 3

Verification

- Chapter 16: [“Running Formal Verification Sessions”](#)
- Chapter 17: [“Verifying Timing and Stack Usage”](#)

16 /Running Formal Verification Sessions

This chapter introduces SCADE Suite Design Verifier and explains how to use this tool to prove properties satisfied by a SCADE Suite model.

- [“Overview of Design Verifier”](#)
- [“Fundamentals in Design Verification”](#)
- [“Setting Design Verifier Environment”](#)
- [“Configuring User Property Verification”](#)
- [“Applying Predefined Properties”](#)
- [“Specifying User-Defined Strategies”](#)
- [“Enabling Analysis Using Abstraction”](#)
- [“Organizing Design Verifier Analysis Objects”](#)
- [“Launching Property Analyses”](#)
- [“Interpreting Property Analyses Results”](#)
- [“Defining Custom Strategies”](#)

Overview of Design Verifier

The purpose, the workflow, and the user interface of Design Verifier supporting the formal verification of SCADE Suite models are briefly presented below.

- [“Purpose of Design Verifier”](#)
- [“Design Verifier Workflow”](#)
- [“Exploring Design Verifier User Interface”](#)
- [“Generated Output Files”](#)

Purpose of Design Verifier

Design Verifier is a formal verification tool which helps SCADE Suite designers verify that their design implements its requirements. This tool supports verification activity and offers an alternative to test and simulation. It can also be used during the design allowing an early detection of errors. Compared to standard testing procedures, Design Verifier proofs have the strength of an exhaustive exploration of the design state space. With Design Verifier, you have access to proof engine that helps validate SCADE Suite designs, performs extended debugging, or just finds out if a given property is fulfilled. Design Verifier can verify properties expressed in Scade language as Boolean formulas using both combinatorial and temporal operators.

- [“Concept of Proof”](#)
- [“Operator Equivalence Checking”](#)

CONCEPT OF PROOF

Consider a SCADE Suite design that implements a cruise control application. You want to find out if the design respects the following property: “*When the brake is pressed, the cruise control must be interrupted or OFF.*” This property can be represented by an operator, called the “*observer*” (see [“Property Expression”](#) on page 729), taking as inputs the data needed to express the property, and combining these inputs into a single Boolean output expressing the property as shown below.

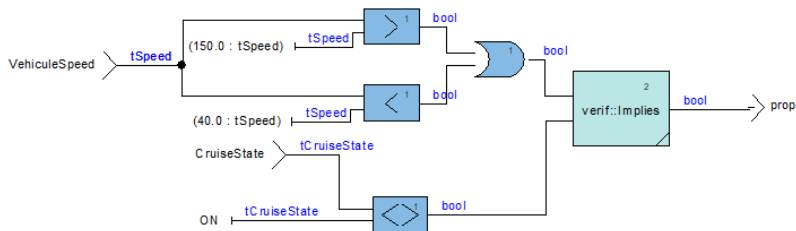


Figure 16.1: Graphical design of a property in an observer operator

In a **top-level operator**, you connect the inputs/ outputs of the model under verification with the observer operator(s) used to verify the expressed properties (see [“Property Expression”](#) on page 729). As shown below, the top-level operator takes all the inputs of the system and produces a single Boolean output from each observer operator.

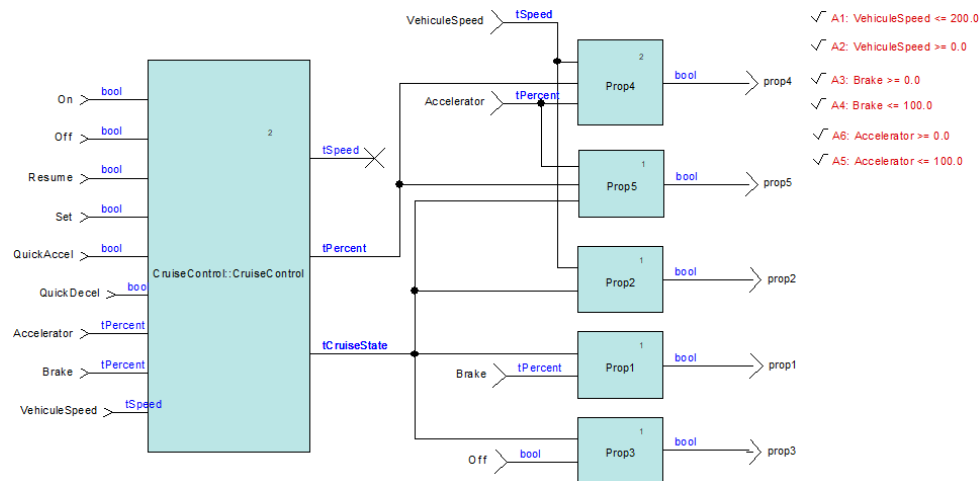


Figure 16.2: Graphical design of a top-level operator with property observer operators

Proving the property consists in proving that, for all input sequences of values taken in the declared input types, the output of the observer is true (*i.e.*, a sequence of true). If Design Verifier proves the property on the model then you know this property holds on the implementation. In cases where the property does not hold, Design Verifier provides a counter-example that can be played step by step by SCADE Suite simulation to understand the problem in the design. In most cases, the expression of the proof of the property is not intrusive; you do not modify the design, but simply connect it to observers.

Note

In some cases, to access data deeply buried into sub-operators in the design, you may have to add outputs to sub-operators and propagate the data up to the top-level operator. For details, see [“Designing Context in Top-Level Operator”](#) on page 739.

OPERATOR EQUIVALENCE CHECKING

Design Verifier can also be used to prove the equivalence of two operators. You simply create a new operator with an instance of each operator to compare and compare their outputs as illustrated in [Figure 16.3](#) below. If Design Verifier succeeds in proving equivalence, both operators are functionally equivalent and can be used equally in any context. Notice that this equivalence is only functional: it says nothing about their relative timing or memory consumption performance.

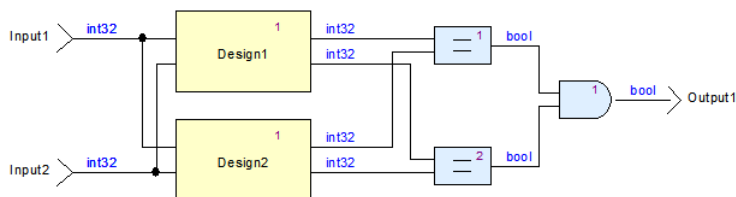


Figure 16.3: Equivalence checking principle between SCADE Suite designs

Design Verifier can be a useful tool in many other cases. For example, in **degraded mode analysis** or in **non-regression analysis**, you may replace the definition of an operator with a design modeling its behavior in degraded mode and then prove that some safety properties still hold in such degraded mode.

Design Verifier Workflow

The workflow of Design Verifier consists of successive user-tasks that you can iterate as much as needed. Once design specifications and system safety requirements are identified, the verification of any SCADE Suite design with Design Verifier follows several phases as described in the figure below:

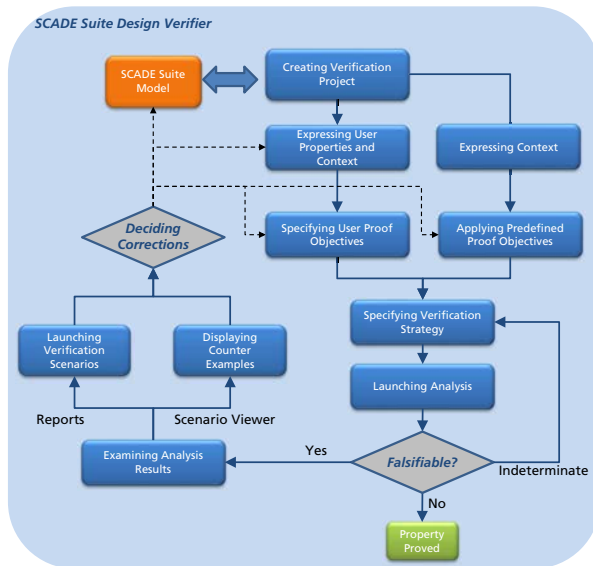


Figure 16.4: Design Verifier workflow

Each block corresponds to a specific phase as described below:

- 1 Expressing the property to verify via observer operators and specifying its context: see [“Expressing User Property and Context”](#) on page 738
- 2 Creating proof objectives for each property to be verified: see [“Specifying User Proof Objectives”](#) on page 740
- 3 Using predefined proof objectives (absence of division by zero, overflows): see [“Setting Predefined Proof Objectives Properties”](#) on page 744
- 4 Setting strategy: see [“Specifying User-Defined Strategies”](#) on page 746

- 5 Analyzing properties: see [“Launching Property Analyses”](#) on page 756
- 6 Exploiting results and counter-examples: see [“Interpreting Property Analyses Results”](#) on page 758.
- 7 Loading simulation input files in SCADE Suite Simulation: see [“Launching Verification Scenarios from Reports”](#) on page 761

The workflow may vary depending on how you initiate the verification session and from which sort of SCADE Suite project. You can define the properties and run verification in the same project as the system itself, or in a separate project using the system’s project as a library. For more details, see [“Setting Design Verifier Environment”](#) on page 736.

Exploring Design Verifier User Interface

The interface of SCADE Suite Design Verifier contains various components for defining and managing formal verification elements, launching analysis, analyzing results and proof information, and reporting results.

- [“Design Verifier Toolbar”](#)
- [“Design Verifier Analysis Reports”](#)
- [“Design Verifier View”](#)
- [“Scenario Viewer”](#)

Design Verifier Toolbar

This toolbar provides shortcuts to create proof objectives, and verification strategies required to run formal property verification. It can be moved in the interface by dragging the title bar to the desired location.

Design Verifier	Managing and defining verification elements: • “Specifying User Proof Objectives” on page 740 • “Applying Predefined Properties” on page 742 • “Specifying User-Defined Strategies” on page 746

To show/hide toolbar

- Select *View > Toolbars > Design Verifier*, or right-click in toolbar area and select the name of the toolbar to show/hide. Alternatively, hide the toolbar by right-clicking on toolbar and selecting *Hide*.

Design Verifier View

The **Design Verifier View** displays all items needed to set property verification and launch proofs as illustrated below.

- User Proof Objectives (user-defined property)
- Predefined Proof Objectives (properties based on default checks)
- Strategies (default and user-defined)

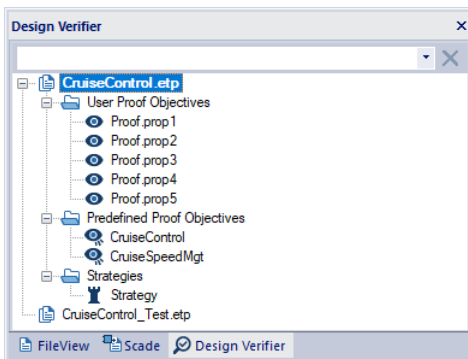


Figure 16.5: Design Verifier View

Tip

Whenever removing an operator from the SCADE Suite model, all associated proof objectives are disabled and crossed-over in red.

You can organize verification elements in a folder hierarchy where you navigate like in any other views. Design Verifier allows to create *task folders* through which one can associate proof objectives or task folders with strategies and further run analysis.

Depending on how you initiate the verification project, Design Verifier may not display in SCADE Suite workspace. If this is the case, see [“Setting Design Verifier Environment”](#) on page 736.

RELATED TOPICS

- [“Organizing Design Verifier Analysis Objects”](#) on page 752
- [“Applying Predefined Properties”](#) on page 742

Design Verifier Analysis Reports

In **Analysis Reports**, you visualize analysis results and proof information. Once the analysis of all requested proof objectives completes, Design Verifier generates a report and displays it automatically.

In the report, you find general information about the analysis (*i.e.*, user name of the person that performed the analysis, date of analysis, analyzed model name) and its result. You also find specific information about the settings of proof objectives and strategies. You finally find the analysis time.

Tip

While navigating in the report, you can browse back using contextual menu options.

PPI Analysis Report x	
Tasks	
Proof.prop5	
Strategies	
Prove (default settings)	
General Info	
Time of analysis	Monday, April 26, 2021 09:44:03
Model	CruiseControl
User	James
Sum Up	
Proof.prop5	Valid
Tasks	
Proof.prop5	
Operator	Proof::Proof
Output	prop5
Strategy	Prove (default settings)
Result	Valid
Location	none
Analysis time	65 s
Assertions	none
Messages	none
Strategies	
Prove (default settings)	
Base strategy	Prove

Figure 16.6: Analysis Report

Analysis Reports may also contain information about simulation input files. Provided you enable a simulation configuration, you can load such scenarios in SCADE Suite Simulation directly from the generated report. You can also access any scenario from previous property analysis and launch them in simulation. Design Verifier generates scenarios containing only input values relevant for property falsification and archives all scenarios in the directories of the SCADE Suite projects.

Note

You find reports generated by Design Verifier in `<Project_dir>/reports/<time_stamp>/index.html`, where `<Project_dir>` is the root directory of the project and `<time_stamp>` is a directory named by the date of the proof.

Proof.prop4

Operator	Proof::Proof
Output	prop4
Strategy	Prove (default settings)
Result	Falsifiable
Scenario	scenarios/Proof.prop4_s0.sss Load Scenario
Location	• Proof::Proof/prop4

Figure 16.7: Link to launch verification scenario directly from Design Verifier reports

When analyzing a proof objective, or a group of proof objectives, a pop-up window shows the work in progress and allows to abort the analysis. This status window displays time information for each phase of analysis and information on proof results. In addition to the report that sums up the analysis and its results, Design Verifier displays proof results in the **Browse** output tab. Design Verifier archives reports as HTML files to allow future use.

The main information in the report relates to the proof results for each proof objective (*i.e.*, property). Possible results are: **Valid**, **Falsifiable**, **Indeterminate**, **Interrupted**, **Raised an Error**, or **Contradictory**. For details about messages, see [“Reading Basic Result Messages”](#) on page 758.

You can run the analysis scenario provided by Design Verifier in SCADE Suite Simulation via active links on the HTML report. For details about the status messages and the use of reports and scenarios, see [“Interpreting Property Analyses Results”](#) on page 758.

Scenario Viewer

Instead of opening text files, **Scenario Viewer** allows to examine graphically the content of the counter-examples found by Design Verifier for analyzed properties. When Design Verifier finds a falsifiable property, it provides you with a counter-example. For using Scenario Viewer, see [“Displaying Counter-Examples in Scenario Viewer”](#) on page 763.

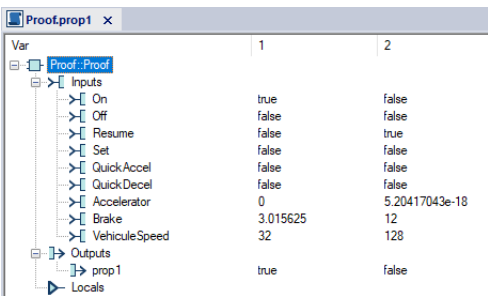


Figure 16.8: Scenario Viewer

Scenario Viewer displays the values taken by inputs/outputs of the top-level operator (for user proof objective) or user operator (for predefined proof objectives) analyzed for the counter-example. If the counter-example extends over several steps, the input/output values for each step are in separate columns. If an input/output is meaningless for the property (*i.e.*, input/output value does not affect the output of the observer), it is not displayed. If at a given discrete step, the value of an input/output is meaningless for the property, a star symbol (*) is displayed. You can also set the level of information displayed in **Scenario Viewer** about inputs, outputs, and local variables (see [“Setting Simulation Binary Generation”](#) on page 589).

Generated Output Files

After Design Verifier analysis, output files are produced under the project directory in a directory named reports/<analysis time> where <analysis time> is the system time of analysis based on <day>_<month>_<date>_<year>_<hours><minutes><seconds> format.

Context of produced files	File Location
SSS scenario file for each counter-example (if any)	reports/<analysis time>/scenarios/
HTML report file	reports/<analysis time>/

Fundamentals in Design Verification

The formal verification of model design with Design Verifier is based on the basic concepts explained below.

- [“SCADE Arithmetic”](#)
- [“Property Expression”](#)
- [“Verifiable Properties”](#)
- [“User Proof Objectives”](#)
- [“Predefined Proof Objectives”](#)
- [“Strategies”](#)

SCADE Arithmetic

Design Verifier implements unbounded integers arithmetic. However, it still operates over fixed-size integers meaning that, for each definition of an integer flow as in “ $x = (2 * y) + 1;$ ”, the right-hand side expression is interpreted in the values domain of the corresponding fixed-size type. Such a model is consistent with SCADE integers arithmetic, as defined in [“Arithmetic Operators”](#) on page 102 in *SCADE Language Reference Manual*, as long as no overflow and no division by zero occur.

Regarding floating-points, Design Verifier relies on the IEEE-754 arithmetics (defined in IEEE Std 754-2008, “IEEE Standard for Floating-Point Arithmetic”). Floating-points are based on the following bits-encoding:

- float32: sign bit, 8 bits used for the exponent, 23 bits for the mantissa
- float64: sign bit, 11 bits used for the exponent, 52 bits for the mantissa

In addition, NaNs, infinities, as well as signed zeros are admitted as correct values.

Property Expression

In Design Verifier verification projects, to prove an operator property, you first have to design an operator implementing this property which takes as inputs the streams involved in the property and produces a single Boolean

output representing the status of the property. This operator is called the *observer operator*. With the help of Scade formalism and SCADE Suite operator libraries, expressing properties is as easy as coding. Yet, some properties, like temporal ones, require some experience to be properly expressed within SCADE Suite. For example, consider the following properties:

- “*The alarm is issued at most three discrete time ticks after the aircraft altitude gets lower than 200 meters and is not in landing state.*”
- “*If the pilot does not clear it, the alarm stays ON for at least ten discrete time ticks after the aircraft altitude gets lower than 200 meters and the aircraft is not in landing state.*”

Expressing both properties requires some knowledge in writing temporal logic formulas with Scade. To help writing the expression of properties, Design Verifier is distributed with a built-in library containing a set of intuitive and powerful operators. For details, consult Chapter 9 about [“Library libverif”](#) in *SCADE Suite Libraries Manual*.

Once properties are expressed in *observer operators*, you create another operator combining the context of the system design with all observer operators correctly connected to the inputs/outputs of the context. This operator is the *top-level operator*. It must have the same inputs as the design operator and one Boolean output per property.

Note

An observer can implement several proof objectives: one per Boolean output of the observer.

Verifiable Properties

Design Verifier can verify properties that do not depend on flows of imported types. Imported types are interpreted depending on the associated constraint as detailed in the following table:

Imported Type Constraint	Design Verifier Interpretation
none	int64
numeric	int64
integer	int64
signed	int64
unsigned	uint64
float	float64

Notice that imported constants are interpreted as free variables¹: proofs are made without any hypothesis on such constants except that they keep the same value during execution. Imported functions and nodes are considered at instance-level: the outputs of a function or operator instance are interpreted as free variables as well.

RELATED TOPICS

- [“Setting Imported Types”](#) on page 214

1. Free variables are unbounded variables that can get all possible values depending on type.

User Proof Objectives

After expressing user properties with a top-level operator and associating a user proof objective with it, you must also select which verification [strategy](#) to apply during analysis. Once the property to prove is properly designed and specified, you can launch Design Verifier analysis to determine whether the user property is valid or falsifiable.

Predefined Proof Objectives

Design Verifier can be used to check predefined proof objectives. These proof objectives and their conditions of application are listed below:

- **Integer Overflow:** checks if all arithmetic expressions always fit in the range of their respective type. Checking the absence of overflow guarantees that the arithmetics of a model remains within the definition domain of the Scade language. The way integer expressions are handled when verifying user proof objectives does not prevent sub-expressions to overflow. To cope with this, expressions are normalized: each sub-expression is extracted leading to new definitions of the "v = x1 bop x2" form where x1 and x2 are fresh identifiers. Compiling a model this way guarantees that each arithmetic expression is checked for overflows and that the model is in the definition domain of Scade language.

Operator	Type	Condition for overflow
Plus, Minus, Multiplication (+, -, *)	intN/uintN	The resulting expression is not in the range of the type
Polymorphic Division (/)	intN	The numerator is equal to the minimum value of the type and the denominator is equal to -1
Logical Left/Right Shift (lsl and lsr)	uintN	The right operand is greater than or equal to the size of the integer type
Numeric Cast (e : t)	intN/uintN	((e : t) : t _e) <> e

- **Integer Division by zero:** checks if the denominator of each division expression is always different from zero. Checking the absence of division by zero guarantees that there are no run time errors in the model. Division by zero differs between floating points and integers in the sense that integer division does not admit an encoding value. On the contrary, according to IEEE-754 standard, division by zero is always defined:
 - it evaluates to NaN if both numerator and denominator equal 0.0;
 - it evaluates to +/- infinity if only the denominator equals 0.0.

Operator	Type	Condition for division by zero
Polymorphic Division (/) and Modulo (mod)	intN or uintN	The right operand is null

- **Float Infinity:** checks that all floating-point expressions never evaluate to infinity.
- **Float Not a Number:** checks that all floating-point expressions never evaluate to Not a Number (NaN).

Note

With respect to IEEE-754 exceptions, the “Infinity” and “Not a Number” checks allow to guarantee the absence of overflow, division by zero, and invalid operations.

- **Dynamic access:** checks if the default value of a dynamic access expression is used or not. Checking the absence of dynamic index overflow allows to identify the “dead parts” (uncoverable) of a model.

Operator	Type	Condition for uncoverable default value
Dynamic projection (a.[idx] default v)	array of size N	idx is always in the range [0, N-1]
Assign of a Structured Element (a with ... [idx _i] ... = e)	mixed structure/ array	all idx _i along the path are always in the range [0, N _i -1], where N _i is the size of the array

- **Consistency:** checks if constraints do not contradict each other. Constraints consistency can only be checked up to a certain depth.

Any predefined proof objectives you generate must be associated with a [strategy](#).

Strategies

For running property analysis, you must select which strategy to apply. While defining proof objectives, you select one of the strategies provided with Design Verifier or you can set your own strategy (see [“Specifying User-Defined Strategies”](#) on page 746).

Table 16.1: Objective of strategies available in Design Verifier

Strategies	Description
Prove	Tries to prove the validity of properties until it either succeeds or finds a counter-example.
Debug (default)	Analyses the property up to a stop cycle. In case the property can be falsified before reaching the stop cycle, Design Verifier returns the corresponding counter-example.
Induction	Tries to prove the property using the induction principle (bases cases and the induction step). A minimum and maximum induction depth can be specified. It also provides counter-example when the property is not valid.
Custom	Tries to analyze the property based on a higher-level strategy that is defined as a combination of simpler ones. The syntax used to build such a strategy is based on Prover PSL strategy definition syntax. For details, see “Defining Custom Strategies” on page 765.

The **Debug strategy** is meant to be used during development, the **Induction strategy** is meant in particular to perform non-regression when knowing the induction depth, and the **Prove strategy** is meant to validate a design once development is stable.

After selecting a strategy, you configure options. The **Timeout** option sets the amount of time allocated to Design Verifier to prove the objectives. If it finds no counter-example in that time and if the proof is incomplete, the process stops, and the result is undetermined. Specific options like **Start depth** and **Stop depth** control how many analysis steps Design Verifier can use to search a counter-example. To find counter-examples, Design Verifier searches over the depth set by the option values.

RELATED TOPICS

- [“User Proof Objectives”](#) on page 732
- [“Predefined Proof Objectives”](#) on page 732

Setting Design Verifier Environment

You can run verification sessions with Design Verifier on SCADE Suite projects directly on SCADE Suite models or through dedicated verification projects.

Using verification projects associated with SCADE Suite models allows to store the design of observers without modifying SCADE Suite models.

- [“Creating Verification Project”](#)
- [“Enabling Design Verifier from SCADE Suite Projects”](#)

Creating Verification Project

You can create a verification project to use Design Verifier for proving properties in SCADE Suite models by referencing the SCADE Suite project as a library.

To create a new verification project

- 1 Select *File* > *New* to open the **New** window and launch the project wizard.
- 2 Choose **SCADE Suite Project for Design Verifier** in the list of projects.

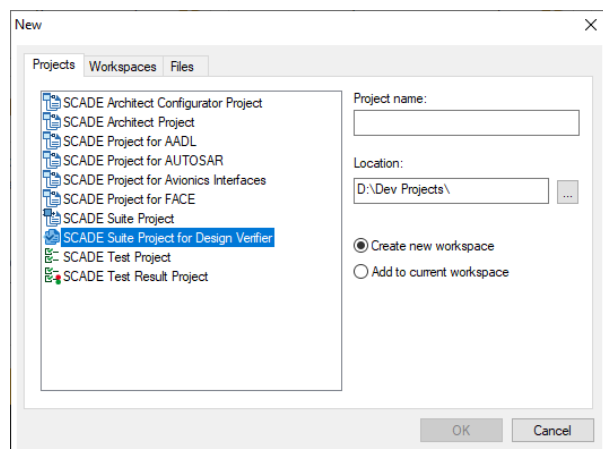


Figure 16.9: Launching the Design Verifier Project wizard

- 3 Name the project in **Project name**.

- 4 Browse for specifying the location of the file in **Location**. Click **OK**

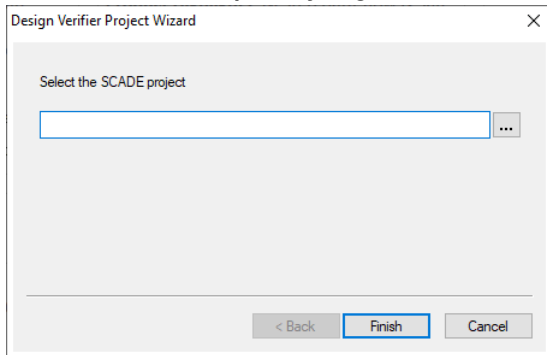


Figure 16.10: Selecting the existing project used as verification project library

- 5 Select the project used as library in the verification project and click **Finish**.

Enabling Design Verifier from SCADE Suite Projects

You can enable Design Verifier from existing SCADE Suite projects. Once enabled, Design Verifier can be used directly on the project. You do not need to create a verification project and select the SCADE Suite project as library.

To enable Design Verifier

- 1 Open the SCADE Suite project and select *Project > Enable Design Verifier*.
- 2 In **New Design Verifier File** window, specify the following:
 - In **Directory**, set the location of the Design Verifier file. Default is project directory.
 - In **File name**: rename the Design Verifier file if needed. Default is <ProjectName>.l4.
- 3 Click **OK**. The **Design Verifier View** appears and SCADE Suite adds the Design Verifier file to the project in **File View**.

Configuring User Property Verification

This section explains how to express properties in a design and how to specify user proof objectives to run formal verification on SCADE Suite models.

- [“Expressing User Property and Context”](#)
- [“Specifying User Proof Objectives”](#)

Expressing User Property and Context

The description of the property to verify within a specific design context is necessary. You must express a user property in an observer operator and describe its context in an top-level operator.

- [“Designing Observer Operators for Property Expression”](#)
- [“Designing Context in Top-Level Operator”](#)

Designing Observer Operators for Property Expression

The properties must be verifiable (see [“Verifiable Properties”](#) on page 731). In a verification project, you formalize each of the properties that you want to prove as a SCADE Suite operator (*i.e.*, observer operator) combining all needed inputs into a single Boolean output expressing the property.

To design observer operators

- 1 In **Scade View**, right-click the Operators folder, select *Insert > Operator*, and rename it as appropriate.
- 2 Right-click the added operator and select *Insert > Input* for each input.
- 3 Right-click the added operator and select *Insert > Output* to set the output expressing the property.
- 4 Open the diagram of the operator and represent its design graphically.
- 5 Connect correctly all inputs to required operators and the property output.
- 6 Proceed with the representation of the context in a top-level operator.

Designing Context in Top-Level Operator

In a verification project, you need to represent the design of all observer operators in their context. You create a *top-level operator* by connecting the operator under verification and the observer operators containing the properties to be verified. The inputs of the observer operator are a subset of the inputs and outputs from the operator under verification; each observer operator has a single Boolean output for each property.

To design top-level operator

- 1 In **Scade View**, right-click the Operators folder, select *Insert > Operator*, and rename it *VerificationDesign* for instance.
- 2 Right-click the operator, select *Insert > Input* for each input, and select *Insert > Output* to set outputs representing each property.
- 3 Open the diagram of the top-level operator and represent its design with the observer operator(s) and outputs needed in the verification session.
- 4 Connect correctly each observer operator to its respective inputs/outputs.

Note that the property to express may depend on internal data of the operator to verify. In this case, you must create a dedicated output for verification of this operator to give access to this data. This output must be propagated up to the top operator. To avoid considering values that are known to be unreachable, the user can define **assertions** on the model inputs. Such assertions can be used to capture various assumptions such as:

- value ranges and limits on rates,
- or relations between inputs information.

Warning

Note that any fact is provable with respect to an inconsistent theory. Be careful not to introduce contradictions in the model when inserting hypothesis or assumptions.

Specifying User Proof Objectives

Before verifying selected properties, it is necessary to create a user proof objective for each built-in property that needs to be proved in the design. User proof objectives can be inserted on observer operator outputs from any view in the project browsers or from diagrams. Then, each user proof objective must be set with the strategy to apply during property analysis.

To create and set a user proof objective

- 1 In **Scade View**, expand each *top-level operator* to display their interface components, right-click a property output (of Boolean type) and select *Insert > User Proof Objective* to create a new proof objective under the User Proof Objectives folder in **Design Verifier View**.

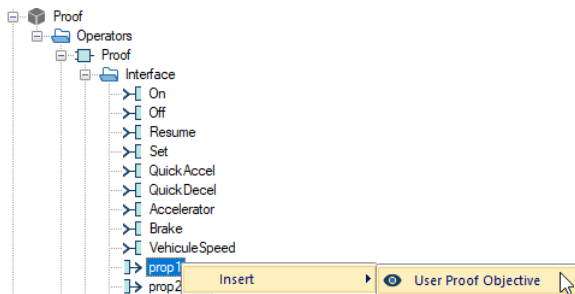


Figure 16.11: Creating the proof objective of a property from SCADe browser

Alternatively, insert proof objectives from the diagram of the top-level operator: right-click a property output instance in **Diagram View** and select *Insert > User Proof Objective* or click **New User Proof Objective**  button in **Design Verifier** toolbar. It is possible to insert proof objectives by multi-selection of property outputs.

- 2 In **Design Verifier View**, rename the proof objective if needed. Naming pattern is: <TopLevelOperatorName> . <OutputName>.

- 3 Right-click the proof objective and select *Properties* to display **General** properties.

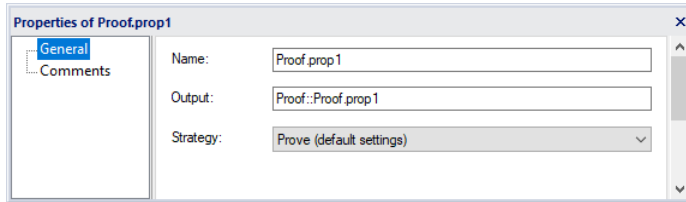


Figure 16.12: Properties of a proof objective

- 4 In **Name**, change the name of the selected proof objective. The **Output** field displays the path of the output associated with the proof objective.
- 5 In **Strategy** drop-down list, select the strategy applied by the proof objective when analyzing the associated output. From a multi-selection of proof objectives in **Design Verifier View**, apply the same strategy to all proof objectives at once. It is also possible to associate the same strategy to several proof objectives through task folders (see [“Associating Strategies with Proof Objectives”](#) on page 754).

Debug strategy with default settings is used by default if no other strategy is set. It is possible to select one of the available strategies with default settings or a user-defined strategy created as explained in [“Specifying User-Defined Strategies”](#) on page 746. For a description of the strategies provided by Design Verifier, see [“Strategies”](#) on page 734.

- 6 Display **Comments** and comment the proof objective.

Applying Predefined Properties

You can set Design Verifier with predefined proof objectives to check statically various predefined properties on SCADE Suite models. To launch analysis on selected operators (directly in models or in context represented in verification projects, you need to specify and set a dedicated proof objective for each operator to analyze.

- [“Specifying Predefined Proof Objectives”](#)
- [“Setting Predefined Proof Objectives Properties”](#)

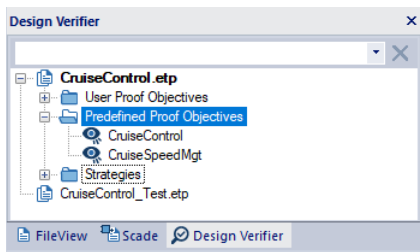


Figure 16.13: Predefined proof objectives organized in Design Verifier View

Launching analysis on predefined proof objectives is described in [“Launching Property Analyses”](#) on page 756.

Specifying Predefined Proof Objectives

The predefined proof objectives provided with Design Verifier allow to verify divisions-by-zero, overflows, infinity, NaN, dynamic accesses, or check the consistency of models. Such proof objectives are specified from operators in any displayed views.

The verification of division-by-zero, overflow, infinity, or NaN detects the possible occurrence of division-by-zero, overflow, infinity, or NaN on any operator. The verification of dynamic access detects the use of the default

expressions in dynamic projections (*i.e.*, the value returned when the index is out of the bounds). The consistency check verifies that the model and the constraints are not contradictory for any execution up to a given depth.

To specify a new predefined proof objective

- 1 In **Scade View**, select the operator(s) to analyze. Design Verifier analyzes automatically all the operators called by the selected operator(s).
- 2 In **Design Verifier** toolbar, click  **Predefined Proof Objectives** to specify a predefined proof objective for the selected operator(s).

New proof objectives are added under the Predefined Proof Objectives folder in **Design Verifier View**. Alternatively, right-click the operator and select *Insert > Predefined Proof Objective*.

- 3 Rename the proof objective if needed.

Predefined proof objectives are set by default with the following checks (Integer Division by Zero, Integer Overflow, Float Infinity, and Float Not a Number) and the Debug strategy. It is possible to modify the default from the **Properties** of each proof objective (see [“Setting Predefined Proof Objectives Properties”](#) on page 744).

Setting Predefined Proof Objectives Properties

You can set the properties of each proof objective with the strategy it applies during analysis. All created proof objectives and their properties (saved in L4 files) are persistent across analysis sessions.

To set predefined proof objective properties

- 1 In **Design Verifier View**, expand folders and select a proof objective.
- 2 Right-click and select *Properties* to open **Properties** window on **General**.

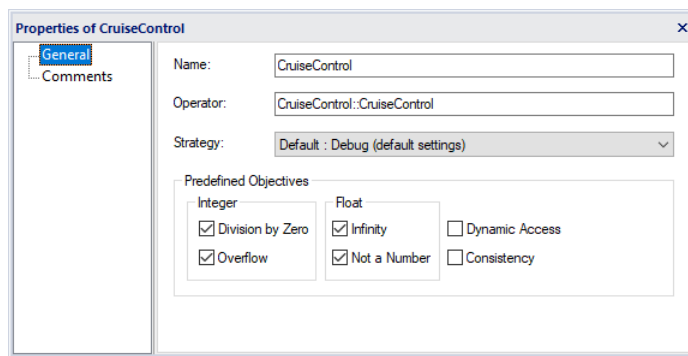


Figure 16.14: Properties of a predefined proof objective

- 3 In **Name**, change the name of the selected proof objective. The **Operator** field displays the path of the operator associated with the proof objective.
- 4 In **Strategy** drop-down list, select the strategy applied by the proof objective when analyzing the associated operator. From a multi-selection of proof objectives in **Design Verifier View**, apply the same strategy to all proof objectives at once.

Debug strategy with default settings is used by default if no other strategy is set. For Consistency proof objectives, only Debug strategies (predefined or user defined) are available. It is possible to select one of the available strategies with default settings or a user-defined strategy created as

explained in [“Specifying User-Defined Strategies”](#) on page 746. For a description of the strategies provided by Design Verifier, see [“Strategies”](#) on page 734.

- 5 Under **Predefined Objectives**, select the type of checks to be applied by the proof objective:
 - **Integer Division by Zero**: This check applies to expressions of type `int<N>` and `uint<N>`. It is generated to ensure that the denominator is always different from 0. For an expression like "`x / y`", the proof objective "`y != 0`" is produced.
 - **Integer Overflow**: This check applies only to expressions of type `int<N>` and `uint<N>`.
 - **Float Infinity**: This check applies to expressions of type `float<N>`. It ensures that the result of an arithmetic expression is not infinite.
 - **Float Not a Number**: This check applies to expressions of type `float<N>`. It ensures that the result of an arithmetic expression is not NaN.
 - **Dynamic Access**: This check helps identifying dead branches of dynamic accesses (dynamic projection or copy with modification). It checks whether the default value of such accesses is used or not. For the index expression "`<e>`", the proof objective "`<e> >= 0` and `<e> < <N>`" is generated where N is the size of the array.
 - **Consistency**: This check is generated to ensure that the assume contracts of the selected operator(s) are consistent before analysis (it is a kind of check on assume contracts).
- 6 Display **Comments** and comment the proof objective.

Specifying User-Defined Strategies

For running property analysis from proof objectives on SCADE Suite models, it is possible to use either the default strategies or user-defined strategies. User-defined strategies must be created first.

To create and set a new user strategy

- 1 In **Design Verifier View**, right-click the **Strategies** folder (or sub-folders) and select *Insert > Strategy* to add a new strategy in the folder, or select any folder and click **New Strategy**  in **Design Verifier** toolbar.
- 2 Right-click the strategy and select *Properties* to display **General** properties.

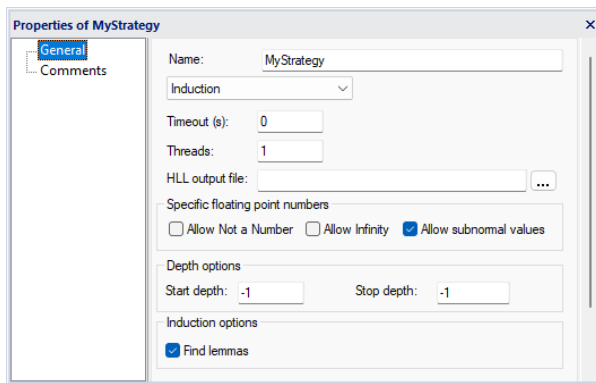


Figure 16.15: Specifying the type of verification strategy

- 3 Rename the new strategy as needed in **Name** field.
- 4 Set **Prove**, **Debug**, **Induction**, or **Custom** strategy to be defined for the user strategy from the drop-down list. For a strategy description, see [“Strategies”](#) on page 734.
- 5 In **Timeout (s)**, set the time (in seconds) during which Design Verifier keeps processing the verification. Default is 0, meaning there is no timeout.

- 6 In **Threads**, set the number of threads per run (within the limit of processor cores available on host). A 0 value means the analysis run is using the maximum number of processor cores.
- 7 In **HLL output file**, specify the text file containing a dump of prover model. The dump file is a public format accessible from [HAL portal](#).
- 8 In **Specific floating point numbers**, use one of the following options:
 - **Allow Not a Number**, **Allow Infinity**, and **Allow subnormal values**: check to specify the class of floating-point values authorized as analysis inputs (root operator inputs, imported operator outputs, or imported constants) for proof analysis and for counter-examples. Without using these options, these floating point class values are ignored from proof analysis and cannot show in counter examples.
- 9 Set the **Depth options** for the selected strategy (only **Debug** or **Induction**):

Strategy	Start depth	Stop depth
Debug	Set the first analysis depth when Design Verifier tries to find a counter-example.	Set the last analysis depth when Design Verifier tries to find a counter-example.
Induction	Set the first depth where the induction "step" case is analyzed.	Set the depth up to which base cases are analyzed.
	Default is -1, meaning the minimum analysis depth is unspecified. In such case, analysis starts at depth 0.	Default is -1, meaning the maximum analysis depth is unspecified. In such case, analysis stops when a proof (Induction) or counter-example (Debug) can be found, when interrupted by user, or when timeout is reached.

To find counter-examples with Debug strategies, Design Verifier searches a counter-example over the depth set by **Start depth** and **Stop depth** values.

- 10 Set the **Induction options** for the selected strategy (only **Induction**):

- **Find lemmas** (default): pre-processing step that strengthens the problem given to the verification engine by automatically identifying and proving additional properties (e.g., stream equivalences). Once strengthened, the problem is more likely to be proved by induction.
- 11** Set the **Prover PSL strategy** (only **Custom**) in **Strategy** field by specifying a string describing the customized strategy to be used by Prover PSL. The string must comply with the Prover PSL strategy definition syntax. For details, see [“Defining Custom Strategies”](#) on page 765.

Enabling Analysis Using Abstraction

When analyzing models containing imported operators, Design Verifier is prone to returning false-negatives (leading to scenarios that cannot falsify the properties). In addition, analysis time can be very long due to the implementation complexity of analyzed operators.

Abstraction operators can be used to replace imported operators by a more constrained behavior as a means to limit the possibility to get false-negatives or to replace complex operators by simpler implementation allowing faster property analysis. For Design Verifier property analysis, abstraction operators must be defined with exactly the same interface (inputs, hidden inputs, outputs, and parameters) and the same node/function property as the operator they abstract. They must be created in a different model as the model to be analyzed. Abstraction operators are not standard SCADE Suite operators:

- their outputs can be partially defined and/or
- they can be characterized using assertions (Assume or Guarantee).

Notice that users are responsible for demonstrating that the abstraction they describe in Abstraction operator is correct.

To create Abstraction operators

- 1 In **Scade View**, right-click the operator to analyze using abstraction and select *Insert > Abstraction Operator*.

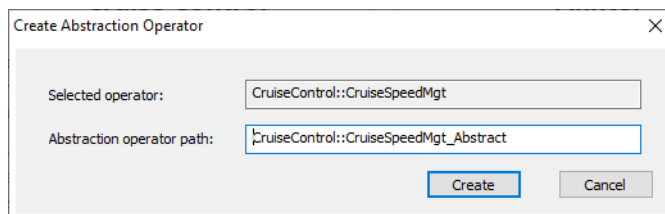


Figure 16.16: Creating Abstraction Operator

- 2 Edit the path to the abstraction operator as needed in **Abstraction operator path**.
- 3 Click **Create** to insert the new Abstraction operator in **Scade View** with the same interface as the operator it replaces. The interface elements match by name and type the interface elements of the original operator.
- 4 Design the abstraction operator behavior using flows and/or assertions.

It is also possible to define abstraction operators manually by declaring explicitly an operator as an abstraction and by designing the behavior of the abstraction operator using flows and/or assertions. You declare an abstraction operator from its properties.

Note that the abstraction operator need to have the same interface (same number of inputs/outputs and parameters) and the same Node/Function Declaration property as the abstracted operator it replaces.

The type of an abstraction operator need not be the same as the type of the abstracted operator. Consider the following imported function defined with float32 type and a function declared as abstraction operator using generic types:

```
function imported sin32 (x: float32) returns (y: float32);
```

```
function
  #pragma dv abstract sin32 #end
  trigo_spec (x: 't) returns (y: 't) where 't float
let
  assume A: y >= -1.0 and y <= 1.0;
tel
```

For details about type equivalence and sub-typing see [“Principles”](#) on page 153 in *Scade Language Reference Manual*.

To declare Abstraction operator from properties

- 1 Select the operator to be declared as Abstraction operator in **Scade View** as display its **Design Verifier** properties

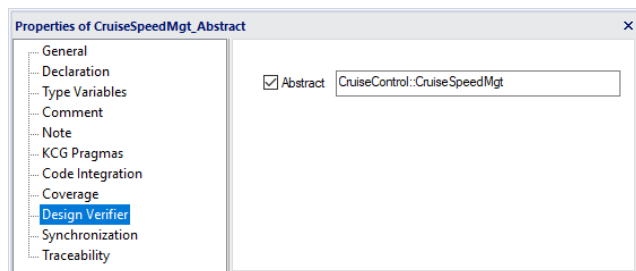


Figure 16.17: Declaring Abstraction operator properties

- 2 Check **Abstract** and specify the path of the operator to be abstracted in text field.

Note

One Abstraction operator can abstract several other operators. In that case, a list must be specified where each path is separated by a coma.

Organizing Design Verifier Analysis Objects

While analyzing models all along the software development process, you may end up with a lot of verification activities represented by numerous analysis objects like proof objectives and strategies. In the verification project, it is a good practice to organize data into a meaningful hierarchy of folders and sub-folders to classify proof objectives and strategies (e.g., tasks to validate a given part of the design, tasks for safety requirements, tasks for integration purpose, tasks for degraded mode analysis). Design Verifier allows for creating sub-folders to classify proof objectives in folders and sub-folders with regard to identified verification activities.

- [“Organizing Proof Objectives in Folders”](#)
- [“Associating Strategies with Proof Objectives”](#)
- [“Editing Properties of Task Folders”](#)
- [“Documenting Verification Projects”](#)
- [“Organizing Strategies in Folders”](#)

Organizing Proof Objectives in Folders

Folders in Design Verifier View allow to classify proof objectives by verification tasks. You just need to create task folders. You may launch analysis from selected folders (see [“Launching Property Analyses”](#) on page 756) to analyze all the proof objectives assembled in the selected folder.

To create a new sub-folder

- 1 In **Design Verifier View**, right-click a default folder (User Proof Objectives or Predefined Proof Objectives) or any existing sub-folder and select *Insert > Task Folder*.
- 2 Rename the newly added sub-folder as appropriate.

- 3 Drag and drop any proof objective, a multi-selection of proof objectives, or any sub-folder on the new sub-folder to organize the verification tasks.

Tip

You can also apply the same strategy to the content of individual sub-folders (see [“Associating Strategies with Proof Objectives”](#) on page 754).

Editing Properties of Task Folders

You can edit the properties of task folders to change their names, but also to associate the same strategy with all proof objectives contained in the selected task folder.

To edit properties of task folders

- 1 In **Design Verifier View**, right-click any created task folder and select *Properties*.

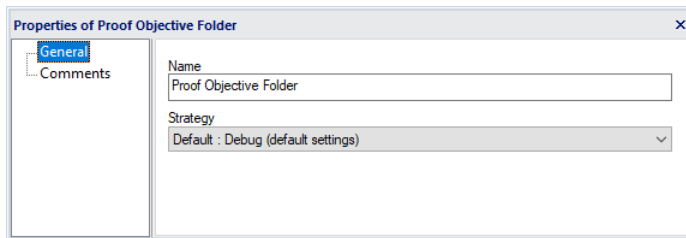


Figure 16.18: Editing properties of a task folder

- 2 In **General** properties, edit the folder name in **Name**.
- 3 From **Strategy** drop-down list, select the strategy to apply to all proof objectives in the folder (see [“Associating Strategies with Proof Objectives”](#) on page 754).
- 4 In **Comments**, comment the folder (see [“Adding and Editing Comments”](#) on page 483).

Organizing Strategies in Folders

If the system model you analyze requires a lot of properties, you may want to classify in various sub-folders all the verification strategies you configure.

To create a strategy sub-folder

- 1 In **Design Verifier View**, right-click the **Strategies** folder and select *Insert > Strategy Folder* to add a new folder.
- 2 Rename the new folder as needed. Drag and drop strategies and strategy folders in this new folder to organize the strategies. You can also edit the name of the strategy folder from its properties and comment strategy folders from **Notes** properties.

Associating Strategies with Proof Objectives

After creating and configuring strategies, you must decide for which proof objective you want to use them. You associate strategies with proof objectives to achieve various results during analysis (see [“Specifying User-Defined Strategies”](#) on page 746).

To associate a strategy with a proof objective

- 1 In **Design Verifier View**, select the strategy to associate from the appropriate folder.
- 2 Drag and drop the selected strategy on the intended proof objective. The selection updates automatically the properties of the proof objective.
- 3 Edit the **Properties** of proof objectives as in [“Specifying User Proof Objectives”](#) on page 740 or in [“Setting Predefined Proof Objectives Properties”](#) on page 744.

The strategy associated with a proof objective is used when analyzing.

To associate a strategy with a task folder

- 1 In **Design Verifier View**, select the strategy from the appropriate folder.
- 2 Drag and drop the selected strategy on the intended task folder.
- 3 Edit the **Properties** of the folder and associate a strategy with all proof objectives in the selected folder (see [“Editing Properties of Task Folders”](#) on page 753).

When associating a strategy with a task folder, it is necessary to consider the following:

- If a proof objective in a task folder has already an associated strategy, Design Verifier uses the associated strategy during analysis. Associating another strategy to this folder does not modify the strategy set for the proof objective.
- If a proof objective has the default strategy associated, Design Verifier uses the strategy associated with the task folder that contains the proof objective.

Documenting Verification Projects

Each item in the verification project (proof objectives, strategies, and task folders) has an associated field where you can enter any piece of comment to document the system.

To write comments on verification project items

- 1 Right-click any item created in **Design Verifier View** and select *Properties*.
- 2 Display **Comments**, enter the comment in the text area, and validate.

Launching Property Analyses

Provided proof objectives are properly configured, you can launch formal property analysis on a single or multiple proof objectives. Analyzing several proof objectives at once may improve the overall analysis time compared to analyzing proof objectives one by one. A progress window shows the time spent for each phase of the analysis, as well as the result. You can abort analysis from this window or close it at the end to proceed.

To launch analysis on proof objective(s)

- 1 In **Design Verifier View**, select a proof objective from a task folder or an entire task folder. Select several proof objectives in different locations holding the CTRL-key.
 - Click **Analyze**  in **Project** toolbar to launch analysis on selected proof objective(s). If several proof objectives sharing the same root node and same strategy options are selected, all are analyzed together. Alternatively, right-click a proof objective, a task folder, or a multi-selection of proof objectives and select *Analyze*.
 - To force analysis one after another on a selection of several proof objectives, right-click a proof objective, a task folder, or a multi-selection of proof objectives and select *Analyze One By One* to launch analysis.

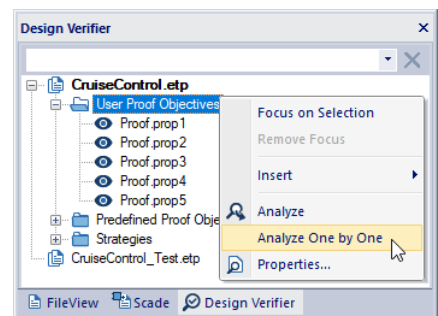
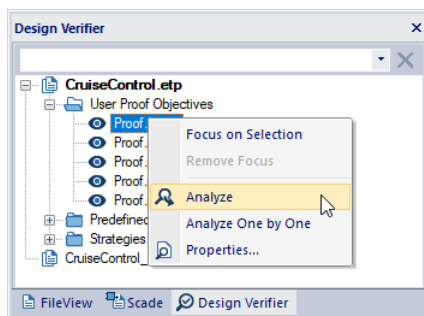


Figure 16.19: Analyzing a specific proof objective or a set of proof objectives

- 2 Design Verifier displays the status of the verification process in the **Analysis** window. The status information provides progress data related to the applied strategy and the examined proof objectives.
- 3 Click **Close** to focus on results in the generated report (see [“Interpreting Property Analyses Results”](#) on page 758). After analysis, the log file `cmd_dv . Log` is generated in the project directory. It contains all command lines used by Design Verifier. Running another analysis overwrites the display of the generated report.

If the analysis is stopped before the strategy terminates, a report containing a status for all proof objectives involved in the run is displayed. The status indicates for each proof objective whether it was proved (valid) or disproved (falsifiable), and shows an indeterminate status for the other cases.

Interpreting Property Analyses Results

At analysis completion, Design Verifier displays a verification result status in the **Browse** output tab and opens a report listing the analyzed proof objective(s), the strategy, overall analysis data, and the verification result. In case of falsifiable property or detected exception, an hyperlink provides access to a counter-example.

- [“Reading Basic Result Messages”](#)
- [“Launching Verification Scenarios from Reports”](#)
- [“Displaying Analysis Information in Reports”](#)
- [“Displaying Counter-Examples in Scenario Viewer”](#)

Reading Basic Result Messages

The result of the verification session is indicated by a status message. The meaning of the verification status messages are described in [Table 16.2](#) below.

To read the status messages of analysis

- 1 Select **View > Output** to display the Output Window.
- 2 Display the **Browse** output tab to see the result for the analyzed proof objective.



Script Item	Task	Result
✖ Proof.prop1	User	Falsifiable
✖ Proof.prop2	User	Falsifiable
✖ Proof.prop3	User	Falsifiable
✖ Proof.prop4	User	Falsifiable
✔ Proof.prop5	User	Valid

Messages Dump Build Simulator MATLAB Browse

Figure 16.20: Status message of a proof objective analysis

3 Check the name of the proof objectives with its status message.

Table 16.2: Meaning of proof results

Proof result	Meaning
Valid	The verified property is always true.
Falsifiable	Design Verifier detects a valuation of system inputs such that the output of the observer operator evaluates to false. Such input valuation is called a counter-example. To access the counter-example of a falsifiable property, see “Displaying Counter-Examples in Scenario Viewer” on page 763.
Indeterminate	Analysis failed in finding a proof or a counter-example. This is generally due to the following: (1) use of debug strategy; (2) the property is not strong enough to be proved by induction with the chosen parameters.
Interrupted	Analysis is manually aborted in the status window or Design Verifier finds no counter-example before strategy time-out (“Specifying User-Defined Strategies” on page 746).
Raised an Error	The cause of error displays in the message field of the report. Possible errors are: use of unsupported Scade language features, bad syntax or semantics in the design, or an internal error of the proof engine.
Consistent	Analysis finds that constraints are always satisfiable.
Inconsistent	Analysis can find a scenario leading one or more constraints to be unsatisfiable.

Displaying Analysis Information in Reports

Design Verifier generates HTML reports to summarize information about the verification activities, the proving process, the assumptions and abstraction operators used for analysis, the analysis results, and a simulation input file when needed. You can use hyperlinks to display information about the analysis configuration.

To display information about property analysis

- 1 In reports, click on the active links to navigate and display information.
- 2 Click on links corresponding to the analyzed property name to highlight the property in its **Diagram View**.
- 3 Click on the link corresponding to the strategy name and display its definition information summarized in the report.
- 4 Click on **Location** link to locate proof objective falsifications (Division-by-zero, Overflow, Dynamic Access) in the model. Right-click in the report and select **Back** or **Forward** contextual menus to switch between page displays in the report.

You can also access any verification report from previous analysis. Design Verifier archives all reports under the same directory of SCADE Suite projects.

To open a report from previous analysis

- 1 Select *File > Open* to browse for files in the root directory of the project. Go to `<Project_dir>/reports/<time_stamp>`, where `<time_stamp>` is a directory named after analysis date and time.
- 2 Set all file type and open a report folder.
- 3 Select `index.html` and click **Open** to display the report in SCADE Suite.

Launching Verification Scenarios from Reports

After running property analysis, you can launch the verification scenario in Simulation from the generated report. A scenario file is created for each analyzed operator when Design Verifier found at least one counter-example (*i.e.*, the property is falsifiable). Generated scenarios contain only input values relevant to property falsification.

To launch a scenario in Simulator

- 1 In **Code Generator** Settings, set a simulation configuration and select the root operator (the same as the one for which a scenario was generated).
- 2 Under **Tasks** in the report, go to **Proof Objectives** information and locate the Scenario line. Click **Load Scenario**. A configuration set for Simulation launch must be active in **Code Generator** toolbar.



Figure 16.21: Loading a scenario from property analysis reports

- 3 Click **Yes** if asked to build a simulation binary file. The scenario is launched in simulation. Read about simulation in [Part 2](#) of *SCADE Suite User Manual*.

If the same root operator is set in code generation and simulation settings, you can launch in simulation any scenario from previous analyses. Design Verifier archives scenarios under the same directory of SCADE Suite projects.

Note

In some particular cases, the counter-example of a falsifiable property may not be reproduced in simulation. This is due to differences between values provided in the model and values used by Design Verifier during proof analysis. This can happen when the analyzed model contains imported constants or imported operators, or uses abstracted operators. In these situations, the values used during proof analysis are provided as comments for information into the generated scenario files.

To launch an archived scenario from previous analysis

- 1 In **Scade View**, right-click an observer and select *Run Node* <PropertyName>. If no menu is available, enable a Simulation configuration in **Code Generator** toolbar and set *Root operator* on <selected operator> in **Code Generator** settings.

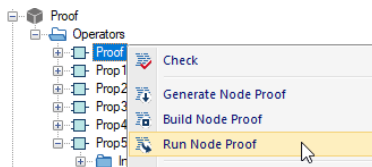


Figure 16.22: Launching simulation of observer operator

- 2 Display **Simulator Extension** toolbar from *View > Toolbars*. and click **Load scenario**  in **Simulator Extension** toolbar to browse for files.
- 3 Navigate to the root directory of the project and open it.
- 4 Follow <Project_dir>/reports/<time_stamp>/scenarios.
- 5 Set file type to **Scenario Files** and select an SSS input file.
- 6 Click **Open** to launch the selected scenario in simulation. Read more about simulation in Chapter 12 about [“Running Simulation in SCADE Suite”](#).

Displaying Counter-Examples in Scenario Viewer

Whenever a property is falsifiable, you can display counter-examples found by Design Verifier in **Scenario Viewer**. The results display as follows:

- If a variable is meaningless for a property, namely the value of this variable does not affect the output of the observer operator, it is not displayed.
- If at one particular discrete cycle, the value of a variable is meaningless, it is displayed with a star symbol (*).
- For debugging, you can set the level of information about inputs, outputs, and local variables (see [“Specifying User-Defined Strategies”](#) on page 746).

To display counter-examples

- 1 In **Browse** tab of the **Output Window**, double-click the proof objective line to open a **Scenario Viewer** named after the proof objective. Alternatively, open files describing counter-examples from HTML reports: click the link to the simulation input file (.sss).
- 2 Expand the inputs, outputs, and other variables or operators to display the values generated for the counter-example.

Scenario Viewer shows the value for each variable (inputs/outputs) of the top-level operator used in the counter-example. If the counter-example extends over more than one discrete execution cycle, the values display for every discrete cycles in separate columns.

Proof.prop1 x			
Var		1	2
Proof::Proof			
Inputs			
On		true	false
Off		false	false
Resume		false	true
Set		false	false
QuickAccel		false	false
QuickDecel		false	false
Accelerator		0	5.20417043e-18
Brake		3.015625	12
VehiculeSpeed		32	128
Outputs			
prop1		true	false
Locals			

Figure 16.23: Scenario Viewer displaying a single counter-example

If Design Verifier finds several counter-examples, each scenario is represented as a specific top-level operator in **Scenario Viewer**.

Defining Custom Strategies

SCADE Suite Design Verifier is built on top of the Prover PSL® SAT-based proof engine by Prover® Technology. Design Verifier is able to run proof analysis using a set of predefined and configurable strategies but also using custom strategies that require a thorough understanding of the Prover PSL strategy definition syntax as explained hereafter.

- [“Configuring Prover PSL SAT-Solvers”](#)
- [“Composing Strategies and Supported Use Cases”](#)
- [“Defining Analysis Strategies”](#)
- [“Description of Strategies Algorithm”](#)

PROVER PSL PROOF ENGINE AND MODEL CHECKING PRINCIPLES

The Prover PSL proof engine implements several analysis strategies which all rely on SAT-solving that aims at computing a solution for the SATisfiability of Boolean expressions. If no solution exists, the expression is said to be *Unsatisfiable*; it is said *Satisfiable* otherwise. When a SAT-solving procedure is used for model checking, the model and the negated property define the Boolean problem that is passed to the SAT-solver.

For property analysis, Prover PSL returns:

- *Valid*: the property is true for all possible values.
- *Falsifiable*: there exists a valuation that falsifies the property.
- *Indeterminate*: the solver exceeded computational resources (memory, time, analysis depth) without determining if the property is Valid or Falsifiable.

For model consistency analysis, Prover PSL returns:

- *Consistent*: the constraints are always satisfiable provided that the model and its constraints contain no memory.
- *Inconsistent*: a scenario leads one or more constraints to be unsatisfiable.
- *Indeterminate*: same as above (for property analysis).

The proof engine relies on sequential logic, which allows checking the invariance of properties. When dealing with sequence-based models, a property is said to be invariant when there exists no input sequence leading to property falsification.

From a mathematical point of view, a 1st order sequence can be described by its initial value and a function F defining the next value of the sequence:

$$A_0 = V$$

$$A_{n+1} = F(A_n)$$

Proving that a property P is an invariant boils down to proving the following: $\forall n \in \mathbb{N} \cdot P(A_n)$. By using the induction principle on natural numbers, proving the property decomposes in two cases (the base and the step) to prove:

$$\text{Base} \quad P(A_0)$$

$$\text{Step} \quad P(A_n) \Rightarrow P(A_{n+1})$$

To prove that a property is an invariant, Prover PSL implements several procedures on top of the SAT-solving procedure, the induction principle being one of them.

Configuring Prover PSL SAT-Solvers

Prover PSL offers the ability to run analysis on several threads, provides multiple SAT-solvers and implements several analysis strategies that can be used to strengthen the problem (refer to [“Flipped Bits”](#) on page 782 and [“Counter-Based Models”](#) on page 783 for illustrations), prove, or disprove a model. The proof engine includes a rich command line mechanism allowing users to take advantage of all these features and let one define more complex analysis strategies.

Preprocessing aims at reducing the size of a problem with regard to the number of variables or the number of clauses. Depending on the solver, analysis may be incomplete¹, leading to indeterminate result if used to prove or disprove a property. The following table presents the characteristics of the available solvers:

Solver	Complete	Usage
1		Pre-processing
2	X	Proof
3	X	Proof
4		Pre-processing
5		Pre-processing
6	X	Proof
7	X	Proof
8	X	Proof
9		Pre-processing

In case no solver is specified, the default solver (solver 1 + solver 3) is used, where solver 1 is used as preprocessor and solver 3 as the main solver.

Example 1:The proof engine uses solver 1 as preprocessor and then solver 3 for analysis (default solver).

```
-solver 1+3
```

Example 2:The proof engine uses solver 5 to analyze the model with <strat1>, then it uses solver 3 to analyze it with <strat2>.

```
-solver 5 <strat1> -solver 3 <strat2>
```

1. If run with enough computational resources (time/memory), a solving procedure is complete when guaranteed to return a satisfiable or unsatisfiable status.

-THREADS/-TURBOTHREADS

This option allows running the proof engine, or a single strategy, in a multithreaded way. If the value passed to the option equals or exceeds the number of cores of the computer, the proof engine uses them all.

- Multithreading is disabled by default and applies only to the currently used solver. Therefore, if users specify different solvers in the command line, multithreading must be re-enabled where needed.
- Multithreading can be used with all the solvers without any restriction.
- Though both options allow to take benefit from computational resources, `-threads` and `-turboThreads`¹ behave differently. This second option is not deterministic (the way the values of a counter-example are chosen is not deterministic², but the length of the scenario is guaranteed to be the lowest one). As a consequence, when using `-turboThreads` to find counter-examples (e.g., coverability analysis), two successive runs may return different, yet correct, counter-examples.

CONJ

This option is applicable only to *Bounded Model Checking* and *Induction* strategies. By default, proof objectives are analyzed one after another. When using the *conj* option, all proof objectives are analyzed altogether. This option is useful when proof objectives are interdependent: part of proof analysis is shared between proof objectives resulting in reduced analysis time. On the contrary, if the proof objectives are not interdependent at all, using this option results in a more complex analysis and may increase analysis time.

-
1. The `-turboThreads` option is faster than `-threads` option. For Coverability Analysis, it is recommended to use `-threads` since it produces the same scenarios for each run and therefore ensures traceability to LLRs.
 2. Determinism covers both temporal and spatial dimensions (data values). Here, only the spatial dimension is impacted, the temporal dimension is preserved.

Defining Analysis Strategies

The following strategies can be defined in the command line to configure model check analysis:

-bmc	-ind
-bcc	-ipol
-eqLemmas	

-BMC

BMC stands for *Bounded Model Checking*. It allows checking if a property holds between a start and a stop cycle. In case the model and its property contain no memory, and the property holds, BMC proves that the property is invariant.

All available options are described in [“List of Strategy Options”](#) on page 779.

By default, Prover PSL BMC option values are:

• Start: 0	• Inc: 1
• Stop: -1 (never stop)	• Timeout: -1 (no timeout)

The returned status can be:

- **Falsifiable** if a counter-example can be exhibited
- **Valid** only if the model and its property contain no memory and no counter-example can be exhibited
- **Indeterminate** if the model and its property contain memories and the property is true from first to stop cycle, or if timeout is reached

For option definition usage, see [“Bounded Model Checking”](#) on page 779.

Example 1: The model and its properties are checked using BMC with all arguments set to their default value. The default solver is used in single-threaded mode.

-bmc

Example 2:The model and its properties are checked using BMC from cycle 2 to 20. The default solver is used in single-threaded mode.

```
-bmc start 2 stop 20
```

Example 3:The model and its properties are checked using BMC from cycle 0 to 15. The default solver is used in single-threaded mode. Note that the `conj` argument allows considering all properties as a conjunction.

```
-bmc conj stop 15
```

Example 4:This run is set up to use solver 3 in multi-threaded mode (4 cores) instead of the default solver. The model and its properties are checked using BMC from cycle 2 to 20.

```
-solver 3 -turboThreads 4 -bmc start 2 stop 20
```

-BCC

BCC stands for *Bounded Consistency Check*. It ensures the assumptions expressed on a model admit a solution. Users must be aware that, from a model where assumptions cannot be satisfied together, a proof can be obtained for any property.

All available options are described in [“List of Strategy Options”](#) on page 779.

By default, Prover PSL BCC option values are:

- Start: 0
- Inc: 1
- Stop: -1 (never stop)
- Timeout: -1 (no timeout)

The returned status can be:

- **Inconsistent** if a counter-example can be exhibited
- **Consistent** only if assumptions admit a solution, and the model, with the constraints, contain no memory
- **Indeterminate** otherwise

For option definition usage, see [“Bounded Consistency Checking”](#) on page 780.

Example 1:The model and its properties are analyzed using Bounded Consistency Check with all arguments set to their default value. The default solver is used in single-threaded mode.

```
-bcc
```

Example 2:The model and its properties are analyzed using Bounded Consistency Check from cycle 2 to 20. The default solver is used in single-threaded mode.

```
-bcc start 2 stop 20
```

Example 3:This run is set up to use solver 3 in multi-threaded mode (4 cores) instead of the default solver. The model and its properties are analyzed using Bounded Consistency Check from cycle 2 to 20.

```
-solver 3 -turboThreads 4 -bcc start 2 stop 20
```

-EQLEMMAS

Equivalence Reasoning (`-eqLemmas`) is a strategy that exhibits, proves, and reuses bit-level equivalence properties. It can rarely be used alone to prove properties, but it can help to strengthen the problem and turn a property from a non-inductive one to an inductive one. This strategy can be valuable if induction (step case check) does not succeed to prove/disprove a property (indeterminate status). The way the analysis is configured has consequences on the strategy performance itself, but also on the overall run:

- **Solver**¹: depending on the solver characteristics (*i.e.*, complete or not), the number of proved lemmas and the analysis time may differ. A complete solver proves more equivalence lemmas, which may in turn lead to performance slow-down.

1. Solver 5 is a good alternative to the default solver: it is well-suited for equivalence reasoning and provides good performances (though not complete).

- **Proof strategy:** equivalence properties are proved by induction. There are two different ways to configure the strategy:

-eqLemmas faster but proving fewer equivalence properties
 -eqLemmas ind full slower but proving more equivalence properties

All available options are described in [“List of Strategy Options”](#) on page 779.

-IND

The *induction* strategy is the most common model-checking strategy for Scade models. It implements the general induction principle.

Considering a property P , where P_k defines the value of P at cycle k , induction-based model checking amounts to checking the following holds:

Base case check	$P_0 \wedge P_1 \wedge \dots \wedge P_k$
Step case check (<i>i.e.</i> , generalization)	$\forall n \cdot P_n \wedge P_{(n+1)} \wedge \dots \wedge P_{(n+k)} \Rightarrow P_{(n+k+1)}$

In case the base check fails, the property is said to be falsifiable. In case the step case fails, nothing can be said about the property leading the property to be indeterminate; the property is said not to be k -inductive. Illustrated examples of properties that are not k -inductive and how to strengthen the problem to get a proof are provided by [“Flipped Bits”](#) on page 782 and [“Counter-Based Models”](#) on page 783).

Note

Equivalence reasoning may be used to strengthen the problem when properties are not inductive. In addition, the `conj` option can be used to get similar effect.

All available options are described in [“List of Strategy Options”](#) on page 779.

By default, Prover PSL induction option values are:

- Start: 0
- Stop: -1 (k is incremented until a proof or counter-example can be found)
- Inc: 1
- Timeout: -1 (no timeout)

The returned status can be:

- **Falsifiable** if a counter-example for base check can be exhibited
- **Valid** if no counter-example for base and step checks can be exhibited
- **Indeterminate** in case the stop cycle is reached and a counter-example for step check is found

For option definition usage, see [“K-Induction”](#) on page 780.

Example 1: Performing a 0-induction proof using the current solver in single-threaded mode

```
-ind stop 0
```

Proving: P_0 then $\forall n \cdot P_n \Rightarrow P_{n+1}$.

Example 2: Performing a 2-induction proof using solver 1+3 (solver 1 used as a preprocessor) in a multi-threaded mode.

```
-solver 1+3 -turboThreads 4 -ind stop 2
```

Proving at depth 0 (default value for start depth is 0): P_0 then

$\forall n \cdot P_n \Rightarrow P_{n+1}$

Proving at depth 1: P_1 then $\forall n \cdot P_n \wedge P_{n+1} \Rightarrow P_{n+1+1}$

Proving at depth 2 (stop depth is reached): P_2 then

$\forall n \cdot P_n \wedge P_{n+1} \wedge P_{n+2} \Rightarrow P_{n+2+1}$

Example 3: Possibility to control when to start proving the step case

```
-ind start 1 stop 2
```

Proving at depth 0: P_0

Proving at depth 1 (start depth is reached): P_1 then

$\forall n \cdot P_n \wedge P_{n+1} \Rightarrow P_{n+1+1}$

-IPOL

Interpolation applied to model checking is used to check if an unexpected state (negation of a property) is reachable from the initial state of the model or not. Interpolation relies on computing, at each cycle, an over-approximation of the reachable states. Though much slower and more consuming than induction, interpolation may be useful when induction fails and no strengthening property can be exhibited. A brief introduction to this strategy can be found from [this paper](#).

All available options are described in “[List of Strategy Options](#)” on page 779.

Example 1:Running the interpolation strategy using the current solver

```
-ipol
```

Composing Strategies and Supported Use Cases

Prover PSL allows combining analysis strategies. The proof engine is initialized with a default solver that can be changed according to user needs. A custom strategy must be thought as a sequence of the following actions:

- Optionally change the solver to be used
- Optionally change the solver running mode
- Define the analysis strategy

In the following examples, pseudo-code algorithms define how strategies are interpreted and handled by the proof engine.

Example 1:Design Verifier **Prove** strategy - the analysis run performs two passes of equivalence reasoning using the default solver. It then switches to a 0-induction proof followed by an interpolation proof.

```
-eqLemmas timeout 30 -eqLemmas ind full timeout 30 -ind conj stop 0 -ipol
```

Example 2: Design Verifier **Induction** strategy

```
-eqLemmas ind full timeout 30 -ind <%start> <%inc> <%stop> <%timeout>
```

Example 3: Design Verifier **Debug** strategy - this strategy uses the directive “-solver current {inc=true}” that turns the currently used solver into an incremental one. An incremental solver usually allows to run faster and longer (in terms of cycles).

```
-solver current {inc=true} -bmc <%start> <%stop> <%timeout>
```

Example 4: The default strategy used for Model Coverability analysis (based on Design Verifier proof engine) is a strategy mixing Bounded Model Checking with induction and interpolation. Each step of this strategy aims at pruning as many points as possible by:

- finding the points coverable by short sequences of inputs
- finding non-coverable points using an efficient strategy (induction with stop parameter set to 0)
- finding the points coverable by longer sequences (up to a maximum depth)
- finding non-coverable points using a more powerful, yet more costly, strategy (*i.e.*, interpolation)

Each step is represented by these successive command lines:

```
-solver current {inc=true} -bmc conj start 0 stop 5  
-solver 5 -eqLemmas ind full timeout 75  
-solver 1+3 -ind conj stop 1  
-solver current {inc=true} -bmc conj
```

The ability to compose strategies by modifying the solvers, their running mode and the strategy definition allow to support various use cases as described below.

- [“Analyzing Sets of Properties”](#)
- [“Dealing with Large Numbers of Proof Objectives”](#)
- [“Equivalence Reasoning Strength”](#)
- [“Dealing with Array-Based Models”](#)

ANALYZING SETS OF PROPERTIES

The **conj** flag is particularly useful when proof objectives are linked by a dependency relation which is the case for model coverability or predefined proof objectives analysis.

EQUIVALENCE REASONING STRENGTH

In the predefined strategies, the equivalence reasoning step makes use of a complete solver which may be time-consuming. To improve performance, it is possible to switch to an incomplete solver. Solver 5 is a good trade-off: it is well-suited for equivalence reasoning and offers good performances.

DEALING WITH LARGE NUMBERS OF PROOF OBJECTIVES

When a model contains a large number of Proof Objectives, as in predefined Overflow check, it may be useful to quickly separate the ones that are falsifiable or trivially valid from the most complex ones.

For this purpose, users can run:

- a short BMC step, a short induction step, or both
- followed by more complex and time-consuming analysis strategies

The following custom command aims at implementing this behavior:

```
-bmc conj stop 2 -ind conj stop 2  
-solver 5 -eqLemmas ind full -solver 1+3 -ind conj start 2 stop
```

Note that each time a property is proved, it amounts to adding it to the knowledge base. This can help to prove other properties (problem strengthening). The way the above strategy is designed allows getting rid of the properties that are falsifiable in at most 2 cycles and to quickly identify the k-inductive properties (where k ranges from 0 to 2).

DEALING WITH ARRAY-BASED MODELS

When analyzing huge problems, in terms of constraints and variables, using Prover PSL in multi-threaded mode can significantly speed up analysis.

Example: Use the following command line derived from Design Verifier predefined Prove strategy

```
-solver 5 -eqLemmas timeout 30 -eqLemmas ind full timeout 30 -solver 1+3 -ind conj stop 0 -ipol
```

In case `-turboThreads` is used at the beginning of the command line, the option has no effect since it applies to the currently used solver. To use the above strategy in multi-threading mode, the option must be set directly after the solver is updated.

Users can choose to enable it for:

- equivalence reasoning analysis which uses solver 5
- and/or induction AND interpolation which both use solver 1+3

To restrict multi-threading to induction and interpolation, use the following command line:

```
-solver 5 -eqLemmas timeout 30 -eqLemmas ind full timeout 30 -solver 1+3 -turboThreads 4 -ind conj stop 0 -ipol
```

Description of Strategies Algorithm

LIST OF STRATEGY OPTIONS

The following options may be used by Prover PSL strategies:

Options	BMC	BCC	Induction	Interpolation	Equivalence Reasoning
start : controls the cycle from where to start running step cases. If option is unset, the step check is performed for each increment	x	x	x	x	x
inc : controls the increment	x	x	x	x	x
stop : controls the cycle where to stop base checks	x	x	x	x	x
timeout : controls the maximum amount of time (in seconds) that can be used by the strategy	x	x	x	x	x
ind full : allows to prove more equivalence properties					x

Each strategy uses its own default option values as described for each.

BOUNDED MODEL CHECKING

```
BMC(Model, Assumptions, Properties, start, stop, inc)
begin
  current_cycle := 0;
  status := indeterminate;
  while (current_cycle <= stop or stop = -1) and status = indeterminate
  do
    if (current_cycle >= start) then status := check(Model, Assumptions, Properties);
    Current_cycle := current_cycle + inc;
    Model, Assumption, Properties := unfold(Model, Assumptions, Properties, inc);
  done;
  return status;
end
```

BOUNDED CONSISTENCY CHECKING

```
BCC(Model, Assumptions)
begin
  current_cycle := 0;
  status := indeterminate;
  while (current_cycle <= stop or stop = -1) and status = indeterminate
  do
    if (current_cycle >= start) then status := assumptions_check(Model, Assumptions);
    Current_cycle := current_cycle + inc;
    Model, Assumptions := unfold(Model, Assumptions, inc);
  done;
  return status;
end
```

K-INDUCTION

```
K-ind(Model, Assumptions, Properties)
begin
  base_status := indeterminate;
  step_status := indeterminate;
  status := indeterminate;
  k := 0;
  while (k <= stop or stop = -1) and status = indeterminate
  loop
    base_status := BaseCheck(k, Model, Assumptions, Properties);
    if k >= start then step_status := StepCheck(k, Model, Assumptions, Properties);
    status := (case base_status, step_status with
      | Falsifiable, _ => Falsifiable -- counter-example
      | _, Falsifiable => Indeterminate -- step counter-example
      | _, _           => Valid);
    k := k + 1;
  end loop;
  return status;
end
```

STRATEGIES EXECUTION

```
executeStrategies(strategies, model, properties)
begin
  i := 0;
  running_mode := Single_threaded;
  solver := default_solver;
  remaining_pos := |properties|;
  while i < |strategies| and remaining_pos > 0
  begin
    solver, running_mode, remaining_pos :=
      execute(strategies[i], solver, running_mode, model, properties, remaining_pos);
    i := i + 1;
  end
end
```

Illustrated Examples

- [“Flipped Bits”](#)
- [“Counter-Based Models”](#)

FLIPPED BITS

The following example defines two Boolean flows that are flipped at each cycle.

```
node bits_flip (x: bool) returns (a, b, lemma, po: bool)
let
  a = x -> not pre a;
  b = not x -> not pre b;
  lemma = a = not b; -- this equivalence property can be inferred by -eqLemmas
  po = a or b;
tel
```

The following table details the status of each property depending on the chosen strategy. It is important to note that, in such a case, the Lemma property can be inferred by -eqLemmas:

Proof Objectives	Strategy			
	-ind stop 0	-ind conj stop 0	-eqLemmas -ind stop 0	-ind stop 1
po	Undecided	Undecided	Valid ^a	Valid ^b
lemma, po	- Valid if Lemma proved first - Undecided ^c otherwise	Valid ^d	Valid	Valid

- a. -eqLemmas allows to infer the same equality as the one defined by Lemma.
- b. po is said to be 1-inductive.
- c. The order of proof objectives plays a role in analysis: properties proved first may strengthen the model.
- d. No matter their order, when proved as a single conjunction, properties may strengthen each other. In the present case, Lemma allows to lower the k-induction depth that is required to prove po.

COUNTER-BASED MODELS

The counters model defines two out-of-phase counters (x counts down to 0 while y counts up to Max and conversely).

The property to be proved, i.e., eq_nind, captures the following: “y=0 if and only if x = Max”.

```
const Max : uint8 = 10;
node counters() returns (eq_ind, eq_nind : bool ; x, y : uint8)
var
  b1, b2 : bool;
  a, b : bool;
let
  a = false -> if (pre a) then pre x < Max else pre x <= 0;
  x = Max   -> if a then pre x + 1 else pre x - 1;
  b = true  -> if (pre b) then pre y < Max else pre y <= 0;
  y = 0     -> if b then pre y + 1 else pre y - 1;
  b1 = x=Max;
  b2 = y=0;
  eq_ind = (x+y = Max) and b1=b2;
  eq_nind = b1=b2;
tel
```

The proof objective eq_nind cannot be proved using a simple induction. To get a result, users need to use more powerful strategies or a combination of strategies. An alternative is to exhibit a relation between both counters such as the one formalized in eq_ind: “the sum of the two counters is invariantly equal to Max”.

The following table summarizes the results obtained using different strategies:

Proof Objectives	Strategy			
	-ind stop 1	-ind	-eqLemmas -ind	-ipol strength 2
eq_nind	Undecided	Undecided	Valid (17-induction when Max =10)	Valid (depth 12)
eq_ind	Valid (1-induction)	Valid	Valid	Valid

17 /Verifying Timing and Stack Usage

This chapter introduces **Timing and Stack Analysis Tools** and explains how to use these tools to perform worst-case execution time (WCET) and stack usage analysis on SCADE Suite models.

- [“Overview of Timing and Stack Analysis Tools”](#)
- [“Fundamentals about Timing and Stack Tools”](#)
- [“Setting Timing and Stack Verification”](#)
- [“Launching Timing and Stack Analysis Tools”](#)
- [“Examining Timing and Stack Analysis Results”](#)

Note

The documentation related to aiT WCET and stack analyzers is available from Absint aiT target installation.

Overview of Timing and Stack Analysis Tools

This section is a general overview of the basic concepts and user interface terminology, and of the generic workflow that support timing and stack analysis on SCADE Suite models.

- [“Purpose of Timing and Stack Analysis Tools”](#)
- [“Exploring Timing and Stack Analysis User Interface”](#)
- [“Timing and Stack Analysis Workflow”](#)

Purpose of Timing and Stack Analysis Tools

Knowing the worst-case execution time (WCET) and the stack usage of programs is crucial for the design of hard real-time systems. Therefore, timing and stack analysis have become an acknowledged part in the design of real-time systems in the last years.

SCADE Suite integrates the following Timing and Stack Analysis Tools:

- **Timing and Stack Optimizer** to compare the WCET and stack size estimation for different versions of a SCADE Suite model to determine the most efficient design. Software designers can use this tool to improve the efficiency of their SCADE Suite models with respect to WCET and stack usage. Timing and Stack Optimizer is installed with SCADE.
- **Timing and Stack Verifiers** to compute precise WCET and stack size for a SCADE Suite model on a specific hardware target. Such analysis runs with respect to specific target processors and C compilers, and requires fine-grained customization to comply with the hardware characteristics. Software designers can use this tool to verify the final efficiency of their application for integration on the target platform. For Timing and Stack Verifier, the needed AbsInt aiT target must be installed under AbsInt license.

Timing and Stack Analysis Tools enable to:

- Rapidly determine worst-case execution times and stack usage at model level.
- Examine how different functions of the SCADE Suite model contribute to the average, maximum, and cumulated execution times or to stack usage.
- Optimize the design of the SCADE Suite model and thus improve the performance of the SCADE Suite application.
- Save considerable development time, cut development costs, and shorten time to market.

Timing and Stack Analysis Workflow

Timing and stack analysis workflow consists of analysis sessions you can iterate to optimize an intermediate design or verify the timing and stack usage achieved by the final design.

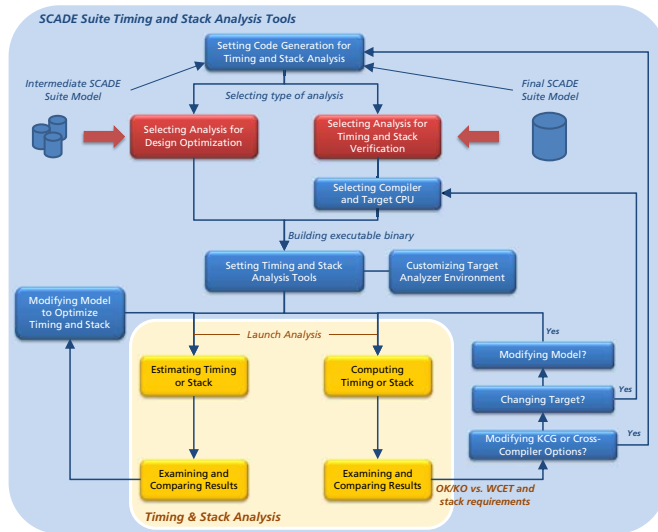


Figure 17.1: Timing and Stack analysis workflow

You run WCET and stack analysis sessions on models in phases: setting SCADE Suite KCG Code Generator for timing and stack analysis; building an executable binary of the SCADE Suite model with SCADE Suite KCG; setting Timing and Stack Analysis Tools; launching analysis for optimization or verification; and examining analysis results.

RELATED TOPICS

- [“Setting Code Generation for Timing and Stack”](#) on page 798
- [“Building Binary for Timing and Stack Analysis”](#) on page 802
- [“Setting Timing and Stack Analysis Tools”](#) on page 803
- [“Launching Timing and Stack Analysis Tools”](#) on page 809
- [“Examining Timing and Stack Analysis Results”](#) on page 811

Exploring Timing and Stack Analysis User Interface

The interface of SCADE Suite contains various components for setting timing and stack analysis, launching model analysis, analyzing and managing results, or reporting coverage.

- [“Timing and Stack Toolbar”](#)
- [“Timing and Stack Result Views”](#)
- [“Timing and Stack View”](#)

Timing and Stack Toolbar

This toolbar provides shortcuts to set and start timing or stack analysis sessions, and import external timing/stack analysis results in the active project. It can be moved in the interface by dragging the title bar to the desired location.

Timing and Stack Analysis Tools	Setting timing and stack analysis: • “Setting Timing and Stack Analysis Tools” on page 803
	Launching timing and stack analysis: • “Launching Timing and Stack Analysis Tools” on page 809
	Importing results: • “Importing External Results” on page 818

To show/hide toolbar

- Select *View > Toolbars > Timing and Stack Analysis Tools*; or right-click in the toolbar area and select the name of the toolbar to show/hide from the interface. Alternatively, hide the toolbar by right-clicking on toolbar and selecting *Hide*.

Timing and Stack View

From the **Timing and Stack View**, you can access the list of all timing and stack analysis sessions present in the project. This view is available when at least one analysis session exists in the project. If this is the case, the view is automatically displayed. Sessions are organized inside folders corresponding to the analyzed operator. In each folder, the sessions are sorted in chronological order.

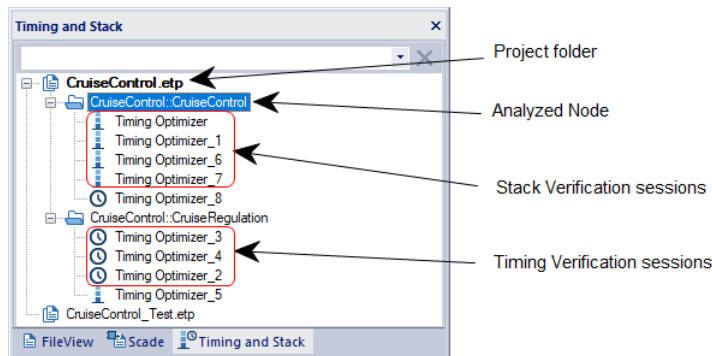


Figure 17.2: Timing and Stack View

RELATED TOPICS

- [“Displaying Analysis Session Results”](#) on page 811
- [“Updating Analysis Sessions”](#) on page 819
- [“Comparing Analysis Sessions”](#) on page 819

Timing and Stack Result Views

The **Result Views** allow you to visualize the results of any timing or stack analysis session listed in the **Timing and Stack View**. You can use these views to compare data results between two sessions of the same level of analysis. Once analysis or result comparison are completed, the results are automatically displayed in a dedicated view.

The main titles of result views refer to the timing analysis tool (and CPU if any) used and subtitles to session name. A summary of all important project information and settings about analysis sessions is provided at the beginning.

RESULT VIEWS FOR TIMING ANALYSIS

Results about timing analysis are displayed in dedicated views. These views compile WCET results from all operators called by the analyzed root operator and provide access to *detailed results*.

Timing Optimizer Analysis Result

Timing Optimizer Session

Date:

02/09/2021 14:10:18

Project:

C:\Users\Public\Documents\ANSYS Inc\v221\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp

Configuration:

Timing Optimizer

Analyzer:

Optimizer

Max unroll:

2

Default unroll:

2

KCG version:

KCG 6.6.2

Keep probes:

false

Debug:

false

Significant length:

31

Name length:

200

Root inputs and outputs as global variables:

false

Result Overview

Root init function CWCET: 642 cycles

Root reset function CWCET: 453 cycles

Root cycle function CWCET: 3095 cycles

Cycle functions

	SCADE Critical Path #	Calls	WCET (sum)	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
	CruiseControl::SaturateThrottle	1	223	223	223.00	223	223	223.00
	CruiseControl::CruiseSpeedMgt	1	461	461	461.00	461	461	461.00
	CruiseControl::CruiseRegulation	1	224	224	224.00	447	447	447.00
	CruiseControl::CruiseControl	1	1985	1985	1985.00	3095	3095	3095.00

Figure 17.3: Result view of a timing analysis session

The result data displayed about timing analysis are summarized below:

Table 17.1: result data about WCET analysis

Information	Description
Calls	Number of calling contexts in the function.
WCET (sum)	Worst-case execution time contribution of the function <i>without</i> descendants (<i>i.e.</i> sum of the function WCETs computed for each context).
WCET (max)	Worst-case execution time contribution of the function <i>without</i> descendants (<i>i.e.</i> maximum of the function WCETs computed for each context).
WCET (avg)	Worst-case execution time contribution of the function <i>without</i> descendants (<i>i.e.</i> average of the function WCETs computed for each context).
CWCET (sum)	Cumulative worst-case execution time contribution of the function <i>with</i> descendants (<i>i.e.</i> sum of the function WCETs computed for each context).
CWCET (max)	Cumulative worst-case execution time contribution of the function <i>with</i> descendants (<i>i.e.</i> maximum of the function WCETs computed for each context).
CWCET (avg)	Cumulative worst-case execution time contribution of the function <i>with</i> descendants (<i>i.e.</i> average of the function WCETs computed for each context).

Note

Results about timing analysis are expressed in machine cycles (not model cycles).

In such views, you can sort results in tables, look for detailed results at operator level, or locate an operator in the model. Read about how to examine analysis results in [“Displaying Analysis Session Results”](#) on page 811.

RESULT VIEWS FOR STACK ANALYSIS

Results about stack analysis are displayed in dedicated views. These views compile stack results from all operators called by the analyzed root operator.

Stack Optimizer Analysis Result

Stack Optimizer Session

Date:

02/09/2021 14:19:46

Project:

C:\Users\Public\Documents\ANSYS Inc\v221\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp

Configuration:

Stack Optimizer

Analyzer:

Optimizer

Max unroll:

2

Default unroll:

2

KCG version:

KCG 6.6.2

Keep probes:

false

Debug:

false

Significant length:

31

Name length:

200

Root inputs and outputs as global variables:

false

Result Overview

Root cycle function stack usage:

64 bytes

Root init function stack usage:

48 bytes

Root reset function stack usage:

56 bytes

Global (with dependencies) stack usage:

64 bytes

Cycle functions

	SCADE Critical Path #	Stack (system)	CStack (system)
	CruiseControl::SaturateThrottle	0	64
	CruiseControl::CruiseSpeedMgt	0	56
	CruiseControl::CruiseRegulation	8	64
	CruiseControl::CruiseControl	32	56

Figure 17.4: Result view of a stack analysis session

The result data displayed about stack analysis are summarized below:

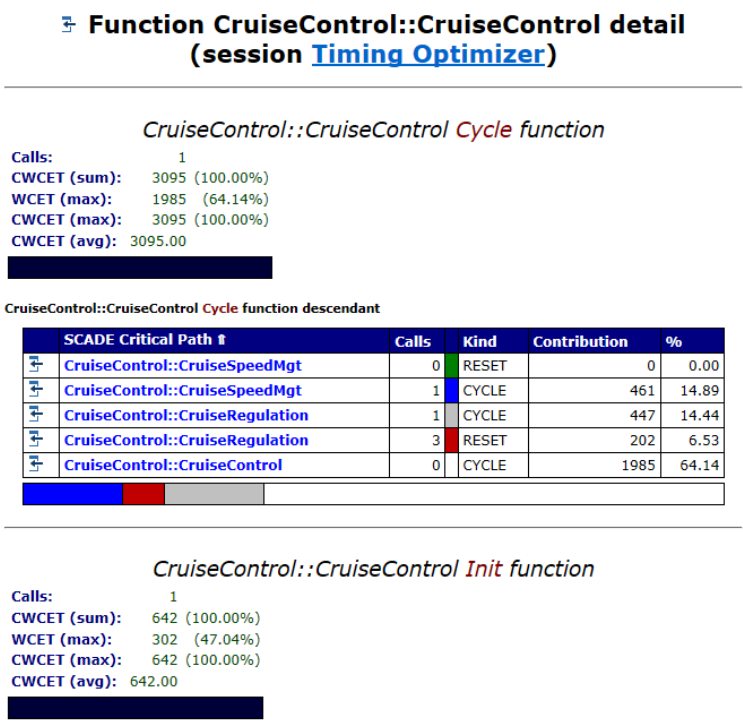
Table 17.2: Result data about stack analysis

Information	Description
Calls	Number of calling contexts in the function.
Stack	Stack usage of the function (in bytes).
CStack	Cumulative Stack usage of the function with its parents (in bytes).

In such views, you can sort results in tables or locate an operator in the model. Read about how to examine analysis results in [“Displaying Analysis Session Results”](#) on page 811.

DETAILED RESULT VIEWS

Timing analysis results about each operator called by the analyzed root operator are detailed in dedicated views in terms of cycle, init, and reset function. You access such views from the result views of timing analysis sessions only. Each detailed result view contains detailed results for the selected function that takes the worst execution time.



In such views, you can sort results in tables, look for detailed results at operator level, or locate an operator in the model. Read about how to examine timing analysis detailed results in [“Displaying Timing Analysis Detailed Results”](#) on page 814.

DIFF REPORTS OF TIMING OR STACK RESULTS

You can compare two analysis sessions and get comparison results in a **Diff Report**. A diff report is composed as result views but provides comparison data about the compared sessions. As other views, it allows locating an analyzed node in the model and sorting the display of results in tables.

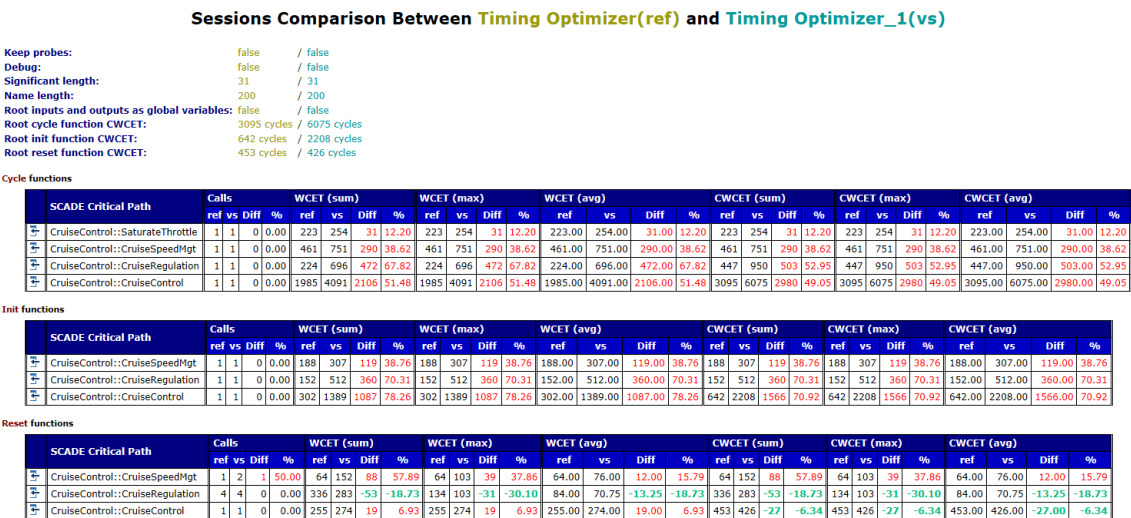


Figure 17.6: Example of a comparison report between two timing analysis sessions

RELATED TOPICS

- [“Comparing Analysis Sessions”](#) on page 819
- [“Locating Analyzed Nodes in Model”](#) on page 816
- [“Sorting Analysis Results”](#) on page 816

Fundamentals about Timing and Stack Tools

Timing and Stack Analysis Tools analyze a binary file of a SCADE Suite model. They compute the worst-case execution time and the stack usage for a specified root operator and display results. The binary file can be built from the code generator using the Timing and Stack Analysis Tools extension and the appropriate cross compiler.

Timing Analysis

Timing and Stack Optimizer and Timing and Stack Verifiers use aiT's WCET Analyzers technology to statically compute the worst-case execution time of the SCADE Suite models.

Stack Analysis

Timing and Stack Optimizer and Timing and Stack Verifiers use StackAnalyzer technology to determine the stack usage of the SCADE Suite models.

Analysis Session and Results

Each time you launch an analysis with Timing and Stack Optimizer or Timing and Stack Verifiers on a SCADE Suite model, you get an analysis session. For each session, the results are presented in HTML format.

Timing and Stack Analysis Tools and KCG

The following figure gives a general overview of how one can use SCADE Suite KCG with Timing and Stack Optimizer or Timing and Stack Verifiers to get information about WCET and stack usage of a SCADE Suite model:

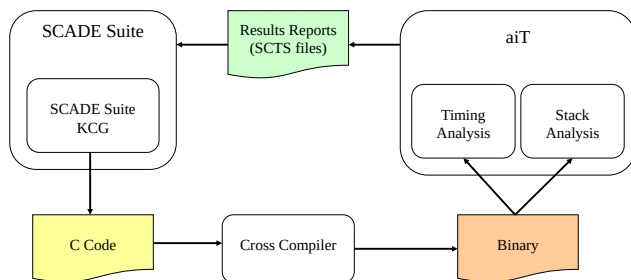


Figure 17.7: Timing and stack usage analysis overview

For each analysis session, Time and Stack Analysis Tools associate an STCS file to the active configuration. Each SCTS file contains the results of a WCET or stack analysis. Such files are named `<ConfigurationName>[_<Number>].scts`. An incremental number is added to the file name if a file with the same name is already in the folder.

Note

For later analysis session and further comparison purposes, it is recommended to store in Configuration Management the model in the state corresponding to each analysis session.

Setting Timing and Stack Verification

Before launching WCET or stack analysis, you set SCADE Suite Code Generator and build an executable binary of the model. Then, you set Timing and Stack Analysis Tools.

- [“Setting Timing and Stack Result Options”](#)
- [“Setting Code Generation for Timing and Stack”](#)
- [“Building Binary for Timing and Stack Analysis”](#)
- [“Setting Timing and Stack Analysis Tools”](#)
- [“Setting Advanced Timing and Stack Analysis”](#)

Setting Timing and Stack Result Options

These options allow to show/hide a legend and data details related to timing and stack analysis session views or diff session views. Options are effective when at least one Timing or Stack result file (STCS file) is in the project.

To select Timing and Stack result options

- 1 Select *Tools > Options* and display **Timing and Stack** to view applicable options.

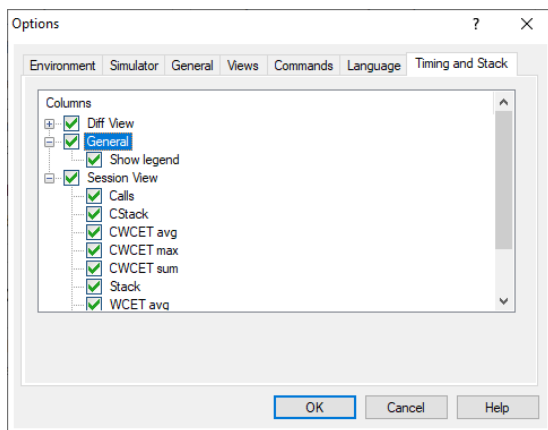


Figure 17.8: Options related to Timing and Stack result views

- 2 Check/uncheck items to show/hide various levels of information:

- Check a category of available information to check all its content.
- Hold **Ctr l**-key to set separate items one by one in multi-selection or **Shift**-key to set consecutive items in multi-selection.
- Press **Space** bar to check/uncheck the list highlighted by multi-selection.

Setting Code Generation for Timing and Stack

You must set up SCADE Suite Code Generator to generate code and build an executable binary that supports WCET and stack analysis.

To set SCADE Suite Code Generator for WCET and stack analysis

- 1 Click **Settings**  in **Code Generator** toolbar to display available settings.

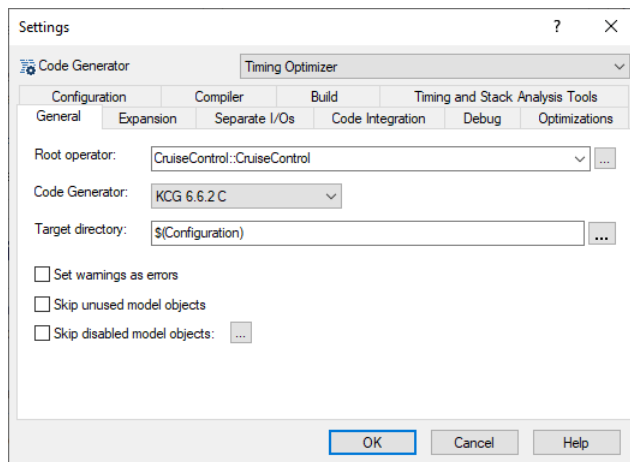


Figure 17.9: SCADE Suite KCG settings for WCET and stack analysis

- 2 Select the active configuration from the drop-down list in upper right corner. It is recommended to use a configuration dedicated to Timing and Stack Analysis Tools. For details about configurations, see [“Managing](#)

[Default or Custom Configurations](#)” on page 117. Selecting configuration in upper right corner of the window only allows to edit the settings of the configuration. It does not enable it for code generation.

- 3 Set the root of the SCADE Suite model from **Root operator** drop-down list.
- 4 Select a code generator from **Code Generator** drop-down list. Available optimization and code generation options are updated accordingly.
- 5 In **Code Integration** settings:

Set **Target** and select the type of analysis session:

- **Timing and Stack Optimizer (TSO)**: to enable timing and stack estimation for design optimization. Use such analysis to optimize model design based on comparison of timing and stack results for different designs.
- **Timing and Stack Verifiers (TSV)**: to enable timing and stack computation for a specific target. Use such analysis to compute timing and stack usage achieved by a model on specific targets.

User-provided sensors declaration: Enable sensor declaration as user-provided (checked) or let the selected Target declare sensors (unchecked). For details, see [User-provided sensors declaration](#) option.

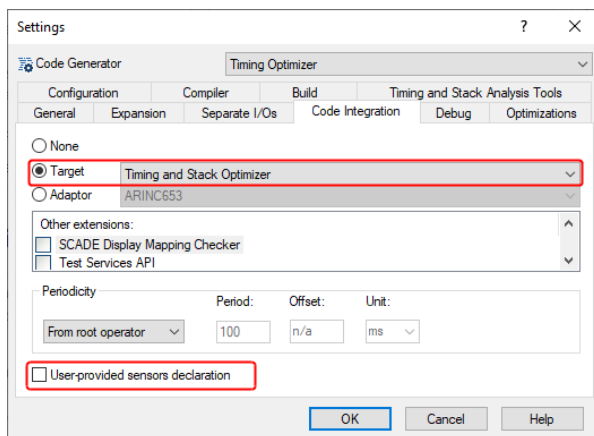


Figure 17.10: Selecting Timing and Stack Analysis Tools

- 6 Display **Optimizations** to optimize the code generation for WCET and stack analysis. For details, see [“Setting Optimization Options”](#) on page 839.
- 7 Display **Compiler** and **Build** to set options for the Timing and Stack binary. For details, see [“Setting Compiler and Build Options”](#) on page 853.
 - If using **Timing and Stack Verifiers** target, it is required to set the compiler and CPU type of the cross-compiler in use. Display **Compiler** and select a cross-compiler from **Compiler** drop-down list and a target CPU from the **CPU type** drop-down list. The compiler and CPU information necessary to build a binary for analysis is only available after defining the cross-compiler to use in a configuration file as detailed in Appendix D about [“Compiler Configuration for SCADE Suite Code Generation”](#).
 - If using **Timing and Stack Optimizer** target, the compiler and CPU type of the cross-compiler are preset.

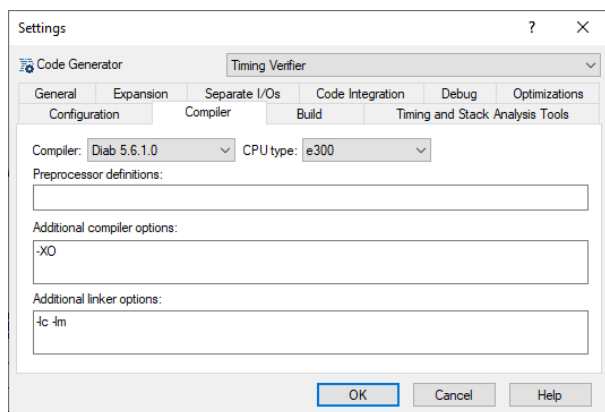


Figure 17.11: Setting compiler and CPU type for Timing and Stack Verifier target

- 8 Display **Timing and Stack Analysis Tools** to set code generation options that influence the computation of WCET and stack analysis.

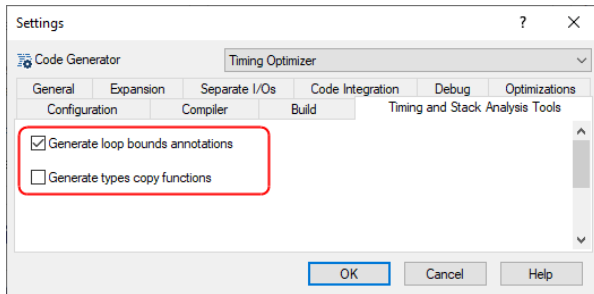


Figure 17.12: Timing and stack code generation options

- 9 Uncheck **Generate loop bounds annotations** if you do not want to add the loop bounds annotations in the aiS constraint file. Timing Analyzer needs this information to determine a maximum bound if automatic detection cannot determine the exact values (e.g., if parameter "Max unroll" is too low). On small models, for more precision in analysis results, it is possible to remove loop bound annotations and set a higher value for [Max unroll](#) until the analysis is performed.
- 10 Check **Generate type copy functions** to generate copy functions "field by field" for complex types instead of using the macro_kcg_assign file.

RELATED TOPICS

Continue with the next phases of the Timing and Stack analysis workflow:

- ["Building Binary for Timing and Stack Analysis"](#)
- ["Setting Timing and Stack Analysis Tools"](#)
- ["Setting Advanced Timing and Stack Analysis"](#)

Building Binary for Timing and Stack Analysis

You can use SCADE Suite KCG Code Generator to generate the code of the model and build an executable binary based on code generator settings.

To build the binary of the model

- 1 Select a configuration set for Timing and Stack analysis in **Code Generator** toolbar.
- 2 Click **Build**  in **Code Generator** toolbar.

SCADE Suite KCG Code Generator generates C code and builds the binary of the model at the ProjectName\ConfigurationName default project directory path.

RELATED TOPICS

Continue with the next phases of the Timing and Stack analysis workflow:

- [“Setting Timing and Stack Analysis Tools”](#)
- [“Setting Advanced Timing and Stack Analysis”](#)

Setting Timing and Stack Analysis Tools

Once the binary for timing and stack analysis is built using SCADE Suite Code Generator, you can specify various hardware characteristics for Timing and Stack Analysis Tools. Modifying the tool settings afterwards does not require to rebuild the binary as they are used for timing and stack analysis only. Configurations are an easy way to preset and reuse options. You simply enable the options and rules needed for timing and stack analysis sessions by activating configurations in toolbars.

Note

For details, see [“Managing Default or Custom Configurations”](#) on page 117.

Once selected, the active configuration automatically enables the options to be edited from the Timing and Stack Analysis Tools **Settings** window.

To set Timing and Stack Analysis Tools options

- 1 Click **Settings**  in **Timing and Stack Analysis Tools** toolbar to display **General** settings.

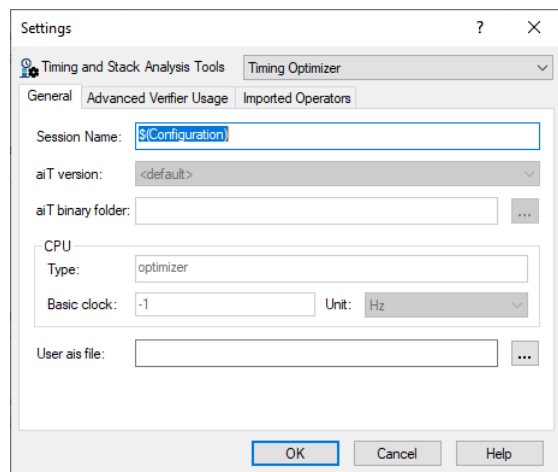


Figure 17.13: Timing and Stack Analysis Tools Settings window

- 2 Select the configuration used to perform WCET or stack analysis from the drop-down list in upper right corner. An executable binary must exist in project folders to launch timing or stack analysis. Without binary, launch commands are disabled.
- 3 Change the name of the created session in **Session Name** to make it different from the active configuration name. The following variables are available for naming sessions: by default `$(Configuration)` references the name of the active configuration, `$(NodePath)` for using the C name of the root operator, and `$(NodeName)` for using the SCADE name of the root operator.
- 4 In **aiT version**, select the AbsInt aiT installation available on the machine. In case the installed aiT version need to be registered manually, specify the folder path containing the aiT binary in **aiT binary folder**.
- 5 Set the CPU-related options (for Timing Verifiers only):
 - **Type** displays the type of CPU target selected for building binary code.
 - In **Basic clock**, specify speed characteristics for the selected CPU.
 - Select the speed unit from **Unit** drop-down list. Available units for CPU speed are:

microseconds: μ s	seconds: s	Kilo Hertz: kHz
milliseconds: ms	Hertz: Hz	Mega Hertz: MHz
- 6 In **User ais file** field, specify a user AIS file by clicking the browsing button and browse for file location. Such files allow to customize the behavior of timing analysis with AIS annotations. For AIS files details: [Search in target-related documentation](#).

You can also access aiT environment from Timing and Stack Analysis Tools to set advanced options from aiT tool environment.

Setting Advanced Timing and Stack Analysis

The advanced options described below must be set by expert users only.

- [“Setting Time Cycle and Stack for Imported Operators”](#)
- [“Setting Advanced Verifiers Options”](#)
- [“Specifying and Referencing aiT Tool Environment”](#)

Once SCADE Suite KCG and Timing and Stack Analysis Tools options are set, you can start Timing and Stack optimization or verification.

Setting Time Cycle and Stack for Imported Operators

Expert users can specify the time cycle and the stack usage for any imported operator defined in SCADE Suite models.

To specify the time cycle and stack of imported operators

- 1 Click **Settings**  in **Timing and Stack Analysis Tools** toolbar and display **Imported Operators**.

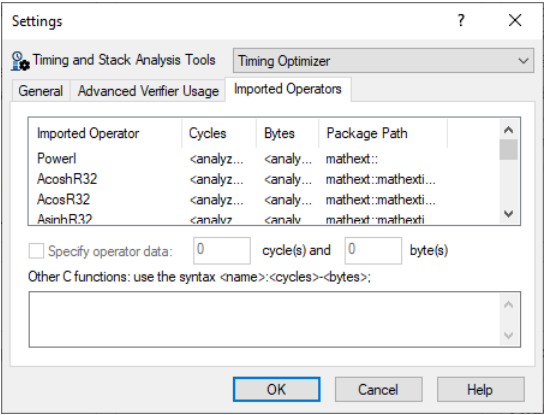


Figure 17.14: Specifying time cycle of imported operators for timing analysis

- 2 Select an operator from the list of imported operators considering:

- Name of each operator for which a **Pragma Name** is specified.
 - Values specified for time cycle and stack usage (default is <analyzed>).
 - The name of the package that contains the imported operator.
- 3 Check **Specify operator data** to enable the specification of time cycle and stack usage values or if the selected operator must not be analyzed. Specify a time cycle value in **cycle(s)** field and a stack usage value in **byte(s)** field. If you do not want to analyze the selected operator, specify 0 (zero) in the appropriate fields. Specified values display in the **Cycles** column of the imported operators list.
- 4 Specify other C functions that are not present in the model as follows:
 <C_function_name>:<nb cycles>-<bytes stack size>;.

Setting Advanced Verifiers Options

Expert users can set additional options for analysis with Timing and Stack Verifiers.

To set advanced options for Timing and Stack Verifiers

- 1 Click **Settings**  in **Timing and Stack Analysis Tools** toolbar and display **Advanced Verifier Usage**.

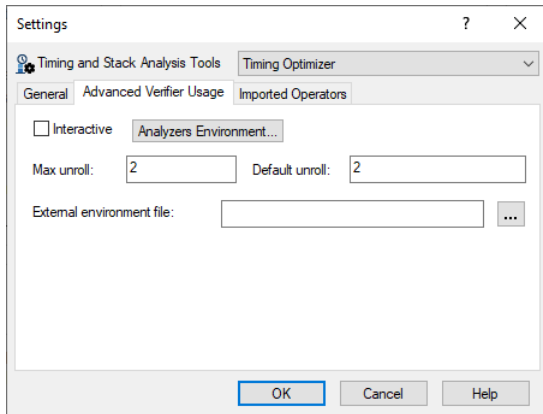


Figure 17.15: Setting advanced options for Timing and Stack Verifiers

- 2 Check **Interactive** to display the results of Timing Verifiers analysis in aiT tool user interface (see [“Launching Analysis in Interactive Mode”](#) on page 810).
- 3 In **Max unroll**, specify the maximum number of calling context for loop analysis (only timing analysis). The higher the value, the finer the analysis, the more resources (time and memory) are required.
- 4 In **Default unroll**, specify the context number for the analysis of loops where the number of iterations is unknown (e.g., imported code) (only timing analysis). For details about “Global Context Specification”: [Search in target-related documentation](#).

Specifying and Referencing aiT Tool Environment

Expert users can access directly the aiT environment, set aiT-specific options, and save the environment description in XTC files. From an XTC file possibly stored on a distant machine, it is possible to reference in SCADE Suite a description of an aiT environment (custom hardware and analysis settings). For Timing and Stack analysis, the original XTC file remains unchanged: it is copied in a `<configuration>.xtc` file and completed with CPU, binary, and ais information from SCADE Suite code generation settings. Refer to aiT documentation for more details about settings in aiT tool.

To access aiT environment, set aiT tool options, and reference environment description

- 1 Click **Settings**  in **Timing and Stack Analysis Tools** toolbar and display **Advanced Verifier Usage**.
- 2 Click **Analyzers Environment** button to open aiT user interface, set aiT-specific options, and save the aiT environment description in an XTC file.
- 3 Back to **Timing and Stack Analysis Tools** settings, in **External environment file**, click the browsing button to specify an external environment XTC (`.xtc`) file as reference. Click **Open** to validate the selected file.

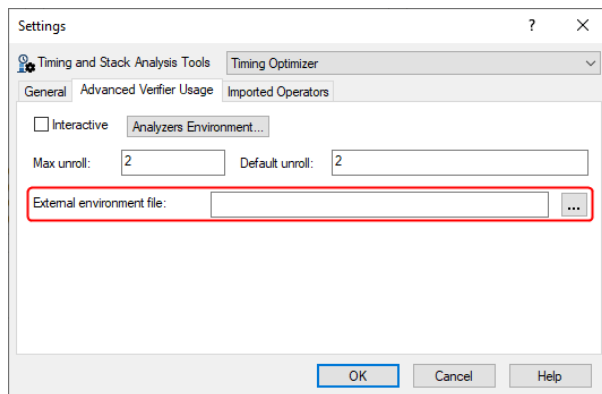


Figure 17.16: Referencing aiT analyzer environment

Launching Timing and Stack Analysis Tools

You perform WCET or stack analysis on the executable binary of the SCADE Suite model. You can use Timing and Stack Analysis Tools to analyze the binary and determine WCET and stack usage.

- [“Launching Timing or Stack Analysis”](#)
- [“Launching Analysis in Interactive Mode”](#)

Launching Timing or Stack Analysis

You can use Timing and Stack Analysis Tools to determine WCET and stack usage in separate analysis sessions.

To launch Timing or Stack analysis

- 1 Select the appropriate configuration from **Timing and Stack Analysis Tools** toolbar. It must be the configuration used to build the binary.

Note

The selected configuration contains all preset options and parameters for timing and stack analysis as described in [“Setting Timing and Stack Analysis Tools”](#) on page 803.

- 2 Launch the analysis as follows:
 - Click **Analyze Timing**  to launch timing analysis.
 - Click **Analyze Stack**  to launch stack usage analysis.

Once analysis completes, the result view is automatically displayed. Several operations are available from timing and stack result views to facilitate the examination of the results.

RELATED TOPIC

- [“Enabling Configurations”](#) on page 119
- [“Examining Timing and Stack Analysis Results”](#) on page 811

Launching Analysis in Interactive Mode

You can launch Timing and Stack Analysis Tools in **Interactive** mode. This mode enables the visualization of analysis results within a³ user interface. You must previously enable the corresponding option in the Timing and Stack Analysis Tools settings (see [“Setting Advanced Verifiers Options”](#) on page 806).

After launching analysis with the interactive mode on from SCADE Suite, a³ requests confirmation of the analyzer build to proceed. You start analysis with respect to a³ documentation. The analysis results are displayed as shown on [Figure 17.17](#). Refer to aiT documentation to access more information about aiT analysis and display various graphs.

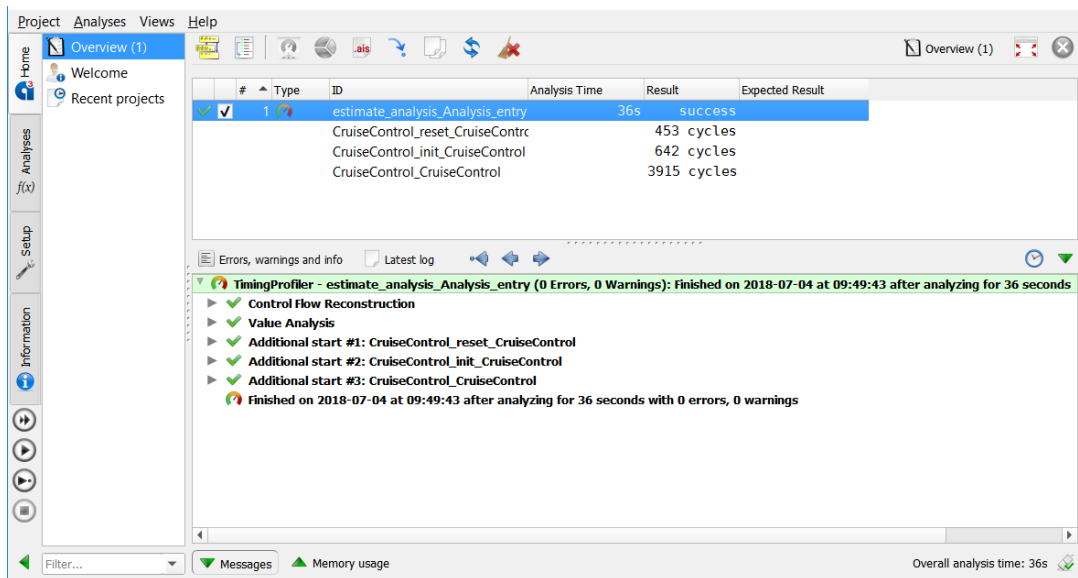


Figure 17.17: aiT displaying timing analysis results

Examining Timing and Stack Analysis Results

When an analysis session is complete, the results are automatically displayed with the timing and stack result views so that you can directly examine them. Furthermore, the session is automatically added in the **Timing and Stack View**. You can also perform several operations to facilitate the examination of the results.

- [“Displaying Analysis Session Results”](#)
- [“Displaying Timing Analysis Detailed Results”](#)
- [“Locating Analyzed Nodes in Model”](#)
- [“Sorting Analysis Results”](#)
- [“Importing External Results”](#)
- [“Updating Analysis Sessions”](#)
- [“Comparing Analysis Sessions”](#)

Tip

For controlling details displayed in results, you must set the Timing and Stack Analysis tools options as described in [“Setting Timing and Stack Result Options”](#) on page 797.

Displaying Analysis Session Results

The **Timing and Stack View** is automatically displayed if a WCET or stack analysis session (SCTS file) exists in the project folder. This view allows displaying all the results of the existing sessions at any moment.

The following information is available for each session in the corresponding view:

- A title specifies the analysis tool used and a subtitle identifies a session name based on the configuration used to run the analysis
- A summary of the code generation options and the main results
- A **Cycle functions** table with results for all non-expanded nodes in the cycle function

- An **Init functions** table with results for all non-expanded nodes in the init function
- A **Reset functions** table with results for all non-expanded nodes in the reset function
- A **Dependencies functions** table with results for all non-Scade functions that are generated and called by the SCADE Suite-generated code. Dependencies are calls to external code. This information displays only if external code is used in the model. It is provided to facilitate the understanding of analysis results.
- A list of abbreviations containing the definitions of the acronyms used in views.

To display the results of an analysis session

- 1 Display the **Timing and Stack View**.
- 2 Double-click on a session name to display the corresponding timing or stack results view. From the result tables, you can:
 - access detailed results for each analyzed operator (only in timing analysis)
 - locate each analyzed operator in the SCADE Suite model
 - sort the results in columns

Timing Optimizer Analysis Result

Timing Optimizer Session

Date: 02/09/2021 14:10:18
Project: C:\Users\Public\Documents\ANSYS Inc\v221\SCADE\examples\CruiseControl\CruiseControl\CruiseControl.etp
Configuration: Timing Optimizer
Analyzer: Optimizer
Max unroll: 2
Default unroll: 2
KCG version: KCG 6.6.2
Keep probes: false
Debug: false
Significant length: 31
Name length: 200
Root inputs and outputs as global variables: false

Result Overview

Root init function CWCET: 642 cycles

Root reset function CWCET: 453 cycles

Root cycle function CWCET: 3095 cycles

Cycle functions

SCADE Critical Path #	Calls	WCET (sum)	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
 CruiseControl::SaturateThrottle	1	223	223	223.00	223	223	223.00
 CruiseControl::CruiseSpeedMgt	1	461	461	461.00	461	461	461.00
 CruiseControl::CruiseRegulation	1	224	224	224.00	447	447	447.00
 CruiseControl::CruiseControl	1	1985	1985	1985.00	3095	3095	3095.00

Figure 17.18: Displaying timing analysis results from Optimizer tool

RELATED TOPICS

- [“Displaying Timing Analysis Detailed Results”](#) on page 814
- [“Locating Analyzed Nodes in Model”](#) on page 816
- [“Sorting Analysis Results”](#) on page 816
- [“Updating Analysis Sessions”](#) on page 819

Displaying Timing Analysis Detailed Results

From WCET analysis result views, you can display the detailed results of each node described in the **Cycle functions**, **Init functions**, and **Reset functions** sections. The following information is available for each detailed view:

- A title specifies the name of the analyzed function (the function name corresponds to its path in the SCADE Suite model) and the related session name
- A summary of the timing results for **Cycle functions** and **Reset functions** with the percentage values of the selected node and an horizontal chart showing the global contribution of the selected node with its descendants
- A table for **Cycle functions**, **Init functions**, and **Reset functions** showing the contribution of each non-expanded node called in the SCADE Suite node instance (if the selected node calls no other function, the table is not displayed)
- A horizontal stacked chart below tables illustrating the contribution of each non-expanded node with a different color for each node
- A list of abbreviations containing the definitions of the acronyms used in views

To display detailed results of a node contribution

- 1 Display the result view of a timing analysis from **Timing and Stack View**.
- 2 Click any hyperlinks in **SCADE Critical Path** column from the **Cycle functions**, **Init functions**, or **Reset functions** tables to display detailed results. From result tables, you can:
 - access detailed results about any other descendant of the node
 - locate each analyzed operator in the SCADE Suite model
 - sort the results in columns

Function CruiseControl::CruiseControl detail (session [Timing Optimizer](#))

CruiseControl::CruiseControl Cycle function

Calls: 1
CWCET (sum): 3095 (100.00%)
WCET (max): 1985 (64.14%)
CWCET (max): 3095 (100.00%)
CWCET (avg): 3095.00

CruiseControl::CruiseControl Cycle function descendant

SCADE Critical Path #	Calls	Kind	Contribution	%
CruiseControl::CruiseSpeedMgt	0	RESET	0	0.00
CruiseControl::CruiseSpeedMgt	1	CYCLE	461	14.89
CruiseControl::CruiseRegulation	1	CYCLE	447	14.44
CruiseControl::CruiseRegulation	3	RESET	202	6.53
CruiseControl::CruiseControl	0	CYCLE	1985	64.14

Figure 17.19: Detailed timing analysis results of a function

If an operator calls one or several iterators, an expand/collapse sign appears in front of the operator in the **SCADE Critical Path** column. Click to show the contribution details of a function and each loop present in the SCADE Suite-generated C code.

Main function		Init functions						
	SCADE Critical Path #	Calls	WCET (sum)	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
ABC_N::Button		4	468	234	117.00	468	234	117.00
ABC_N::ABC_N		1	618	618	618.00	1086	1086	1086.00
Function			311	311		1086	1086	
L1(4)								
[1]			212	212		446	446	
[2..]			95	95		329	329	

Figure 17.20: Displaying contribution details of an operator containing iterators

RELATED TOPICS

- [“Displaying Timing Analysis Detailed Results”](#) on page 814
- [“Locating Analyzed Nodes in Model”](#) on page 816
- [“Sorting Analysis Results”](#) on page 816

Locating Analyzed Nodes in Model

From WCET analysis result views and detailed result views, you can locate any analyzed node in the SCADE Suite model. This feature allows identifying and accessing easily the part of the SCADE Suite model that contributes the most to WCET.

To locate a design element in model

- 1 Display the result view of a Timing or Stack analysis session.
- 2 In any result table, click **Show in model**  in front of the node you wish to locate in the SCADE Suite model. SCADE Suite opens a **Diagram View** of the selected node.

RELATED TOPICS

- [“Displaying Analysis Session Results”](#) on page 811
- [“Displaying Timing Analysis Detailed Results”](#) on page 814
- [“Comparing Analysis Sessions”](#) on page 819

Sorting Analysis Results

Results of a WCET or stack analysis are automatically sorted according to the **SCADE Critical Path** column, and in a decrease order. This is the result views default sorting. You can sort the display of results according to a selected column and in increasing/decreasing order.

To sort the results according to a table column

- Click on the header titles of any result table. An upward arrow appears next to the header title of the column. Results are sorted according to that column. Upward arrow shows results sorted in decreasing order (default order).

Column according to which results are sorted

Up arrow indicates decreasing sorting order

Init functions

SCADE Critical Path	Calls	WCET (sum)	WCET (max)	WCET (avg) ↑	CWCET (sum)	CWCET (max)	CWCET (avg)
CruiseControl::CruiseControl	1	302	302	302.00	642	642	642.00
CruiseControl::CruiseSpeedMgt	1	188	188	188.00	188	188	188.00
CruiseControl::CruiseRegulation	1	152	152	152.00	152	152	152.00

Figure 17.21: Sorting results in columns of result tables

To sort the results in decrease or increase order

- Click once the header of the column according to which the results are sorted (the one with an arrow next to header title). The following happens:
 - The arrow transforms to a downward arrow if it was previously upward. The results are now sorted in increasing order.
 - The arrow transforms to an upward arrow if it was previously downward. The results are now sorted in decreasing order.

Down arrow indicates increasing sorting order

Cycle functions

SCADE Critical Path	Calls	WCET (sum) ↓	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
CruiseControl::SaturateThrottle	1	223	223	223.00	223	223	223.00
CruiseControl::CruiseRegulation	1	224	224	224.00	447	447	447.00
CruiseControl::CruiseSpeedMgt	1	461	461	461.00	461	461	461.00
CruiseControl::CruiseControl	1	1985	1985	1985.00	3095	3095	3095.00

Figure 17.22: Sorting results in increasing/decreasing order

RELATED TOPICS

- [“Displaying Analysis Session Results”](#) on page 811
- [“Displaying Timing Analysis Detailed Results”](#) on page 814
- [“Comparing Analysis Sessions”](#) on page 819

Importing External Results

If you use aiT tools to launch timing and/or stack analysis on executable files containing Scade code, you can import the results (in XML format) in SCADE Suite and examine them using the result views.

To import external results

- 1 Click **Import Timing/Stack Result**  in **Timing and Stack Analysis Tools** toolbar to open the **Import Result** window.

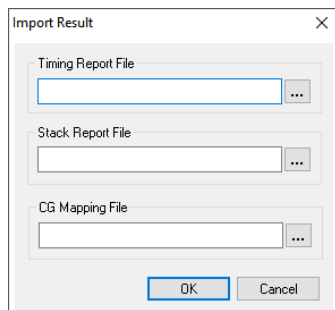


Figure 17.23: Import Result window

- 2 Specify the **Timing Report File** path or click the browsing button to browse for the Timing Report XML file to import in the project.
- 3 Specify the **Stack Report File** path or click the browsing button to browse for the Stack Report XML file to import in the project.
- 4 Specify the **CG Mapping File** path or click the browsing button to browse for the KCG- or MCG-generated Mapping XML file related to the file to import. This file is used by Timing and Stack Analysis Tools to map the analyzed nodes (and the analysis results) to the related SCADE Suite model.

The selected Timing report and/or Stack report is added to project files and the related session is displayed in **Timing and Stack View**. Open it as any other session file and perform any action described in the following sections.

Updating Analysis Sessions

You can run analysis again on previous sessions to update the results.

To update analysis session

- Right-click a session in **Timing and Stack View** and select *Update Session*. The analysis is performed again with the executable present in the code generation target directory and with the current aiT options. Updating session results overwrites all results recorded from a previous session.

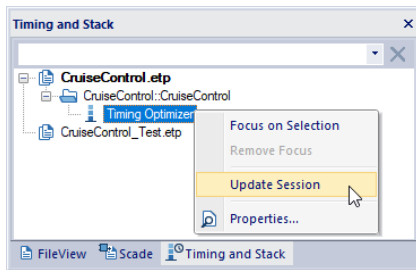


Figure 17.24: Updating analysis sessions

Comparing Analysis Sessions

You can compare the results of two sessions to identify and evaluate optimizations made in SCADE Suite model.

To compare two sessions

- 1 Select two sessions in **Timing and Stack View**, holding C t r L-key.
- 2 Right-click on selection and select *Compare Sessions*.

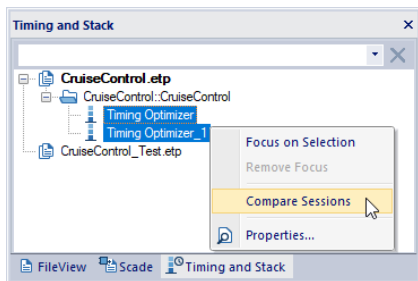


Figure 17.25: Launching comparison between two analysis sessions

As illustrated below, the results of comparison analysis sessions are displayed as in other result views. The **Diff Report** allows to compare results and analyze variations between the oldest and the newest session selected.

To facilitate the identification of differences in the report, the values that are different are represented in two different colors:

- Values that represent a **benefit** (of time or of space) are colored in *green*.
- Values that represent a **loss** (of time or of space) are colored in *red*.

Reset functions

SCADE Critical Path	Calls				WCET (sum)				WCET (max)				WCET (avg)				CWCET (sum)				CWCET (max)				CWCET (avg)			
	ref	vs	Diff	%	ref	vs	Diff	%	ref	vs	Diff	%	ref	vs	Diff	%	ref	vs	Diff	%	ref	vs	Diff	%	ref	vs	Diff	%
CruiseControl::CruiseSpeedMgt	1	2	1	50.00	64	152	88	57.89	64	103	39	37.86	64.00	76.00	12.00	15.79	64	152	88	57.89	64	103	39	37.86	64.00	76.00	12.00	15.79
CruiseControl::CruiseRegulation	4	4	0	0.00	336	283	-53	-18.73	134	103	-31	-30.10	84.00	70.75	-13.25	-18.73	336	283	-53	-18.73	134	103	-31	-30.10	84.00	70.75	-13.25	-18.73
CruiseControl::CruiseControl	1	1	0	0.00	255	274	19	6.93	255	274	19	6.93	255.00	274.00	19.00	6.93	453	426	-27	-6.34	453	426	-27	-6.34	453.00	426.00	-27.00	-6.34

Figure 17.26: Sample of comparison results from two different analysis sessions

Tip

Sessions Comparison results are not added to the project files. If you want to keep them at the end of examination, you can save them from *File > Save as*.

RELATED TOPICS

- [“Locating Analyzed Nodes in Model”](#) on page 816
- [“Sorting Analysis Results”](#) on page 816

Part 4

Code Generation

- Chapter 18: [“Fundamentals about SCADE Suite Code Generation”](#)
- Chapter 19: [“Setting and Launching Code Generation”](#)
- Chapter 20: [“Implementing Models with Imported Code”](#)
- Chapter 21: [“Integrating Monolithic Code”](#)
- Chapter 22: [“Generating and Integrating Task-Based Code”](#)

18 / Fundamentals about SCADE Suite Code Generation

This chapter is an overview of the basic concepts, mechanisms, and user interface that support code generation from SCADE Suite models.

- [“Code Generation Principles”](#)
- [“Exploring Code Generator User Interface”](#)
- [“Code Generation Workflow”](#)

Code Generation Principles

Code generation from SCADE Suite models is based on the following notions.

- [“Generation of Monolithic or Task-Based Code”](#)
- [“SCADE Suite Input Models”](#)
- [“Code Generation Files”](#)
- [“Optimization of Generated Code”](#)
- [“Assertions in Generated Code”](#)
- [“Probes in Generated Code”](#)
- [“Code Generation Checks”](#)

Generation of Monolithic or Task-Based Code

SCADE Suite code generation capabilities empower users to produce monolithic or task-based code from SCADE Suite models. Both *monolithic* and *task-based* versions of code generated from SCADE Suite models have the same predictable deterministic behavior.

SCADE Suite KCG provides the capability to generate monolithic C or Ada code from SCADE Suite models. The KCG output code is a single cycle function that allows running applications developed with SCADE Suite on single core

targets. SCADE Suite KCG is a qualified Code Generator. For instructions about code generation with KCG, see Chapter 19 about [“Setting and Launching Code Generation”](#) and Chapter 21 about [“Integrating Monolithic Code”](#).

Note

SCADE Suite KCG Code Generator integrated into SCADE Suite user interface is available with Certification Kits for DO-178C, IEC 61508, ISO 26262, and EN 50128 standards.

SCADE Suite Multicore Code Generator (MCG) provides the capability to generate task-based C code for multicore target integration from any legacy SCADE Suite models. The production of such code requires minimal additions to existing SCADE Suite models. This code generator is able to extract each operator instance into separate tasks by means of specific pragma attached on operator instances at model level to express parallelism. MCG output code is no longer a single cycle function but a set of tasks communicating through one-to-one channels, namely a Kahn Process Network (KPN), that allows deploying applications developed with SCADE Suite on multicore target. The KPN is described in the generated traceability information. This code is also target independent and can be easily integrated in many different contexts such as the number of cores (from 2 to several hundred), the nature of communications (shared memory or message passing), or the real-time OS in use. For instructions about code generation with MCG, see Chapter 22 about [“Generating and Integrating Task-Based Code”](#).

Both Code Generators support the implementation of imported code in models as described in Chapter 20 about [“Implementing Models with Imported Code”](#). Multicore Code Generator supports the same pragmas as SCADE Suite KCG 6.6.2 for C code generation. For setting non MCG-specific pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859. For the syntax of non MCG-specific pragmas, see [“Pragma Syntax and Usage”](#) on page 239 in *SCADE Suite Technical Manual*.

SCADE Suite Input Models

SCADE Suite Code Generators take SCADE Suite models composed of SCADE Suite graphics and/or Scade textual format as input and produce code behaving as the input model. As batch tools they take as inputs:

- Models in textual or graphical format (XML file)
- Command line options to modify generated file names, code API, and code optimization
- The root operator from which code is generated

From the above inputs, SCADE Suite Code Generators produce:

- C code (KCG and MCG) or Ada code (KCG only)
- Textual Scade description when input model in graphical format
- Log file containing errors and warnings found during input model checking
- Traceability information between the model and the generated code
- Other information for helping integration

Code Generation Files

SCADE Suite code generation requires a set of input and output files used by KCG / MCG Code Generators. The basic description of the files and outputs created during code generation is illustrated in [Figure 18.1](#).

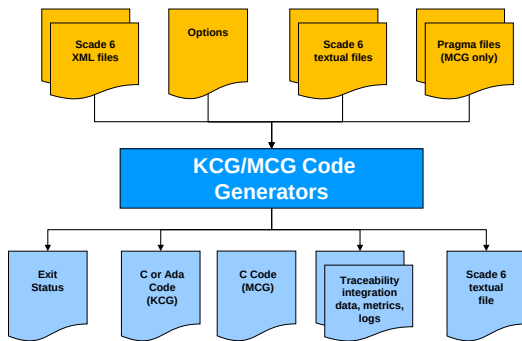


Figure 18.1: Input and output files of KCG / MCG Code Generators

Optimization of Generated Code

The target code generated by KCG/MCG is optimized by default. Several options allow to modify optimization levels, constrain the number of inputs on non-expanded operators, or optimize local variables. For guidelines about code optimization and options, see [“Setting Optimization Options”](#) on page 839. For the correspondence between batch and user interface options, see Chapter 11 about [“Code Generators Command Line Options”](#) in *SCADE Suite Technical Manual*.

Assertions in Generated Code

Assertions are introduced by two Scade language constructs: Assume and Guarantee. Both are used to describe the “contract” of the node. Assume is used to describe the condition of use. Guarantee is used to describe the invariant property of the node. For option selection, see [“Setting Code Configuration Options”](#) on page 842. For details about assertions in Scade language, see [“Scade Contracts: Assume and Guarantee Observers”](#) on page 92 in *Scade Language Primer*. For assertions and batch code generation, see [“Option -keep_asserts”](#) on page 314 in *SCADE Suite Technical Manual*.

Probes in Generated Code

A probe is a variable that you want to keep in the generated code in order to observe the values it carries. This pertains to local variables. To specify that a local variable is a probe, the "probe" keyword precedes the declaration of related variable.

For option selection, see [“Setting Code Configuration Options”](#) on page 842. For probes syntax, see [“Probing a Flow”](#) on page 94 in *Scade Language Primer*. For probes and batch code generation, see [“Option -probes”](#) on page 316 in *SCADE Suite Technical Manual*.

Example: Unlike a, y, and z variables, x is a probe

```
node N(e:int) returns (s:int);  
var  
  probe x, y, z :int;  
  a : bool;  
let;
```

Code Generation Checks

Prior to code generation, the syntax and semantics of the model description provided to SCADE Suite Code Generators are checked.

- [“Syntactic Checks”](#)
- [“Semantic Checks”](#)

Syntactic Checks

SCADE Suite Code Generators perform a lexical and syntactical analysis of the description of the input model. Its scope of analysis is the file content as well as all files included in the SCADE Suite project.

Note

External type syntax is analyzed with respect to Scade syntax. The scope of analysis is the content of files related to imported types and all other project files.

During such verification phase, SCADE Suite Code Generators can detect two types of syntax inconsistencies:

- Inconsistencies that do not cause misunderstandings but which must be mentioned to the user. These are called **syntax warnings**.
- Inconsistencies that prevent understanding the description (e.g., missing operands of an operator). These are referred to as **syntax errors**.

The number of errors and/or warnings detected is indicated at the end of the analysis phase. If a syntax error is detected, code generation is stopped. Warnings do not prohibit continuation with the next phase of verification. For a description of possible error and warning messages returned during code generation, consult Chapter 12 about [“Code Generation Messages”](#) in *SCADE Suite Technical Manual*.

Semantic Checks

When no problems are detected during syntax analysis, a certain number of checks related to the Scade language are performed. The semantic checks are performed in the following sequence:

- **Post-syntax analysis:** to verify that the input model is correct with respect to syntactic rules that are not detected at model loading.
- **Namespace analysis:** to verify that all names specified in the input model are correctly declared in the model.
- **Type analysis:** to verify that the input model is correctly typed.
- **Clock analysis:** to verify that the input model is correctly clocked.
- **Causality analysis:** to verify that all input model dependencies are correct.
- **Initialization analysis:** to verify that the flows of the input model are correctly initialized.

All checks are performed systematically for code generation and cannot be customized.

The verification covers the description or operators (nodes or functions) that are part of the call graph of the root operator defined for code generation. All global entities (like types and constants) are analyzed. Some variations may result in **semantic errors** and others in **semantic warnings**.

For more details, consult Chapter 12 about [“Code Generation Messages”](#) in *SCADE Suite Technical Manual*.

Exploring Code Generator User Interface

SCADE Suite Code Generators are fully integrated in SCADE Suite Editor. For code generation sessions, SCADE Suite provides the following components.

- [“Code Generation Toolbar”](#)
- [“Code Generation Capabilities”](#)
- [“Code Generator Settings”](#)
- [“Display of Build and Log Information”](#)

Code Generation Toolbar

This toolbar provides shortcuts to access code generation options and manage code generation sessions. It enables you to generate embeddable code, to generate additional wrapping code, to build, or to execute code. It can be moved in the interface by dragging the title bar to the desired location.

Code Generator	Managing code generation sessions: • “Setting and Launching Code Generation” on page 835 Launching simulation: • “Launching Simulation Session” on page 596

To show/hide toolbar

- Select *View > Toolbars > Code Generator*; or right-click in toolbar area and select the name of the toolbar to show/hide. Alternatively, hide the toolbar by right-clicking on toolbar and selecting *Hide*.

Code Generator Settings

You use this window to specify the operator for which code is generated and modify the code generation options specified in the default configurations.

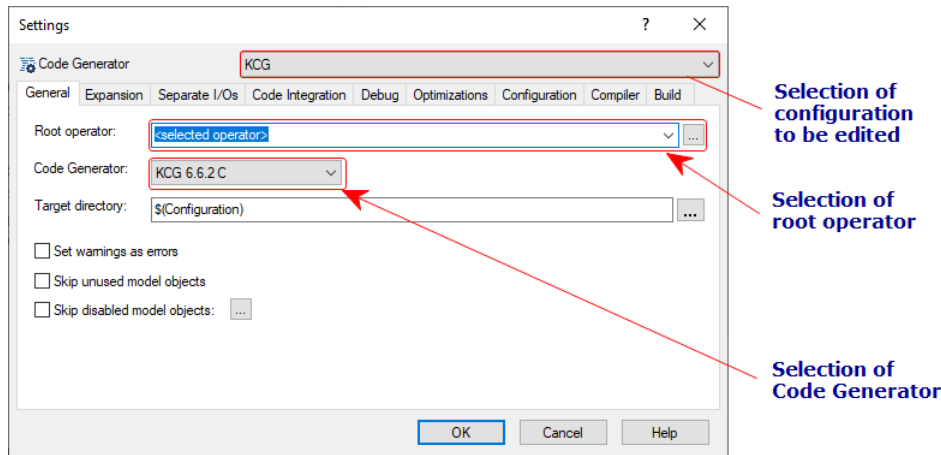


Figure 18.2: Setting window for code generation options

For user instructions about all code generation options, see Chapter 19 about [“Setting and Launching Code Generation”](#).

Code Generation Capabilities

SCADE Suite Code Generators may be extended for use and implementation in different environments by the following code generation capabilities:

- *Target code generation*: to generate and build an executable or DLL for specific execution environments. Typically, you use target code generation for simulation, model coverage, timing and stack analysis in SCADE Suite, and for all gateways enabling co-simulation with other tool environments.

- *Adaptor code generation*: to generate wrapping code necessary to adapt KCG-generated code to specific environments. Typically, you use adaptor code generation for all RTOS supported by SCADE Suite, for model calibration with ASAM MCD-2-MC (a.k.a. ASAP2) information, and for code generation with models containing SCADE Display graphical panels.
- *Other code extensions*: to produce additional files that depend on the SCADE Suite model such as mapping or configuration files. Typically, these extensions are used with Model Coverage to generate additional files to be included in the build of the simulation DLL.

Code Generator settings are used for specifying code generation capabilities.

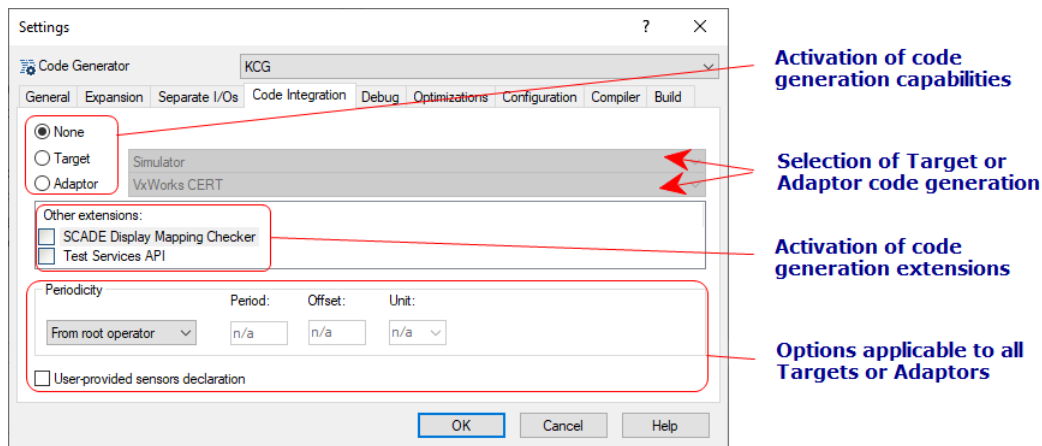


Figure 18.3: Enabling Code Generator to produce extended code

For user instructions about code extension settings, see [“Setting Code Integration Capabilities”](#) on page 849.

Display of Build and Log Information

You can read run-time info about ongoing code generation and access generated files.

- Consult status information sent by SCADE Suite Code Generators in the **Build** output tab and check whether code generation succeeded.

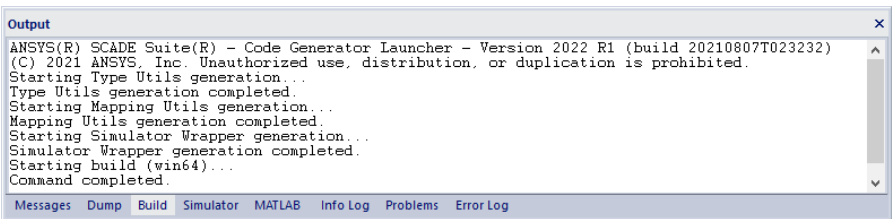


Figure 18.4: Code generation build information

- At code generation completion, SCADE automatically displays the **Info Log** output tab. Check for log file information, files generated by KCG/MCG, or any possible errors or warnings encountered during code translation. In **Info Log** tab, navigate to generated files as follows:
 - Double click files to open them in SCADE Suite.
 - Right-click any file category or any file in the list and select *Show in Folder* to open the directory containing the corresponding file(s).

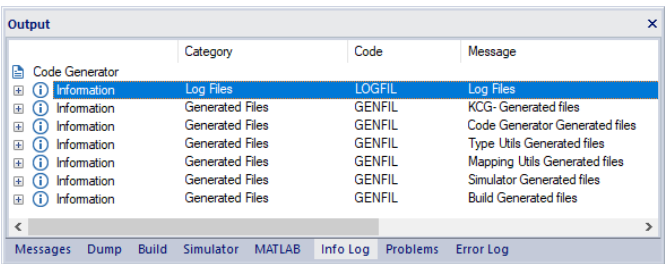


Figure 18.5: Code generation log information

At code generation completion, a `config.txt` file lists all options produced by SCADE Suite to launch Code Generator in batch.

Code Generation Workflow

The workflow for monolithic code generation consists of various tasks that you can perform at model level before generating code and integrating it on target environment.

For task-based code generation purposes, the workflow is slightly different and implies specific user tasks. For details, refer to [“Task-Based Code Generation Workflow”](#) on page 897.

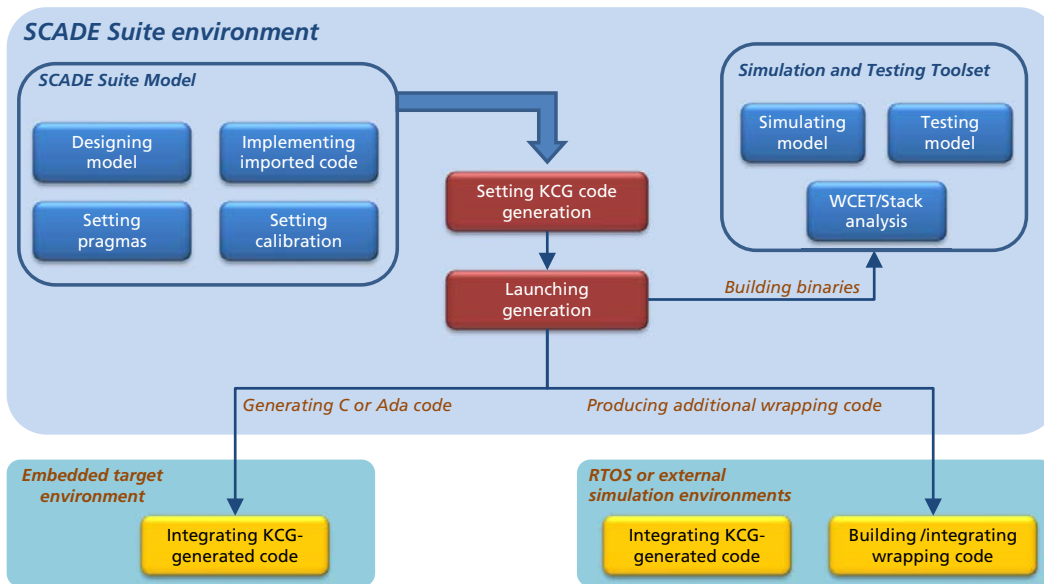


Figure 18.6: Code generation workflow with SCADE Suite KCG

The following user tasks are available for code generation purposes:

- 1 Setting pragmas at model level: [“Setting Pragmas on Model Elements”](#) on page 859.
- 2 Preparing models with calibration information: [SCADE Automotive Package Guidelines](#).

- 3 Setting generation options with respect to target code and environment: Chapter 19 about [“Setting and Launching Code Generation”](#).
- 4 Implementing models with imported code: Chapter 20 about [“Implementing Models with Imported Code”](#).
- 5 Launching generation: [“Launching KCG Code Generation from UI”](#) on page 864 or [“Launching KCG Executable in Command Line”](#) on page 866.
- 6 Integrating code on target: Chapter 21 about [“Integrating Monolithic Code”](#).

19 /Setting and Launching Code Generation

This chapter explains how one can configure code generation and how one can launch code generation from SCADE Suite user interface or in command line. It focuses on the generic settings available for SCADE Suite Code Generators. The generation of target code can be handled using generic settings. Code optimization and generation is complemented with a few code-specific options.

- [“Specifying Code Generation”](#)
- [“Setting Optimization Options”](#)
- [“Setting Debug Options”](#)
- [“Setting Code Configuration Options”](#)
- [“Setting Code Expansion Mode”](#)
- [“Setting Code Integration Capabilities”](#)
- [“Setting Periodicity of Code Execution”](#)
- [“Setting Compiler and Build Options”](#)
- [“Adjusting Variable Name Length for Compiler”](#)
- [“Setting I/O Separation Options”](#)
- [“Setting Pragmas on Model Elements”](#)
- [“Launching KCG Code Generation from UI”](#)
- [“Launching KCG Executable in Command Line”](#)

Configurations are an easy way to preset and reuse options. You simply enable the options and rules needed for code generation by activating configurations in toolbars. For details, see [“Managing Default or Custom Configurations”](#) on page 117.

Once selected, the active configuration automatically enables the code generation options that you can edit from the Code Generator Settings window.

Specifying Code Generation

Code generation requires that you specify a code generator, a file directory for generated code, and the operator(s) used as root in the model.

Note

For technical details about these options in command line, consult [“Description of General Options”](#) on page 288 in *SCADE Suite Technical Manual*.

To specify code generation

- 1 Click **Settings**  in **Code Generator** toolbar to display **General** settings.

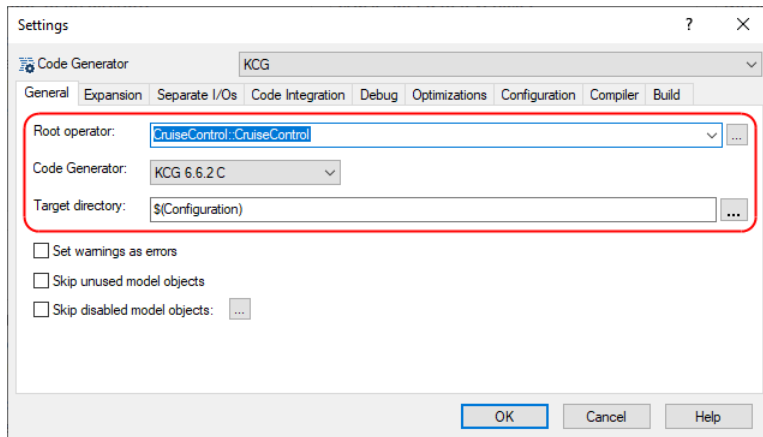


Figure 19.1: Specifying mandatory settings for code generation

Warning

Selecting configuration from drop-down list in upper right corner of the window only allows to edit the settings of the configuration. It does not enable it for code generation.

- 2 Specify the root operator from the **Root operator** drop-down list. SCADE Suite generates code for the operator selected in the **Scade View** when `<selected operator>` is selected.

It is possible to specify multiple root operators with KCG and MCG: click the browsing button to open the **Root Operators Selection** dialog, expand the selection tree, check/uncheck operators, and click **OK** to validate. All selected operators are displayed separated by a comma in the **Root operator** field.

- 3 Set the code generator from the **Code Generator** drop-down list:
 - **KCG 6.6.2 C** or **KCG 6.6.4 C** to generate C code with SCADE Suite KCG.
 - **KCG 6.6.2 Ada** (for certification activities) or **KCG 6.6.4 Ada** (no certification credits) to generate Ada code with SCADE Suite KCG. Package constraints in Ada and Scade being different, code generation in Ada requires to pay attention to package structure: refer to [“Direct Package Hierarchy Correspondence”](#) on page 232 in *SCADE Suite Technical Manual*.
 - **Multicore Code Generator** to generate task-based code with Multicore Code Generator (see Chapter 22 about [“Generating and Integrating Task-Based Code”](#)).
 - **AUTOSAR Code Generator** to generate code for Software Components and ARXML information for integration on AUTOSAR platforms. For details, consult [SCADE Automotive Package Guidelines](#).
 - **<select generator>** appears if the settings reference an obsolete code generator.
- 4 Click the browsing button to change the default location of generated files set in **Target directory**. List of environment variables for directory naming:
 - `$(Configuration)`(default): references the name of the active configuration.
 - `$(NodeName)`: references the name of the first root operator in current list.
 - `$(CG)`: references the name of the current code generator.

- 5 Check **Set warnings as errors** to consider warnings as errors during code generation and stop generating code whenever a warning is raised.
- 6 Check **Skip unused model objects** to generate code corresponding only to the call graph of the root operator(s).
- 7 Check **Skip disabled model objects** to ignore elements during code generation. Click the browsing button to select and disable elements (package, operator, or diagram).

The last two options also allow to simulate incomplete models or models with erroneous parts as detailed in [“Enabling Partial Model Simulation”](#) on page 593.

Setting Optimization Options

SCADE Suite KCG and Multicore Code Generator generate directly optimized code. You set optimization options that impact globally the generated C code. All options available in Code Generator settings contribute to optimize the memory and the execution time of produced code.

Note

For technical details about these options in command line, consult [“Description of Optimization Options”](#) on page 292 in *SCADE Suite Technical Manual*.

To set optimization options (not for Ada code generation)

- 1 Click **Settings**  in **Code Generator** toolbar and display **Optimizations**.

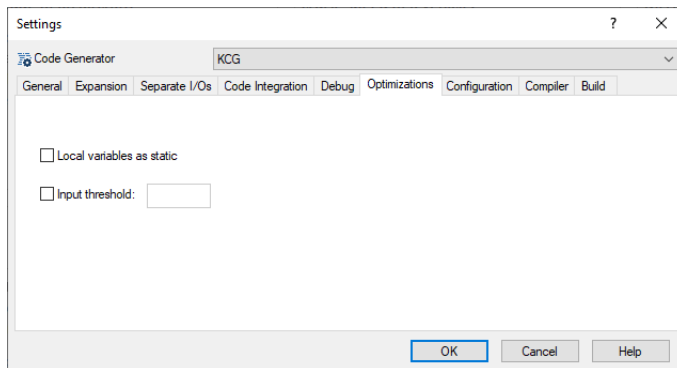


Figure 19.2: Setting global optimization options

- 2 Check **Local variables as static** to perform standard optimization on local variables: a “static” C class storage specifier added in front of C local variables. In case of MCG, this option raises an error if tasks are generated.
- 3 Check **Input threshold** to restrict the number of inputs for cyclic functions generated for non-expanded operators. Specify an integer value. If the number of inputs exceeds the specified value, the definition of a structure containing the inputs is generated. Not applicable to root operator.

Setting Debug Options

When generating code for simulation with SCADE Suite Code Generators, you must set which model elements are observable. You set debug options to add in the generated code local variables with code counterpart, probes, and assertions to the operator context. You can use the following debug options:

- All operators are observable.
- A list of operators identified as *observable operators*.
- A list of operators identified as *non-observable operators*.

Note

For more details about debug options, see [“Description of Debug Options”](#) on page 295 in *SCADE Suite Technical Manual*. No debug options are applicable to Ada code generation.

To set debug options

- 1 Click **Settings**  in **Code Generator** toolbar and display **Debug** to enable the debug options as follows:
 - Check **Debug All** to make observable all operators in the model.

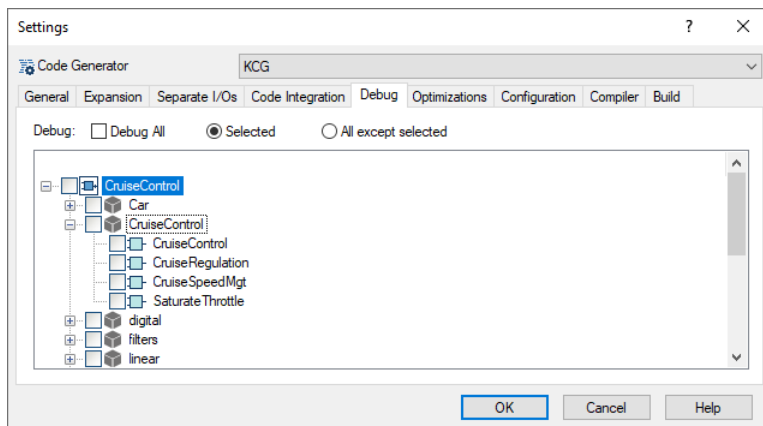


Figure 19.3: Setting debug options to make all operators observable

- Uncheck **Debug All** and set **Selected** radio button to define the list of observable operators. Unchecked operators are identified as operators that are not observable.

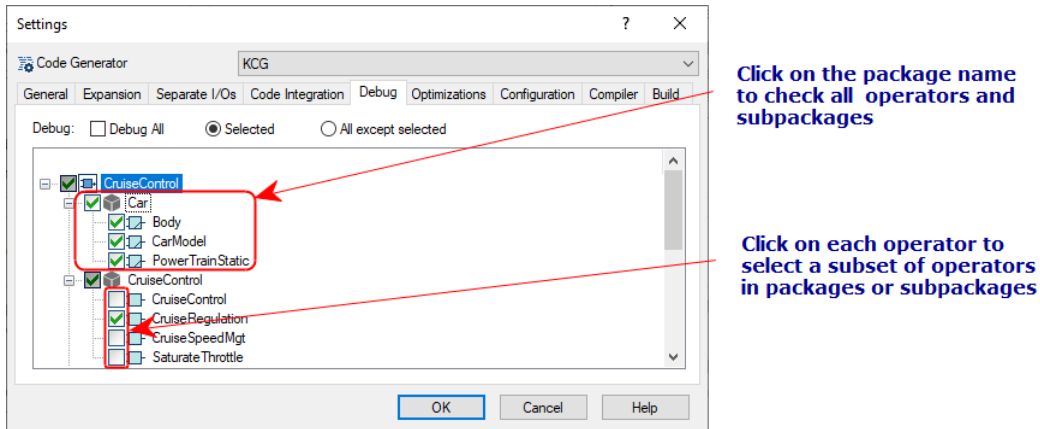


Figure 19.4: Setting debug options to select observable operators

Tips

- Check a package to check all its content.
- Hold **C t r l**-key to set separate items (package or operator names) one by one in multiselection and press **S p a c e** bar to check/uncheck the highlighted list.
- Hold **S h i f t**-key to set consecutive items (package or operator names) in multiselection and press **S p a c e** bar to check/uncheck the highlighted list.
- To check/uncheck by mouse click a multiselection list, keep the **C t r l** or **S h i f t** keys pressed while clicking on checkboxes.

- Uncheck **Debug All** and set **All except selected** radio button to define the list of non-observable operators. Unchecked operators are identified as observable operators.

Setting Code Configuration Options

You may want to configure the generated code with specific content and information, or customize the content and naming of generated files to fit with one's own development environment, policies or needs.

Note

For technical details about these options in command line, consult [“Description of Configuration Options”](#) on page 309 in *SCADE Suite Technical Manual*.

- [“To set C code configuration options”](#)
- [“To set Ada code configuration options”](#)

To set C code configuration options

- 1 Click **Settings**  in **Code Generator** toolbar and display **Configuration**. Select MCG or KCG for C code generation to access these options.

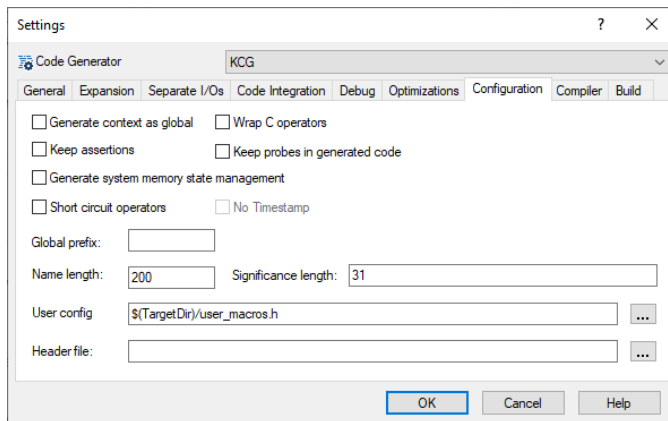


Figure 19.5: Setting C code configuration options

- 2 Check **Generate context as global** to generate the context and the inputs of the root operator as C global variables. (Not available for AUTOSAR code generation)

- 3 Check **Wrap C operators** to wrap arithmetic, binary, or relation operators in C protected macros (see [“Option -wrap_c_ops”](#) on page 326 in *SCADE Suite Technical Manual* for the complete list of protected macros by operators). If users want to redefine such protected macros, they need to give the new implementation in `user_macro.h`. Such customization may have an impact on SCADE tools like SCADE Suite Simulation, SCADE Test Model Coverage, and SCADE Test Environment for Host. For details, see [“Implementing C Macro User Definitions”](#) on page 676.
- 4 Check **Keep assertions** to translate assume and guarantee constructs using macros in the generated code. Generated macros are empty; provide their definitions in user configuration file specified below in **User config**.
- 5 Check **Keep probes in generated code** when required to keep in generated code all variables corresponding to probes in their context.
- 6 Check **Generate system memory state management** to add in generated code memory data and functions for saving and restoring operator state.
- 7 Check **Short circuit operators** to generate the short circuit Boolean operators instead of using C bitwise operators.
- 8 Check **No timestamp** to generate KCG C files without date information in header and footers. (Only with KCG 6.6.4)
- 9 In **Global prefix**, specify the prefix added on names of C global variables and used macros for protection against multiple inclusions. The prefix applies to sensors, types, constants, root operator, non-expanded functions, macros, global variables.
- 10 Specify **Name length** and **Significance length** if the compiler in use supports specific C identifier sizes. See [“Adjusting Variable Name Length for Compiler”](#) on page 856.

11 In **User config**, specify H file to be included at the beginning of `kcg_types.h`. Such file allows to modify default type/macro definitions as detailed in [“Implementing C Macro User Definitions”](#) on page 676. User can set configuration file as follows:

- Leave empty: no file is included.
- Specify any user file: the file is directly set in the command line used when launching code generation. Users are responsible of the content of such files.
- Specify `user_macros.h`: the file is automatically built when launching code generation from UI. It contains inclusion of all header files referenced in the project and identified as [macro file](#).

Note

The specified file path is relative to the target directory to comply with C preprocessor search include rules. The `$(TargetDir)` internal environment variable can be used to reference the the directory where generated code is saved.

12 In **Header file** field, browse for files and specify a file whose content is added at the beginning of each generated file before the banner generated by Code Generator. The specified file must contain only valid C comments. A relative path is interpreted as relative to the project directory.

To set Ada code configuration options

- 1** Click **Settings**  in **Code Generator** toolbar and display **Configuration**. You must select KCG for Ada code generation to access these options.

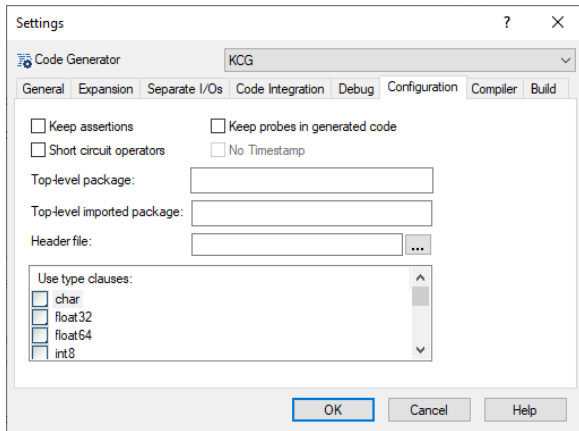


Figure 19.6: Setting Ada code configuration options

- 2 Check **Keep assertions** to translate assume/guarantee constructs using procedure calls in the generated code. Next to generated code, users must provide all procedure definitions in `kcg_config.ads`/`adb` files. For details, see [“Description of External Files for Ada Code”](#) on page 226 in *SCADE Suite Technical Manual*.
- 3 Check **Keep probes in generated code** when required to keep in generated code all variables corresponding to probes in their context.
- 4 Check **Short circuit operators** to generate Boolean rather than bitwise operators.
- 5 Check **No Timestamp** to generate KCG Ada files without date information in header and footers. (Only with KCG 6.6.4)
- 6 In **Top-level package**, specify a path where Ada code is generated.
- 7 In **Top-level imported package**, specify a path where imported code is provided.

- 8 In **Header file**, browse for files and specify a file whose content is added at the beginning of each file before the banner. The specified file must contain valid Ada comments. A relative path is interpreted as relative to the project directory.
- 9 Set **Use type clauses** to provide operators with visibility on selected predefined Scade types (bool, char, float32, float64, int8, int16, int32, int64, uint8, uint16, uint32, uint64). For each type, a use type clause is generated in the Ada code.

Setting Code Expansion Mode

It is possible to select between three modes of code expansion with SCADE Suite Code Generators. By default, no expansion is performed. The modes specify expansion as follows:

- All operators are expanded.
- A list of operators identified as *non-expanded operators*: For each operator in the list, there is a corresponding C function in the generated code.
- A list of operators identified as *expanded operators*: The code of operators is inlined inside the generated code of the first non-expanded operator they are instantiated in.

Note

For technical details about these modes in command line, consult [“Description of Expansion Options”](#) on page 299 in *SCADE Suite Technical Manual*.

To set code expansion mode

- 1 Click **Settings**  in **Code Generator** toolbar and display **Expansion** to enable the expansion mode as follows:
 - Set **All** to expand all operators in the model.

Tips

- An operator set with expand pragma is marked as ‘expanded’ or with no_expand pragma is marked as ‘not expanded’ in the list of operators. About pragma settings, see [“Setting Pragmas on Model Elements”](#) on page 859.
 - To remove all operators unused by the selected root operator from this list, select the [Skip unused model objects](#) option in the General settings.
-

- Set **Selected** (default) to define the list of expanded operators. Unchecked operators are operators generated with no expansion.

Note

For causality analysis, KCG and MCG support the expansion of root operators: the root operators can be present in the selection tree.

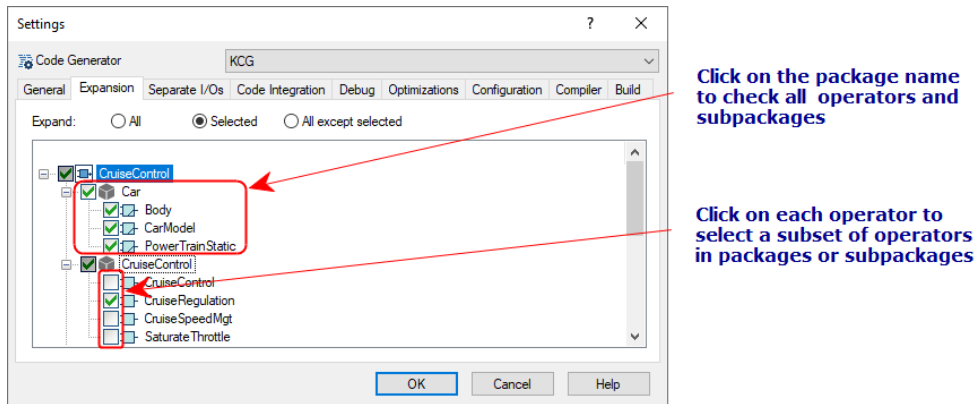


Figure 19.7: Setting code expansion on selected operators only

- Set **All except selected** to define the list of non-expanded operators. Unchecked operators are identified as expanded operators.

Tips

- Check a package to check all its content.
- Hold **Ctrl**-key to set separate items (package or operator names) one by one in multiselection and press **Space** bar to check/uncheck the highlighted list.
- Hold **Shift**-key to set consecutive items (package or operator names) in multiselection and press **Space** bar to check/uncheck the highlighted list.
- To check/uncheck by mouse click a multiselection list, keep the **Ctrl** or **Shift** keys pressed while clicking on checkboxes.

RELATED TOPICS

- [“Setting Variable Visibility”](#) on page 594

Setting Code Integration Capabilities

Whenever you need to use the code generated from SCADE Suite models in dedicated development environments, you can generate additional wrapping code or build executable code dedicated to the environment. By default, no code integration capability is enabled unless using configuration settings dedicated to the various SCADE Suite tools (see [“Enabling Configurations”](#) on page 119).

To enable and set code integration

- 1 Click **Settings**  in **Code Generator** toolbar and display **Code Integration** to set code generation as follows:
 - **None**: only embeddable code is produced during code generation.
 - **Target**: code generation builds executable or DLL for the selected target.
 - **Adaptor**: code generation produces additional wrapping code.

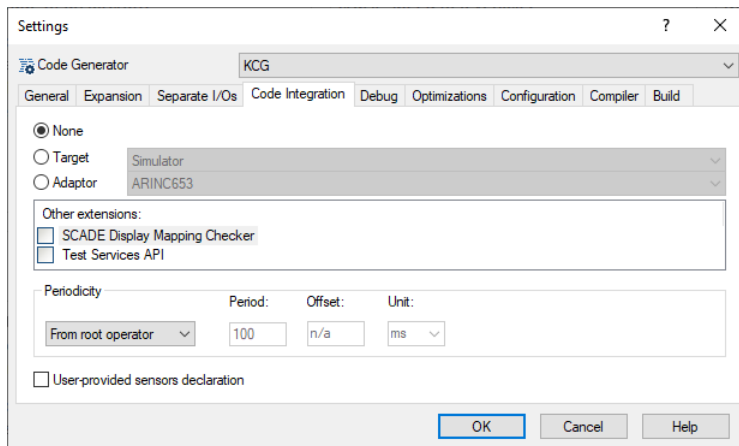


Figure 19.8: Enabling code integration capabilities

- 2 If setting code generation for specific execution environment, select the environment in **Target** drop-down list (not applicable to MCG). After selection, the **Compiler** and **Build** settings appear. The **Compiler** settings contain default compiler options relevant to the target. You may edit these options as described in [“Setting Compiler and Build Options”](#) on page 853.

FMU	Export SCADE Suite models as FMU (Functional Mockup Unit) for simulation in FMI-compliant tools, see “Simulating Models in FMI-Compliant Tools” on page 629. Only C
Mobile	Generation of mobile applications based on graphical panels, see Chapter 15 about “Generating Desktop and Mobile Applications” . Only C
NI VeriStand	DLL build for co-simulation with VeriStand environment as detailed in Gateway Guidelines for VeriStand . Only C
Simulator	Simulation executable before launching SCADE Suite simulation sessions, see Chapter 12 about “Running Simulation in SCADE Suite” . C or Ada
Simulink	DLL build for co-simulation with Simulink environment, see in “Simulating Models in Simulink Environment” on page 656. Only C
Standalone Executable	Generation of desktop applications with graphical panels, see Chapter 15 about “Generating Desktop and Mobile Applications” . Only C
Timing and Stack Optimizer	Generation of executable binary to analyze worst-case execution time and stack usage, see Chapter 17 about “Verifying Timing and Stack Usage” . Only C
Timing and Stack Verifiers	
Twin Builder	Export of SCADE Suite models in for simulation in Ansys Twin Builder, see “Simulating Models in Twin Builder Environment” on page 647. Only C

- 3 For generating specific wrapping code for dedicated environment, select an adaptor in **Adaptor** drop-down list (not applicable to MCG).

ARINC 653	Production of ARINC 653 compliant code, see “Generating A653 Configuration Records and Code” on page 59 in <i>SCADE Avionics Package Guidelines</i> . Only C
ASAM MCD-2 MC	Production of calibrated code, see Appendix D about “Calibrating C Code for ASAM MCD-2 MC” in <i>SCADE Automotive Package Guidelines</i> . Only C
dSPACE MicroAutoBox	Production of code for MicroAutoBox, see Appendix C about “Fast Function Prototyping with dSPACE MicroAutoBox II” in <i>SCADE Automotive Package Guidelines</i> .
FACE	Production of FACE code for integration on FACE platforms, see “Generating C and C++ Code for FACE” on page 129 in <i>SCADE Avionics Package Guidelines</i> . Only C
INTEGRITY-178B / MicroC_OS-II / OSEK / PikeOS / VxWorks CERT	Production of RTOS-specific files, see RTOS Adaptor Guidelines . Only C
SCADE Display Co-generation	C code generation of display applications, see “Generating and Integrating SCADE Display Code” on page 885. The generated code includes graphics, logics, and a wrapping C code between SCADE Suite-generated code and SCADE Display-generated code. Only C

- 4 Check **User-provided sensors declaration** to declare sensors using either:
 - redefinition using `#define <sensorCname> <cVar>` in H file set in **User config**, see [“Setting Code Configuration Options”](#) on page 842.
 - variable definition with `<sensor C type> <sensor C name>;` syntax in an external C file configured in **Simulation** properties as [Source File for C](#).
 - variable definition with `<sensorAdaName>: <sensorAdaType>;` syntax in an external ADS file configured in **Simulation** properties as [Source File for Ada](#).

If unchecked, sensors are defined by the Target or Adaptor (except ASAM MCD-2 MC).

- 5 Set options on dedicated tabs displayed after selecting Targets/Adaptors.

Setting Periodicity of Code Execution

Periodicity of code execution may be necessary when generating code for OSEK RTOS, VxWork 653 RTOS, VeriStand, Simulink, FMU, or Twin Builder. Due to incompatibilities with the selected Target or Adaptor, enabling the periodicity of code execution may result in warnings at code generation time.

To set code execution periodicity

- 1 Click **Settings**  in **Code Generator** toolbar, display **Code Integration** content, and select the needed **Target** or **Adaptor**.
- 2 Set the **Periodicity** options to modify the execution of generated code from the drop-down list. Select between the following options:
 - **From root operator** to base code execution on root operator periodicity (default 100 ms). Uneditable **Period**, **Offset**, and **Unit** are values set at root operator level. Set the values by editing the **Code Integration** properties of the operator ([“Editing Operator Properties”](#) on page 444).
 - **Override** to enable periodic code execution and override periodicity specified at root operator level (default 20 ms). Specify the period and/or offset in the corresponding **Period** and **Offset** fields, and set the unit applied to execution cycles in seconds (s), milliseconds (ms), or microseconds (μs) from the **Unit** drop-down list.
 - **No periodicity**: code execution is not periodic. Applicable to RTOS Adaptors. Not applicable to NI VeriStand or Simulink (if error is raised).

RELATED TOPICS

- [RTOS Adaptor Guidelines](#)
- [Gateway Guidelines for VeriStand](#)
- [“Simulating Models in FMI-Compliant Tools”](#) on page 629
- [“Simulating Models in Twin Builder Environment”](#) on page 647
- [“Simulating Models in Simulink Environment”](#) on page 656

Setting Compiler and Build Options

When generating code for targets selected in **Code integration** settings, SCADE Suite generates code based on compiler and build options that may be specific to the environment of the selected target. These options allow to optimize code for specific uses and environments (not applicable to MCG).

- [“To set compiler options”](#)
- [“To set build options”](#)

You may need to adjust compiler and build options in the following cases:

- SCADE Suite Simulation
- SCADE Test Model Coverage and SCADE Test Environment for Host
- Rapid prototyping and co-simulation in VeriStand environment
- Co-simulation in Simulink or FMU-compliant environments
- WCET and stack usage analysis with Timing and Stack Verifiers

To set compiler options

- 1 Click **Settings**  in **Code Generator** toolbar and display **Compiler** after enabling a Target in **Code Integration** settings (see [“Setting Code Integration Capabilities”](#) on page 849).

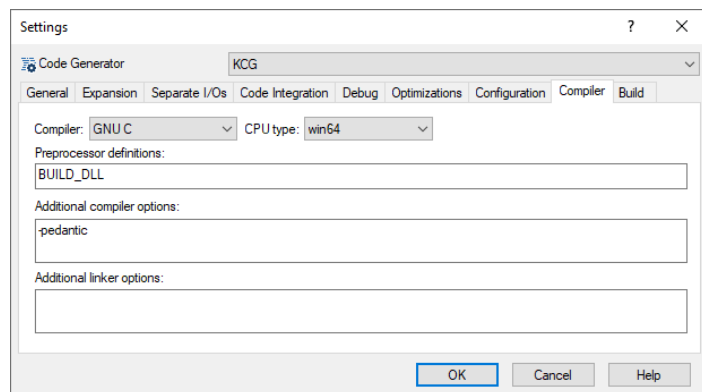


Figure 19.9: Setting compiler options

- 2 Select the compiler to be used during code generation from the **Compiler** drop-down list. The following compilers are available:

- **GNU C** (default): to enable build with mingw.
- **VC<X.n>**: to enable build with Visual C compilers present on the computer where SCADE Suite is installed.

When using SCADE Suite KCG, make sure the Visual C version in use supports C99 code. GNU C and VC12.0 and higher are compatible with KCG C and MCG generated code. Only GNU C is compatible with KCG Ada.

- 3 Specify the CPU supported by the selected compiler from the **CPU type** drop-down list. The CPU Type list changes according to selected Target:
- For **Timing and Stack Verifiers**, set the compiler and CPU type of the cross-compiler in use (see [“Setting Code Generation for Timing and Stack”](#) on page 798).
 - For **Timing and Stack Optimizer**, no Compiler and CPU type is selectable.
 - For **FMU, Mobile, NI VeriStand, or Simulink**, select **win32** or **win64**.
 - For **Simulator, Twin Builder, or Standalone Executable**, **win64** is set.

Note

The compiler and CPU information necessary to build a binary for FMU, NI VeriStand, Simulink, Standalone Executable, Timing and Stack Verifiers, Twin Builder can also be set based on a configuration file as detailed in Appendix D about [“Compiler Configuration for SCADE Suite Code Generation”](#).

- 4 Specify any list of preprocessing definitions to be considered during build of executable in **Preprocessor definitions**. Enter each definition separated by space. For instance: `BUILD_DLL OTHERDEF`.
- 5 Specify other compiler options in **Additional compiler options**. Enter each option separated by space; use double quotes if an option argument contains spaces. For example: `-pedantic -I“Ext Lib\include”`.

- Specify linker options in **Additional linker options**. Enter each option separated by space; use double quotes if an option argument contains spaces. For example: `-L"Ext Lib\bin"`. For details about compiler and linker options, consult the documentation of the compiler in use.

To set build options

- Click **Settings**  in **Code Generator** toolbar and display **Build** after enabling a Target in **Code Integration** settings (see [“Setting Code Integration Capabilities”](#) on page 849).

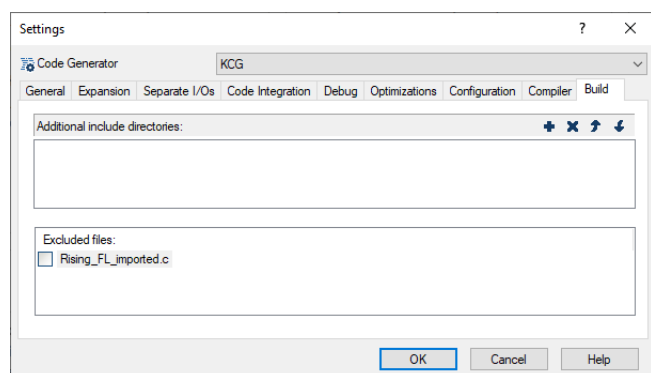


Figure 19.10: Setting build options

- Insert other directories to be considered for the build from the **Additional include directories** list. Use this option when the model contains imported code and if you need to include directories stored in various locations.
- In **Excluded files** list, check each file to be excluded from build. Files displayed in the list are all files declared in the **File View** of the project or present in any project library and specified as [Source File for C](#) or as [Source File for Ada](#) depending on code generator in use.

RELATED TOPICS

- [“Setting Variable Visibility”](#) on page 594
- [“Setting Simulation Binary Generation”](#) on page 589

Adjusting Variable Name Length for Compiler

Depending on C compilers (not for Ada code generation), the maximum size authorized for identifiers may vary. It is possible to modify the maximum length of generated variable names and the name length as needed.

Note

For technical details, consult [“Option -name_length”](#) on page 315 and [“Option -significance_length”](#) on page 322 in *SCADE Suite Technical Manual*.

To adjust name length of variables generated in C code

- 1 Click **Settings**  in **Code Generator** toolbar and display **Configuration**.

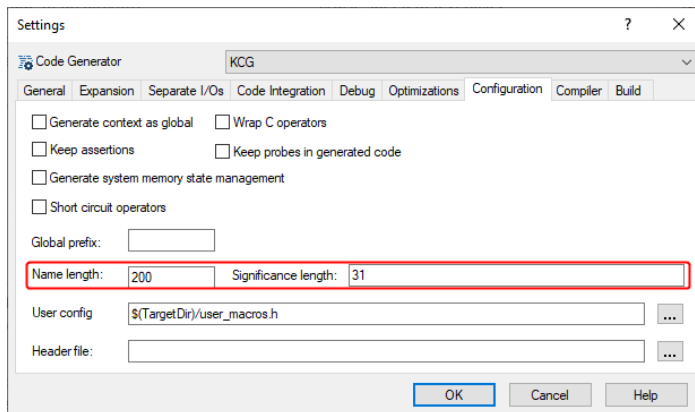


Figure 19.11: Modifying maximum length for variable names for C code generation

- 2 Specify in **Name length** the maximum size of variable names authorized by used C compiler. The value must be a positive integer not lower than 16.
- 3 Specify in **Significance length** the number of initial significant characters to use to help differentiate created C variables. The value must be a positive integer not lower than 16 and not larger than the specified *Name length*.

Setting I/O Separation Options

It is possible to mark operators with three modes of input/output separation. By default, no I/O separation is performed. The modes specify separation as follows:

- All operators are marked for separate I/O
- A list of operators is not marked for separate I/O: the generated cyclic function of selected operators references inputs and outputs respectively in the input and output context of the root operator.
- A list of operators is marked for separate I/O: the generated cyclic function of selected operators is different for inputs and outputs. Outputs are passed as separate parameters. Inputs are no more passed in the input context of the root operator.

Note

For technical details about these modes in command line, consult [“Description of I/O Separation Options”](#) on page 306 in *SCADE Suite Technical Manual*.

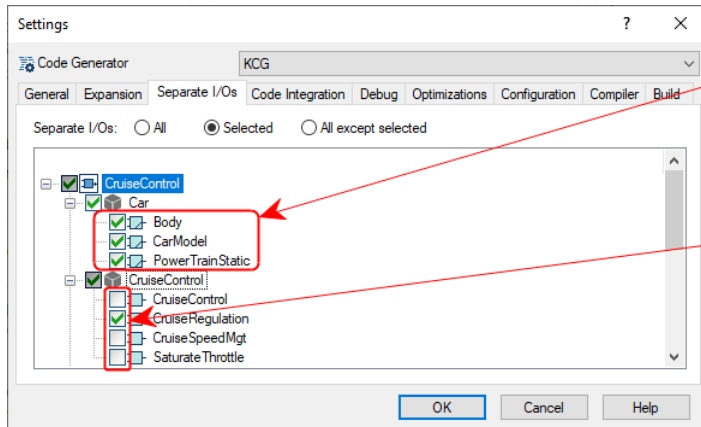
To set I/O separation in generated code

- 1 Click **Settings**  in **Code Generator** toolbar and display **Separate I/Os** to enable the separation mode as follows:
 - Set **All** to mark all operators for separate I/O in the model.

Tips

- Any operator set with `separate_io` pragma is marked as ‘separate I/Os’ or with `no_separate_io` pragma marked as ‘no separate I/Os’ in the trees. About pragma settings, see [“Setting Pragmas on Model Elements”](#) on page 859.
 - To remove all operators unused by the selected root operator from this list, select the [Skip unused model objects](#) option in the General settings.
-

- Set **Selected** to define the list of operators marked for separate I/O. Unchecked operators are identified as operators generated with no I/O separation.



Click on the package name to check all operators and subpackages

Click on each operator to select a subset of operators in packages or subpackages

Figure 19.12: Setting I/O separation on selected operators only

- Set **All except selected** to define the list of operators not marked for separate I/O. Unchecked operators are identified as operators generated with I/O separation.

Tips

- Check a package to check all its content.
- Hold **Ctrl**-key to set separate items (package or operator names) one by one in multiselection and press **Space** bar to check/uncheck the highlighted list.
- Hold **Shift**-key to set consecutive items (package or operator names) in multiselection and press **Space** bar to check/uncheck the highlighted list.
- To check/uncheck by mouse click a multiselection list, keep the **Ctrl** or **Shift** keys pressed while clicking on checkboxes.

Setting Pragmas on Model Elements

It is possible to produce pragma information for further compilation purposes by explicitly attaching pragmas to specific elements in SCADE Suite models. SCADE Suite KCG, AUTOSAR Code Generator (ACG), and Multicore Code Generator (MCG) take into account such information during code generation. Multicore Code Generator also supports MCG-specific pragma information.

Note

Pragmas are managed in XML and text formats. In XML format, they are attached to their including object. In text format, they are associated with respect to tool-specific rules. SCADE Suite Editor manages and interprets user pragmas set in SCADE Suite models, but pragmas from other tools are only loaded and stored without any possible interpretation.

Pragma information can also be included in model documentation as detailed in [“Modifying Report Content”](#) on page 36 in *SCADE LifeCycle User Manual*.

To attach pragma information to declarations of model elements

- 1 Right-click an element declaration in **Scade View** and select *Properties*.
-

Note

Selecting an element from diagrams applies to all instances used in the diagram. Multi-selecting similar elements (i.e., only operators, variables, or constants) allows to edit all pragma information at once for the selected elements.

- 2 Select **KCG Pragmas** on the left and set required properties on the right. Available pragmas depend on the selected code generator as listed from [“Pragmas by Code Generator Version”](#) on page 258 in *SCADE Suite Technical Manual*.

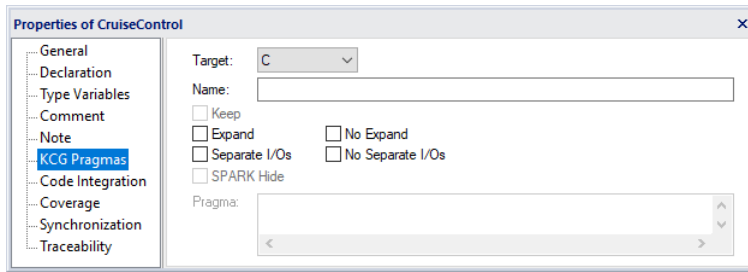


Figure 19.13: Setting pragma on design elements

- 3 Select a target code **C** or **Ada** from **Target** drop-down list to enable available pragmas.
- 4 In the **Name** field, specify the text to be forced by KCG (using pragmas `kcg C : name with KCG C`, `MCG`, and `ACG` or `kcg Ada : name with KCG Ada`) in the code generated for user-defined operators (imported or not), constants, sensors, inputs/outputs, size parameters, signals, local flows, types, enumeration values, structure labels, state machines, states, If/When blocks, and assertions. Name pragma must be a unique and valid C/Ada identifier.

Note

The specified text replaces the name of the element in the code.

- For an enumerate value without pragma, the code is:

```
typedef enum {blue, red, green} Color;
```
 - With the pragma set on the first value as dummy, the code becomes:

```
typedef enum {dummy, red, green} Color;
```
-

- 5 Check/uncheck the displayed pragma according to selected elements. The list of pragmas by model elements is in [Table 19.1](#). About pragma syntax, consult [“Pragma Syntax and Usage”](#) on page 239 in *SCADE Suite Technical Manual*.

Table 19.1: Description of available pragmas

Pragmas	Attached to	Description	Note
Pragma information for C/Ada code generation:			
Default	State or enumeration value	With this pragma, the generated construct for a state or enumeration value handles the state or enum value in the “default” branch (C code) or in the “others” branch (Ada code).	Generated pragma: default
Enum Value	Enumeration value	Forces the enumeration label value in the generated code.	Generated pragma: C : enum_val for KCG C, MCG, ACG Ada : enum_val for KCG Ada
Expand	User-defined operator	Expands the selected operator in the generated code unless the operator is specified in the list of <i>non-expanded operators</i> in code generation settings.	Pragma information is not displayed unless text is editable as in text diagrams, in textual expressions on graphical diagrams, or in property tabs. They do not apply to imported operators. About expansion for code generation, see “Setting Code Expansion Mode” on page 847. Generated pragma: expand or noexpand
No Expand		Does not expand the selected operator in the generated code unless the operator is specified in the list of <i>expanded operators</i> in code generation settings.	
Keep	Variable (inputs/ outputs, hidden inputs, locals, signals, probes)	Forces the variable to be present in generated code.	For ensuring the traceability of local memories. Generated pragma: keep

Table 19.1: Description of available pragmas (Continued)

Pragmas	Attached to	Description	Note
Manifest	User-defined type	Forces the name of the user-defined type in place of the struct_XXX / array_XXX and of the derived names (from kcg_use_<type>, kcg_copy, and kcg_compare_<type>).	No more indirection for manifest type. Generated pragma: <code>manifest</code>
Separate I/Os	User-defined operators	Forces generated code for cyclic function to be different: Outputs are no more in the context but passed as separate parameters. Inputs are no more passed in input context for the root operator.	Generated pragma: <code>separate_io</code>
No Separate I/Os		Forces generated code for cyclic function of operator with attached pragma to declare outputs in output context and inputs in input context for the root operator.	Generated pragma: <code>no_separate_io</code>
Type Initializer	Imported type	In C code: forces to name the C initialization function of the associated C type. In Ada code: forces a reference to a variable or function name whose values are used to set the initial values of the type.	Generated pragma: <code>C:initializer</code> for KCG C <code>Ada:initializer</code> for KCG Ada

Table 19.1: Description of available pragmas (Continued)

Pragmas	Attached to	Description	Note
Pragma information specific to C code generation:			
Const	Imported constant	If selected, constants are declared as const in the generated code. If not, they are declared as macros (#define).	For imported constants as macros, see “Setting Imported Constants with External Code” on page 225. Generated pragma: C : const
Scalar	Imported type	Forces the generated code to consider the imported type as a scalar to avoid pointer operations.	For code optimization purposes. Generated pragma: C : sca lar
Pragma information specific to Ada code generation:			
SPARK Hide	Package and user-defined operator	Propagates SPARK hide directive to counterpart package or subprogram in generated Ada code	Generated pragma: Ada : spark hide
Pragma	User-defined or imported operator, type, constant, sensor, signal, local	Forces pragma declaration on the Ada counter part of the element declaration in the generated code. Each line in Pragma field declares a pragma. Use the “\” character at the end of a line to specify pragma value on multiline: this_is_a_very_long\ value_of_one_pragma\ on_three_lines	Generated pragma: Ada : pragma
Pragma information specific to MCG code generation:			
Task	Any operator instance	For details, see “Specifying Task Pragmas” on page 898	
Unroll	Iterator instance	For details, see “Declaring Iterator Instances as Unrolled” on page 904	

Launching KCG Code Generation from UI

You can launch code generation from SCADE Suite user interface to produce source code out of models.

To launch code generation from user interface

- 1 Launch SCADE Suite and open the project.
- 2 Select the appropriate configuration from the **Code Generator** toolbar. For details, see [“Enabling Configurations”](#) on page 119. The active configuration contains all preset options and parameters for code generation.
- 3 Proceed as follows from **Code Generator** toolbar:
 - Click **Generate**  to produce code (if required only) for the root operator(s) set in the active configuration or for the one(s) selected in **Scade View**, code relative to the [Target/Adaptor](#) set in the active configuration, and Target-related files required for the build.

Note

- Output files are stored where specified in Target Directory of the Code Generator General settings. For instructions, see [“Specifying Code Generation”](#) on page 836.
 - The Generate command produces a `sc2c_integration_files.txt` file providing the list of files necessary for code integration, see [“List of KCG-Generated Files for Module Integration”](#) on page 880.
-
- Click **Build**  to produce all binaries relative to the [Target](#) set in the active configuration and produce code as with **Generate** command above. In case the active configuration references graphical panels (SCADE Display, ARINC 661, or Rapid Prototyper model files), the build of graphical panel binaries is done only if input graphical panels have changed since previous successful build.
 - Click **Rebuild All**  to rebuild everything.
 - Click **Stop**  to interrupt code generation or build once launched.

When launching code generation, the **Generate** command checks if it is necessary to run the selected Code Generator. This is required under one of the following conditions:

- Target directory specified in the active configuration never filled by successful generation (directory empty, partially filled, or does not exist)
- SCADE Suite input model and referenced libraries changed since previous generation
- At least one Code Generator option changed since previous generation
- Output generated code modified since last generation

You may use SCADE Suite KCG in command line to generate code as if launching code generation from SCADE Suite user interface. A dedicated command line format and options are available as described in [“Code Generator Command Line”](#) on page 920.

Note

SCADE Suite models should not contain any of the reserved keywords. See [“Keyword List”](#) on page 5 in the *Scade Language Reference Manual*.

Warning and error messages raised by SCADE Suite KCG and MCG are described in Chapter 12 about [“Code Generation Messages”](#) from *SCADE Suite Technical Manual*.

Note

- All options passed to SCADE Suite KCG or MCG are logged in the `config.txt` file stored in the target directory specified for the project. For using code generation options, see previous sections of this chapter. For details about these options in command line, consult Chapter 11 about [“Code Generators Command Line Options”](#) in *SCADE Suite Technical Manual*.
 - For C code qualification/certification purposes with SCADE Suite KCG, the options set in SCADE Suite to launch SCADE Suite KCG must be verified from the `config.txt` file generated in target directory.
-

Launching KCG Executable in Command Line

You must use SCADE Suite KCG executable directly in command line on SCADE Suite models to produce C code within the framework of certification activities. For details, see Chapter 9 about [“Code Production Using SCADE Suite Code Generators”](#) in *SCADE Suite Technical Manual*.

To start KCG executable from list of options

Use the following command line format and parameters to launch KCG:

```
kcg<version_number> [options] -root <NodeName> <FileName_1> ... <FileName_n>
```

Available command options and parameters:

- `FileName_n` are all files composing the input model with an absolute or relative path. It is **mandatory**. Input files can be in Scade graphical syntax (.xscade) or Scade textual syntax (.scade). At least one file must define the model. For details about model files used by SCADE Suite KCG, consult [“Schema of Input/Output Files”](#) on page 181 in *SCADE Suite Technical Manual*.
- `NodeName` is the name of the root operator or a comma-separated list of `NodeName`. It is **mandatory**.

To start KCG executable using a configuration file

Use the following command line format and parameters to launch KCG:

```
kcg<version_number> -config <config_file>
```

Available command options and parameters:

- `config_file` is the name of the file that contains the command line except `kcg<version number>`. The `config_file` argument may contain a path.

Note

For C code qualification/certification purposes with SCADE Suite KCG, the options set in SCADE Suite Editor have to be verified from the `config.txt` file generated in target directory.

Using KCG command line options bypasses SCADE Suite Code Generator UI. It remains possible to use SCADE Suite Code Generator integrated in SCADE Suite UI with command line as well (see [“Code Generator Command Line”](#) on page 920 in Appendix A about [“SCADE Suite in Command Line”](#)).

The following options allow to run KCG code generation in command line:

KCG (C and Ada)

```
[-h/-help]
[-exp, -noexp, -expall]
[-header <filename>]
[-keep_asserts]
[-name_length <n>]
[-probes]
[-root]
[-separate_io, -no_separate_io, -separate_io_all]
[-short_circuit]
[-target C|Ada]
[-target_dir <dir>]
[-warn_error]
[-no_timestamp] (only KCG 6.6.4)
[-xml_basename]
```

KCG (only C)

```
[-debug, -observe, -no_observe]
[-global_root_context]
[-globals_prefix <prefix>]
[-input_threshold <n>]
[-significance_length <n>]
[-state_vector]
[-static]
[-user_config <filename>.h]
[-wrap_c_ops]
```

KCG (only Ada)

```
[-root_package <AdaPackagePath>]
[-imported_root_package <AdaPackagePath>]
[-use_types <typeslist>]
```

For option descriptions, consult Chapter 11 about [“Code Generators Command Line Options”](#) in *SCADE Suite Technical Manual*. For the description of messages returned by SCADE Suite KCG, consult Chapter 12 about [“Code Generation Messages”](#) in *SCADE Suite Technical Manual*.

If using Multicore Code Generator rather than SCADE Suite KCG, refer to [“Launching MCG Executable in Command Line”](#) on page 905.

20 /Implementing Models with Imported Code

This chapter explains how one can implement imported code in SCADE Suite models. It is possible to introduce in SCADE Suite models user types and constants that are externally defined, operators in host language not suited for SCADE Suite modeling, or any operator definition existing in legacy code. Once imported code is correctly implemented and its source files properly declared, the corresponding imported entities can be used in simulation of SCADE Suite models. The implementation of imported code is facilitated by means of file templates specific for C and Ada code.

- [“Generating Templates for Imported Operators”](#)
- [“Implementing Imported C Code”](#)
- [“Implementing Imported Ada Code”](#)

Note

SCADE Suite KCG generates file templates directly from models.

Generating Templates for Imported Operators

SCADE Suite Code Generator can help you producing files that contain C/Ada code necessary to implement imported operators in SCADE Suite models. When the input model contains imported declarations, SCADE Suite KCG Code Generator and Multicore Code Generator generate file templates.

Note

Never modify KCG generated files for managing imported code implementation. Use these files as templates and keep them as stored by the tool. KCG generated files are overwritten during each code generation session.

To generate file templates for imported operator code

- 1 Declare imported operator as in [“Declaring and Editing Imported Operators”](#) on page 194.
- 2 Define the operator interface as needed in SCADE Suite model. For user instructions, see [“Declaring Operator Interface”](#) on page 162.
- 3 Create a node to instantiate graphically the imported operator(s). The node must be semantically correct (see [“Checking Model Semantics”](#) on page 274).
- 4 Select the created node and click **Generate**  to launch code generation.
- 5 Proceed as follows:
 - For imported functions in C, see [“Implementing C Code for Imported Functions”](#)
 - For imported nodes in C, see [“Implementing C Code for Imported Nodes”](#)

Implementing Imported C Code

There are two ways for implementing imported C code in SCADE Suite environment. You can either rely on the default mapping in dedicated files or define user mapping between external entities declared in SCADE Suite and their implementation in manual code.

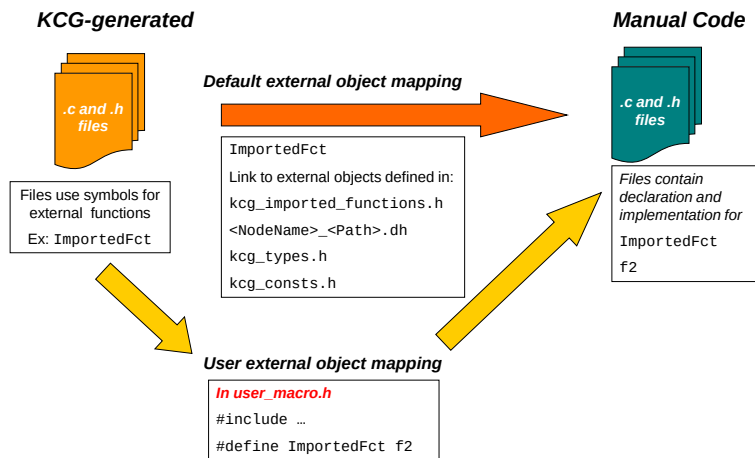


Figure 20.1: Imported C code implementation methods in SCADE Suite

In case you override the default mapping using macros, you give the object mapping and point to the external file that contains its declaration and implementation from a `user_macro.h` file.

By default, this macro file is located in the target directory of the project. You can edit the location of this macro file from the Code Generator Settings as shown below.

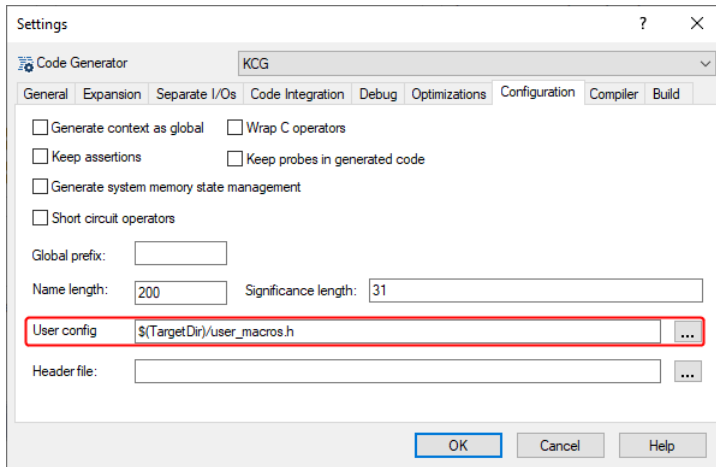


Figure 20.2: Setting user_macro.h file location

The implementation of C code is managed differently depending on imported entities:

- [“Implementing C Code for Imported Functions”](#)
- [“Implementing C Code for Imported Nodes”](#)

Implementing C Code for Imported Functions

The external C code defining imported functions can be implemented with the help of a dedicated file generated by KCG. This file is a template that contains the function prototypes for all imported function operators.

To implement C code for imported functions

- 1 Double click `kcg_imported_functions.h` in the list of generated files to open it.

```

kgc_imported_functions.h x
/* ***** SCADE Suite KGC 32-bit 6.6 (build i8) *****
** Command: kgc66.exe -config C:/Dev Projects/R17 projects/ImportedCodeImplementation/KGC/config.txt
** Generation date: 2015-08-26T13:32:25
*****$ */
#ifndef _KGC_IMPORTED_FUNCTIONS_H_
#define _KGC_IMPORTED_FUNCTIONS_H_

#include "kgc_types.h"

#ifndef ImportedFunction_Package1
/* Package1::ImportedFunction/ */
extern kgc_int32 ImportedFunction_Package1(
/* Package1::ImportedFunction/Input1/ */ kgc_int32 Input1,
/* Package1::ImportedFunction/Input2/ */ kgc_bool Input2,
/* Package1::ImportedFunction/Input3/ */ kgc_bool Input3);
#endif /* ImportedFunction_Package1 */

#endif /* _KGC_IMPORTED_FUNCTIONS_H_ */
/* ***** SCADE Suite KGC 32-bit 6.6 (build i8) *****
** kgc_imported_functions.h
** Generation date: 2015-08-26T13:32:25
*****$ */

```

Figure 20.3: Using kgc_imported_functions.h as file template

- 2 Open the source file of the declared function in SCADE.
- 3 Copy the function signature from the H-file template and paste it in the source file you declared in the function **Properties**.
- 4 Delete the **extern** keyword copied from the source file.
- 5 Define the body of the function after the signature and save the file.

```

My_imported_functions.c x
// DEFINITION OF USER CODE FOR IMPORTED FUNCTION
#include "kgc_types.h"

/* Package1::ImportedFunction/ */
kgc_int32 ImportedFunction_Package1(
/* Package1::ImportedFunction/Input1/ */ kgc_int32 Input1,
/* Package1::ImportedFunction/Input2/ */ kgc_bool Input2,
/* Package1::ImportedFunction/Input3/ */ kgc_bool Input3)
{
////////////////////////////////////
// Body of the user code for imported function
if (Input2 && Input3) {
    return Input1;
}
else {
    return -17;
}
////////////////////////////////////
}

```

Figure 20.4: Completing body of imported function

RELATED TOPICS

- [“Generating Templates for Imported Operators”](#)
- [“Declaring and Editing Imported Operators”](#)

Implementing C Code for Imported Nodes

The external C code defining imported nodes can be implemented with the help of dedicated files (.dh and .dc files) generated by KCG. These files are used as templates that contain context declarations (in .dh files) and definitions of the initialization, reset, and cycle C functions (in .dc files).

To implement C code for imported nodes

- 1 Locate in project directories .dc and .dh files corresponding to imported node(s) and named: `NodeName_<path>`. There is a template file for each imported node.
- 2 Copy `NodeName_<path>.dc` into `NodeName_<path>.c` and `NodeName_<path>.dh` into `NodeName_<path>.h`.
- 3 Open `NodeName_<path>.h`.

```
ImportedNode_Packagel.h X
/* $***** SCADE Suite KCG 32-bit 6.6 proto (build i8) *****
** Command: kcg66.exe -config C:/Users/gm/Desktop/GB/Simulation/config.txt
** Generation date: 2015-08-25T16:18:26
*****$ */
#ifndef _ImportedNode_Packagel_H_
#define _ImportedNode_Packagel_H_

#include "kcg_types.h"

/* ===== no input structure ===== */
/* ===== no output structure ===== */
/* ===== context type ===== */
typedef struct {
    /* ===== outputs ===== */
    kcg_int32 /* Packagel::ImportedNode/Output1/ */ Output1;
    /* ===== insert eventual inits and memories ===== */
    // INSERT A MEMORY HERE:
    kcg_int32 mem;
} outC_ImportedNode_Packagel;

/* ===== node initialization and cycle functions ===== */
/* Packagel::ImportedNode/ */
extern void ImportedNode_Packagel(
    /* Packagel::ImportedNode/Input1/ */ kcg_int32 Input1,
    /* Packagel::ImportedNode/Input2/ */ kcg_bool Input2,
    outC_ImportedNode_Packagel *outC);

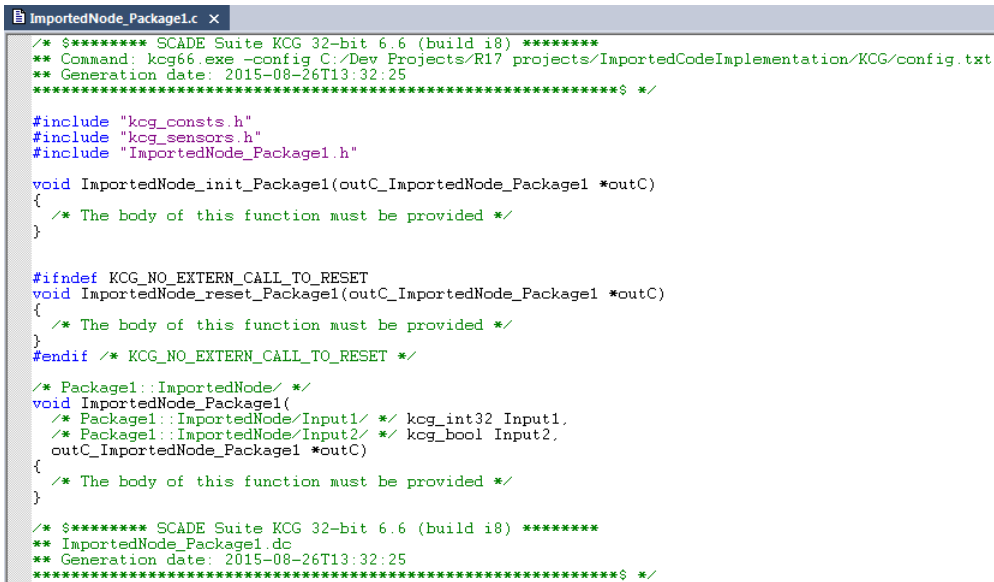
#ifndef KCG_NO_EXTERN_CALL_TO_RESET
extern void ImportedNode_reset_Packagel(outC_ImportedNode_Packagel *outC);
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

extern void ImportedNode_init_Packagel(outC_ImportedNode_Packagel *outC);

#endif /* _ImportedNode_Packagel_H_ */
/* $***** SCADE Suite KCG 32-bit 6.6 proto (build i8) *****
** ImportedNode_Packagel.dh
** Generation date: 2015-08-25T16:18:26
*****$ */
```

Figure 20.5: Completing header template of imported node

- 4 Complete context structure by adding initialization and memory fields, save the file.
- 5 Open `NodeName_<path>.c`.



```

ImportedNode_Packagel.c
/* $***** SCADE Suite KCG 32-bit 6.6 (build i8) *****
** Command: kcg66.exe -config C:/Dev/Projects/RI7/projects/ImportedCodeImplementation/KCG/config.txt
** Generation date: 2015-08-26T13:32:25
*****$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "ImportedNode_Packagel.h"

void ImportedNode_init_Packagel(outC_ImportedNode_Packagel *outC)
{
    /* The body of this function must be provided */
}

#ifndef KCG_NO_EXTERN_CALL_TO_RESET
void ImportedNode_reset_Packagel(outC_ImportedNode_Packagel *outC)
{
    /* The body of this function must be provided */
}
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

/* Packagel::ImportedNode/ */
void ImportedNode_Packagel(
    /* Packagel::ImportedNode/Input1/ */ kcg_int32 Input1,
    /* Packagel::ImportedNode/Input2/ */ kcg_bool Input2,
    outC_ImportedNode_Packagel *outC)
{
    /* The body of this function must be provided */
}

/* $***** SCADE Suite KCG 32-bit 6.6 (build i8) *****
** ImportedNode_Packagel.dc
** Generation date: 2015-08-26T13:32:25
*****$ */

```

Figure 20.6: Completing body template of imported node

- 6 Fill the function bodies of the imported node:
 - The initialization function must initialize all fields of the context definition with default values.
 - The reset function must update the context into its initial state.
 - The cycle function must compute outputs and update the context to make a step.

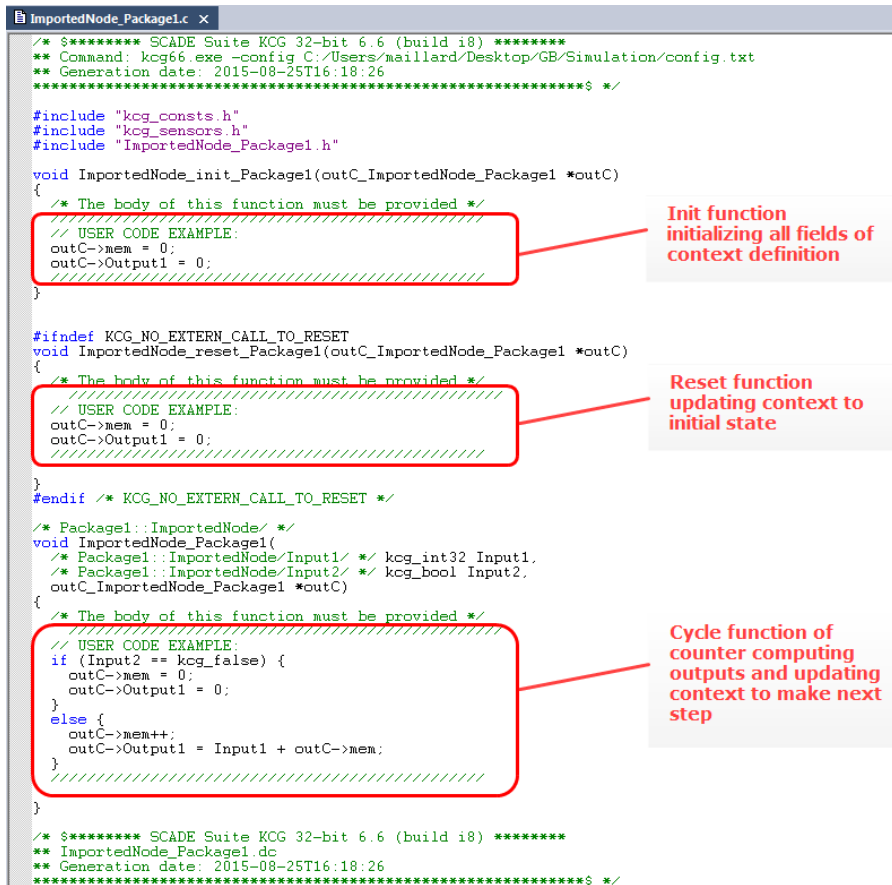


Figure 20.7: Example of function bodies for imported node

- 7 Copy the content of the `NodeName_<path>.c` file into the source file already declared in the imported node **Properties** or declare this `NodeName_<path>.c` file as the new source file of the imported node.
- 8 Save the file.

RELATED TOPICS

- [“Generating Templates for Imported Operators”](#)
- [“Declaring and Editing Imported Operators”](#)

Implementing Imported Ada Code

The external Ada code defining imported entities can be implemented with the help of dedicated files generated by SCADE Suite KCG. When the input model contains imported declarations, KCG generates a template of specification (. dads) and bodies (. dadb) files:

- . dadb template files contain templates for all imported functions and nodes
- . dads template files contain templates for all imported types, constants, and sensors

Starting from these template files, you should create the corresponding . ads and . adb files with the following modifications:

Entities	How to define imported entities
Imported functions	• fill in the cycle procedure body in . adb file
Imported nodes	• replace the empty context in . ads file by a record declaration containing all needed memories for the imported node • fill in the cycle/reset/init procedure bodies in . adb file
Imported types	• replace the type declaration (type T is record end record;) in . ads file by the actual type definition of the imported type
Imported constants	• replace the object declaration (C: T) in . ads file with a constant declaration (C: constant T := <va lue>;)
Imported sensors	• fill in sensor declarations. In Simulation context, there is no need to provide sensor declarations. They are automatically declared by Simulator.

RELATED TOPICS

- [“Declaring and Editing Imported Operators”](#)
- [“Setting Imported Types”](#)
- [“Setting Imported Constants with Imported Types”](#)

21 / Integrating Monolithic Code

This chapter explains how one can integrate SCADE Suite KCG-generated monolithic code in software applications and introduces the ability to co-generate code with SCADE Display and integrate generated code in software applications.

- [“Integrating C/Ada Code in Software Application”](#)
- [“Generating and Integrating SCADE Display Code”](#)

Integrating C/Ada Code in Software Application

Once C or Ada code is generated with SCADE Suite KCG Code Generator from models, you may want to integrate the module in a software application. You must write the main function/procedure to call the generated functions.

- [“List of KCG-Generated Files for Module Integration”](#)
- [“Module Integration with C Code”](#)
- [“Writing Main Function/Procedure”](#)
- [“Module Integration with Ada Code”](#)

Note

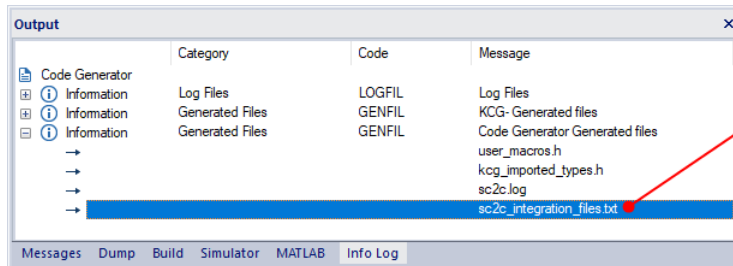
Check *SCADE Suite KCG Tool Operational Requirements* distributed with SCADE Suite KCG Certification Kit for specific considerations about code integration (chapter 18, KCG User).

List of KCG-Generated Files for Module Integration

For module integration purposes, you need to know the exhaustive list of external files necessary to build a binary and all KCG-generated files in their exact location. External files can be any imported C/Ada files, or any imported files from user libraries. To help integration, SCADE Suite provides the **sc2c_integration_files.txt** file listing:

- all KCG-generated code files
- external user files (only if inserted in **File View** with correct file type)
- any include directories

This file is located in the target directory specified for code generation. You can also open it by double click directly from **Info Log** tab in **Output** window.



Double click the file name
to open the file and check
the list of files used for
code generation

Figure 21.1: Checking list of external and KCG files for code integration

Writing Main Function/Procedure

Module integration is not unique but depends on selected code generation options and KCG pragmas set in model:

- generated files vary according to KCG options
- communications between generated code and main application depend on KCG options:
 - **At root operator level:** interfaces are as stable as possible; the main impact depends whether `-global_root_context` option is used.
 - **At sub-operator level:** interfaces are extremely variable depending on the nature of the operators (nodes, functions, functions with a single output) and on the use of KCG options (`-input_threshold`)

For details about KCG options, see Chapter 11 about [“Code Generators Command Line Options”](#) in *SCADE Suite Technical Manual*.

- generated code API may change according to KCG pragmas like name or expand

For details about KCG pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859)

The main cyclic loop is written with respect to the following pattern:

```
call generated initialization function
begin_loop
  wait for an event (usually clock signal)
  treatment of inputs
  call generated cycle function
  treatment of outputs
end_loop
```

The functions available for using the operator context are defined in [“Definition of Operator Context”](#) on page 193 of *SCADE Suite Technical Manual*.

Module Integration with C Code

Module integration in C depends also on the implementation of basic Scade types (`bool`, `char`, `[u]intN`, `floatN`, `size`). Default types are defined in the `kcg_types.h` file generated with the rest of the files. Such definition uses the types defined in `stddef.h`, which is part of the C89 standard library. It is possible to redefine these default types (except `char` type) by providing the definition of each type in a user configuration file. The path to such file is specified as code generation option for KCG.

For more information about providing C counterparts of predefined types and related issues, consult section 18.2.3 in *SCADE Suite KCG Tool Operational Requirements* (TOR).

Note

In industrial projects, the cyclic and reset functions are often called by an external scheduler (not necessarily main function) able to run several tasks in parallel including possible interruptions. In this case, the integration principle remains the same: a call to reset or initialization function and a cyclic call to cyclic function as presented below.

The reset function allows to initialize partially some fields of the contexts. The initialization function allows to initialize all context fields.

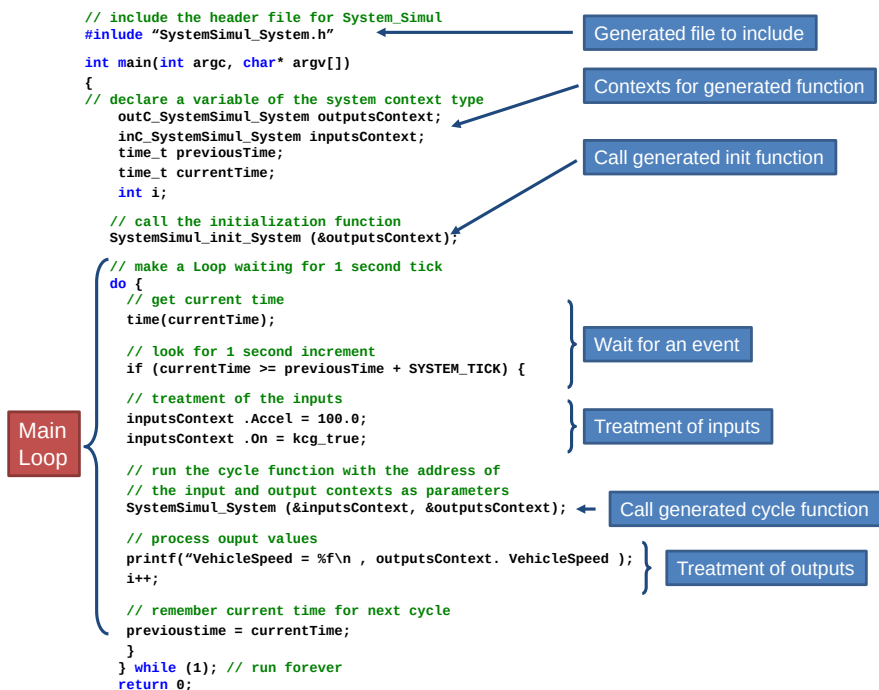


Figure 21.2: Sample of main function file for C code module integration

Module Integration with Ada Code

Module integration in Ada depends also on the implementation of basic Scade types ([u]intN, floatN, char, size) and assertions. You must provide a Kcg_Config package allowing to:

- map Scade basic types (Kcg_[U]IntN, Kcg_FloatN, Kcg_Char, Kcg_Size) with specific Ada types (integer, range, character, ...)
- define Kcg_Assume and Kcg_Guarantee assertions

Note

In industrial projects, the cyclic and reset functions are often called by an external scheduler (not necessarily main function) able to run several tasks in parallel including possible interruptions. In this context, the integration principle remains the same: a call to reset function and a cyclic call to cyclic function as presented below.

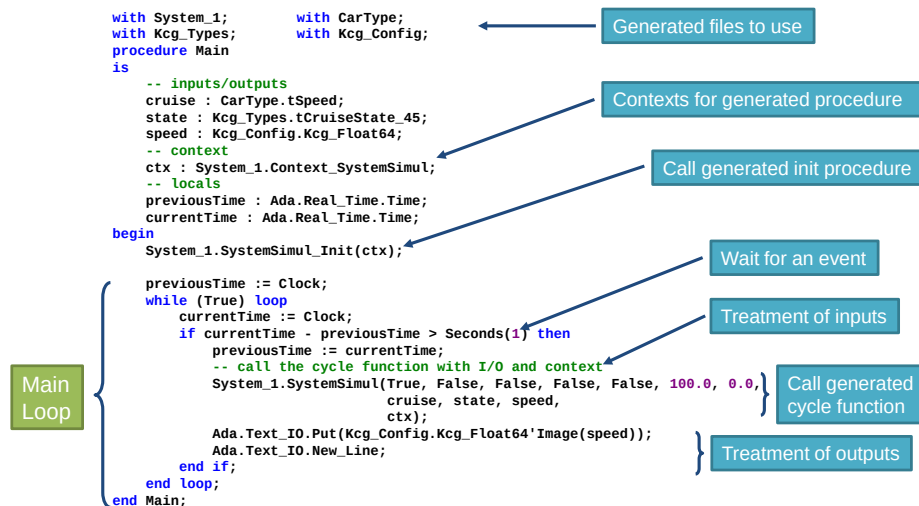


Figure 21.3: Sample of main function file for Ada code module integration

For details about Kcg_Config package, see [“Input/Output Files for Ada Code Generation”](#) on page 182 in *SCADE Suite Technical Manual*.

Generating and Integrating SCADE Display Code

You can generate code for display applications from SCADE Suite provided your project includes one or several SCADE Display graphical panels. The generated code includes both graphical and logical parts, and a wrapping C code. Once generated, the wrapping C code must be integrated with the SCADE Suite/SCADE Display-generated code.

- [“Launching Co-Generation of Display Applications”](#)
- [“Integrating Wrapping C Code with Display Application”](#)

Launching Co-Generation of Display Applications

You can launch code generation of display applications from SCADE Suite projects including SCADE Display graphical panels.

To launch code generation of display application

- 1 In SCADE Suite, click **Settings**  in **Code Generator** toolbar.
- 2 Select KCG configuration from the drop-down list in the upper right corner.
- 3 Display **Code Integration**, set **Adaptor**, and select **SCADE Display Co-generation** in the corresponding drop-down list.

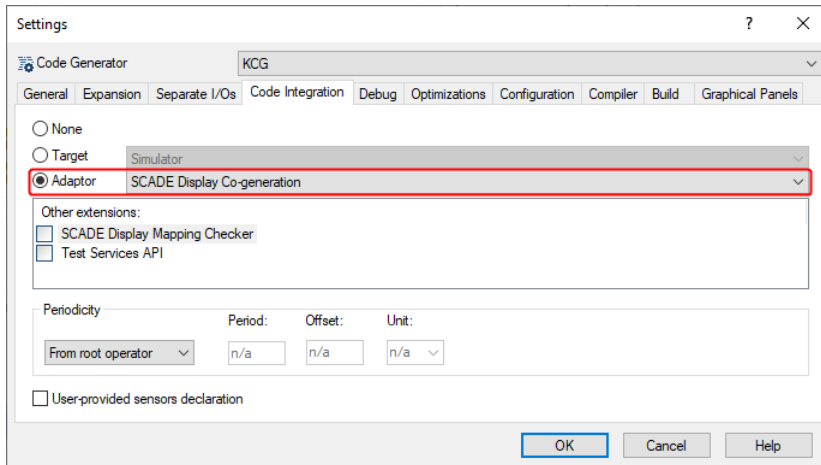


Figure 21.4: Enabling SCADE Display co-generation Adaptor

- 4 Specify the periodicity of the code execution either at root operator level or override root operator periodicity. For details, see [Periodicity](#) option.
- 5 Enable sensor declaration as user-provided (checked) or let the selected Adaptor declare sensors (unchecked). For details, see [User-provided sensors declaration](#).
- 6 Display **Graphical Panels**. For each SCADE Display graphical panel included in the project, select the configuration to use for SCADE Display code generation. Select **None** if you want to exclude the panel from the code generation process.

Note

For information on SCADE Display configurations, see [“Setting Code Generation Configuration”](#) on page 484 in *SCADE Display User Manual*.

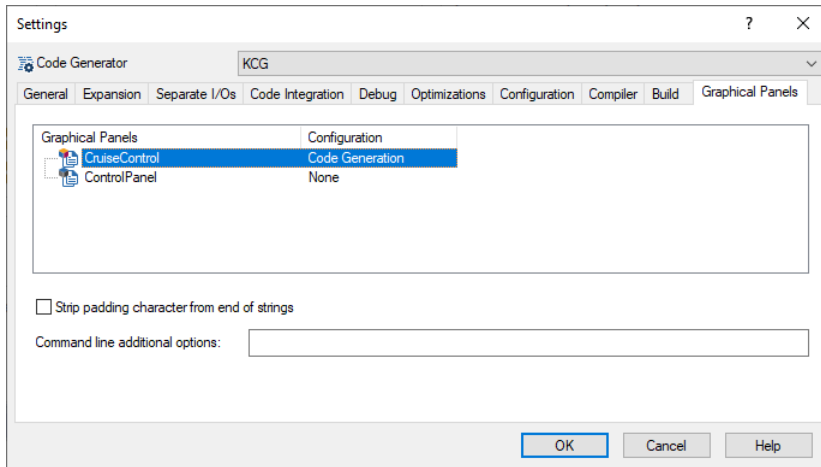


Figure 21.5: Selecting configuration for SCADE Display co-generation

- 7 Check **Strip padding character from end of strings** to prevent copying padding characters added at the end of strings in the generated code.
- 8 In **Code Generator** toolbar, select KCG configuration and click **Generate**  to produce the code specified in the active configuration:
 - the SCADE Suite KCG-generated code for the logical part of the application
 - the SCADE Display KCG-generated code for the graphical part of the application
 - the wrapping C code

Note

Output files are generated where specified in the Target Directory of the Code Generator settings. For instructions, see [“Specifying Code Generation”](#) on page 836.

Integrating Wrapping C Code with Display Application

Once the display application code is generated, you can integrate the wrapping C with the logical and graphical parts of the application. The wrapping C code calls the SCADE Suite/SCADE Display-generated code.

SCADE Suite KCG-generated files include:

- C code for the specified root operator of the active project
- a file named `<SuiteRootNode>_display_adaptor.c` containing the integration code

The integration code defines two functions: `Reset ( )` and `Step ( )`:

- `void Reset(void)` function makes:

```
1. /* Reset SCADE Suite KCG output context */ (<SUITEROOTNODE>_reset(&outputs_ctx);)
2. /* Reset SCADE Display KCG contexts */ (<SDYLAYER>(&symbology_layer_ctx, 0x0);)
3. /* Apply SCADE Suite => SCADE Display mapping = copy SCADE Suite variables into SCADE
Display plugs*/
4. /* Initialize SCADE Display drawings */ (<SDYLAYER>_draw_init();)
```

- `void Step(void)` function makes:

```
1. /* Perform SCADE Suite cycle computation */(<SUITEROOTNODE>(&outputs_ctx);)
2. /* Apply SCADE Suite => SCADE Display mapping = copy SCADE Suite variables into SCADE
Display plugs*/
3. /* Perform SCADE Display drawings */ (<SDYLAYER>_draw(&symbology_layer_ctx, 0x0);)
```

You must adapt the integration code to match project needs by calling:

- the `Reset ( )` function during the application Initialization time
- the `Step ( )` function during the application Run-time

22 /Generating and Integrating Task-Based Code

This chapter introduces task-based code generation and describes how to use SCADE Suite models with Multicore Code Generator (MCG) that generates code from any legacy Scade application exploiting the intrinsic parallelism of Scade programs.

The generated C code makes no hypothesis on the target architecture. The goal is to allow users to take their model and generate a code structured to exploit the computational resources of their target. An integration step is performed to instantiate OS-specific services like communication, synchronization, or memory protection.

The chapter explains how to annotate SCADE Suite models to express potential parallelism in generated code and describes integration activities on multicore targets.

- [“Task-Based Code Generation Principles”](#)
- [“Annotating Models for MCG Code Generation”](#)
- [“Setting Task-Based Code Generation”](#)
- [“Launching Task-Based Code Generation”](#)
- [“Computing Task WCET on MCG-Generated Code”](#)
- [“Implementing Code for Multicore Targets”](#)

Task-Based Code Generation Principles

SCADE Suite code generation for multicore targets relies on annotation activities at model level, specific code structure, and implementation activities for further integration of generated code.

- [“Model Annotation for Task Identification”](#)
- [“Task Allocation to Computing Resources”](#)
- [“Task-Based Generated Code”](#)
- [“Task-Based Code Generation Workflow”](#)

Model Annotation for Task Identification

The operator instances that need to be structured as separate tasks in MCG-generated code must be annotated by a dedicated task pragma (C : task) in SCADE Suite models. Such pragma does not impact the semantics of the model, it only changes the structure of the generated code. When a task pragma is attached to an operator instance, the call to the cycle function for this operator is replaced by channel operations to allow executing this instance separately, for instance in another thread.

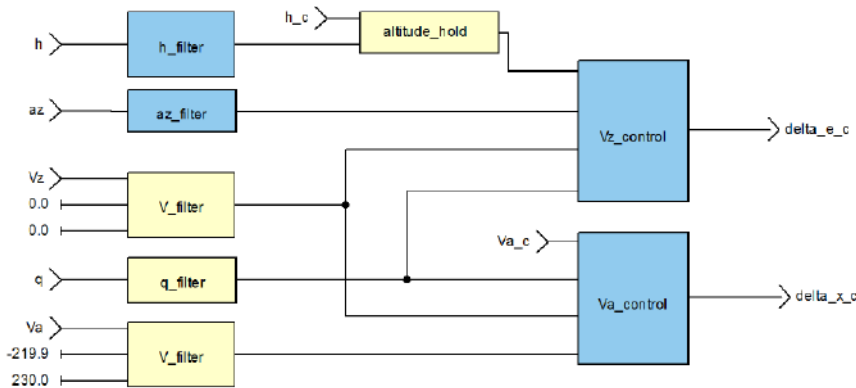


Figure 22.1: Operator instances identified as tasks with pragma information (in blue)

Tasks can be extracted from any instance in the model. It is possible to extract instances inside an activate block or a state machine. Tasks are not restricted to instances in the root operator, but can also be nested: a task can be extracted from an instance inside an operator which is itself a task. Yet, there are some restrictions to consider when using pragmas on instances as detailed in [“Identifying Operator Instances as Separate Tasks”](#) on page 899.

Task-Based Generated Code

When tasks are generated, the code generated by Multicore Code Generator (MCG) is a set of tasks communicating through one-to-one channels corresponding to a Kahn Process Network ([KPN](#)). The code is no longer a single cycle function. One task, namely the root task, executes the root operator of the SCADE Suite model¹. For each operator instance annotated in the model, one task is generated. This task receives data on an input channel, calls the operator, and sends the result on an output channel. In the code generated by SCADE Suite KCG or if there is no task pragma, there is only one task and no channels.

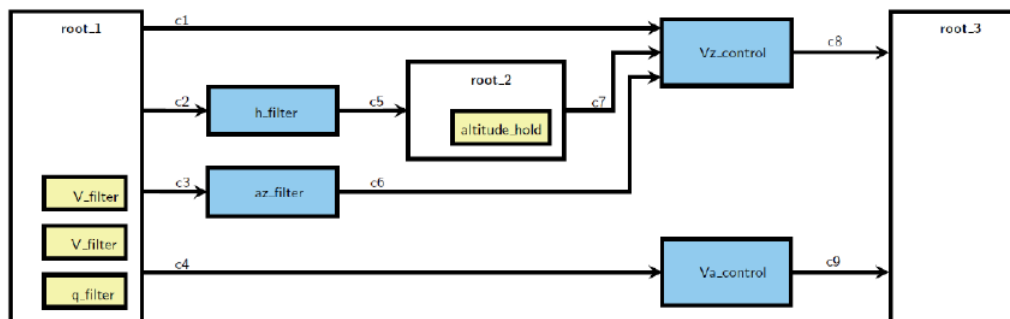


Figure 22.2: Generated KPN from SCADE Suite model with instances identified as tasks

1. In case of multiple root operators, one KPN is generated for each root operator.

Each task has a context, containing its memories, outputs, other elements like probes, and several **methods**: *reset* and *init* methods as well as one or several *cycle* method(s). With MCG-generated code, the cycle method can be split into several methods. The methods of the same task communicate directly through the context. Each cycle method has input and output channels to communicate with other tasks. This information is available in the mapping file. Other operators called by the tasks have the same structure and can also have multiple cycle methods. In MCG-generated code, only the first cycle method reads the inputs of the operator. The outputs can only be read after all cycle methods are executed.

The objective of introducing several cycle methods is to reject communication out of the method body and to preserve the possible schedules in order not to interfere with the efficiency of the multi-core integration. Consider both scheduling examples below where the root operator is split so that the most efficient scheduling is possible.

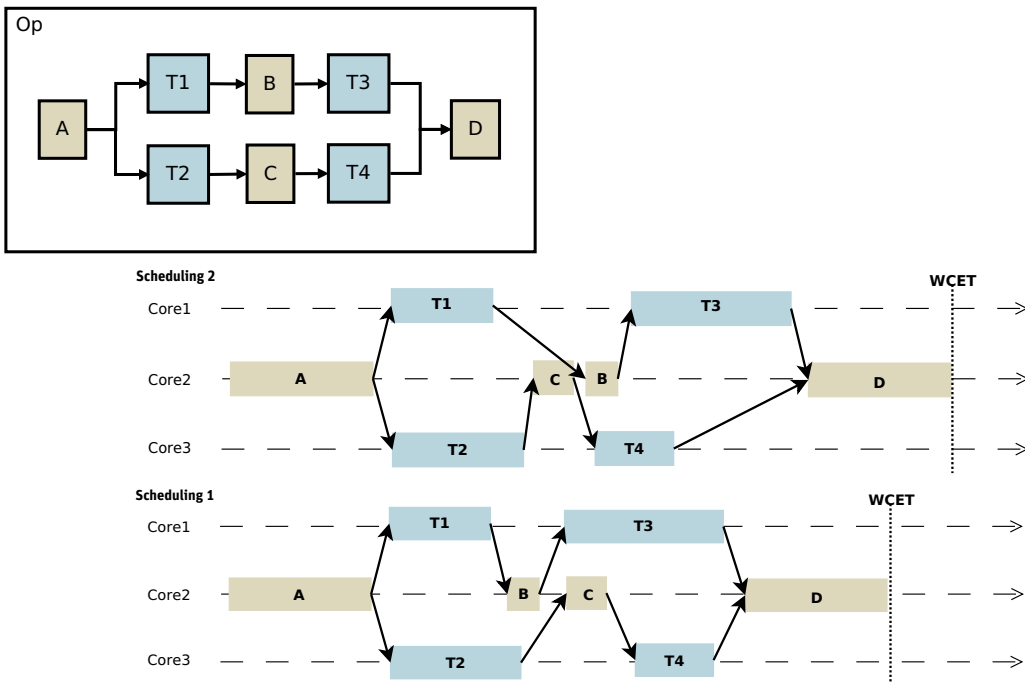


Figure 22.3: Optimizing scheduling out of root operator code splitting

A channel is a FIFO of size one with one sender and one receiver. A structure type is generated for each channel, which represents the data transmitted on the channel. The channel sender writes to (resp. the receiver reads from) a field in its context, which is a pointer to the structure type of the channel. The actual implementation of the channel must be done by users in the integration code.

EXAMPLE OF GENERATED CODE

Consider the following example with one task:

```
function F1(i1: int32) returns (o1: int32)
  o1 = i1*i1;

function F2(i2: int32) returns (o2: int32)
  o2 = (i2-1)*(i2+1);

function root (i: int32) returns (o: int32)
  o = #pragma kcg C:task F1_task #end F1(i)
  - #pragma kcg C:task F2_task #end F2(i);
```

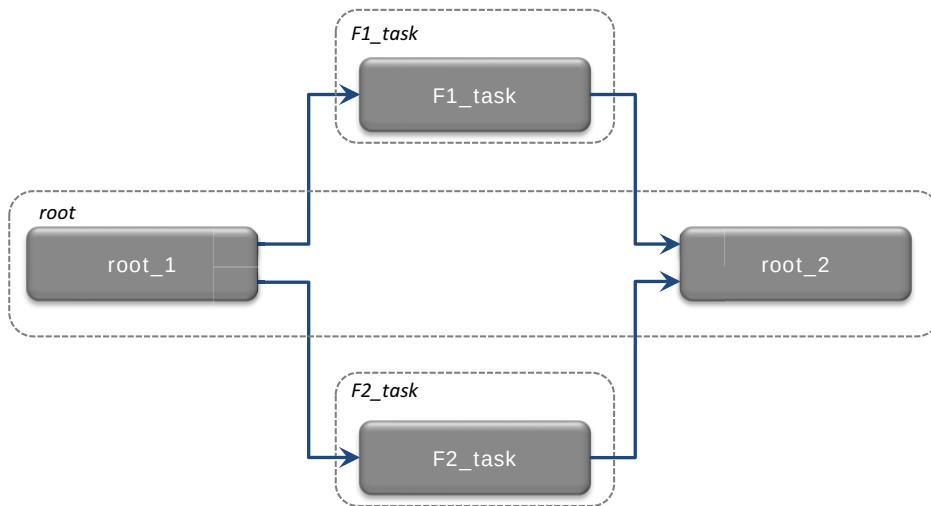


Figure 22.4: KPN generated for the root example

The following code is generated for F1_task:

```
/* channel types */
typedef struct kcg_tag_F1_task_in_ch_type {
    kcg_int32 i1;
} F1_task_in_ch_type ;
typedef struct kcg_tag_F1_task_out_ch_type {
    kcg_int32 o1;
} F1_task_out_ch_type;

/* context type */
typedef struct {
    /* ----- no memorised outputs ----- */
    /* ----- no local probes ----- */
    /* ----- no local memory ----- */
    /* ----- channels ----- */
    F1_task_in_ch_type * /* o=(F1)/ */ F1_task_in_ch;
    F1_task_out_ch_type * /* o=(F1)/ */ F1_task_out_ch;
    /* ----- no sub nodes ' contexts ----- */
    /* ----- no clocks of observable data ----- */
} outC_F1_task;

/* cycle function */
void F1_task (outC_F1_task *outC)
{
    (*outC -> F1_task_out_ch). o1 = /* o=(F1)/ */ F1 ((*outC -> F1_task_in_ch). i1);
}
```

The body of F1_task calls the F1 function on the contents of the input channel F1_task_in_ch and writes the results to the context field associated with F1_task_out_ch.

The code generated for root is similar, but its cycle function is split in two:

```
void root_1 (inC_root *inC , outC_root * outC)
{
    (* outC -> F2_task_in_ch ). i2 = inC ->i;
    (* outC -> F1_task_in_ch ). i1 = inC ->i;
}
void root_2 (outC_root * outC)
{
    outC ->o = (*outC -> F1_task_out_ch). o1 - (*outC -> F2_task_out_ch). o2;
}
```

The method root_1 sends on F2_task_in_ch and F1_task_in_ch, while the method root_2 receives on F1_task_out_ch and F2_task_out_ch.

Task Allocation to Computing Resources

After generating code with MCG, one of the first steps for implementing multicore integration code is to allocate generated tasks to computing resources by means of *workers*. A *worker* represents a set of tasks executed sequentially in the same core/thread/OS task depending on the target. The calling order of methods in a worker must take into account dependencies between methods:

- Data-flow dependencies coming from channels: the receiver of a channel must be executed after the sender.
- Additional dependencies between methods of the same task: this happens when a method reads a context field written by another method. Note there may be several possible orders for the methods of a given task.

These dependencies are provided with the traceability information generated by MCG in the mapping file.

Finding an allocation of the generated tasks to workers is not specific to MCG. This is a traditional problem in implementing KPNs. A static scheduling typically requires to compute the Worst Case Execution Time (WCET) of each method in order to balance the load between the different cores. The specific aspect with MCG is the need to take into account the additional dependencies between methods of the same task and the constraint to put them in the same worker.

Task-Based Code Generation Workflow

The workflow for multicore code generation consists of various user activities you can perform at model level before generating code and integrating it on target environment.

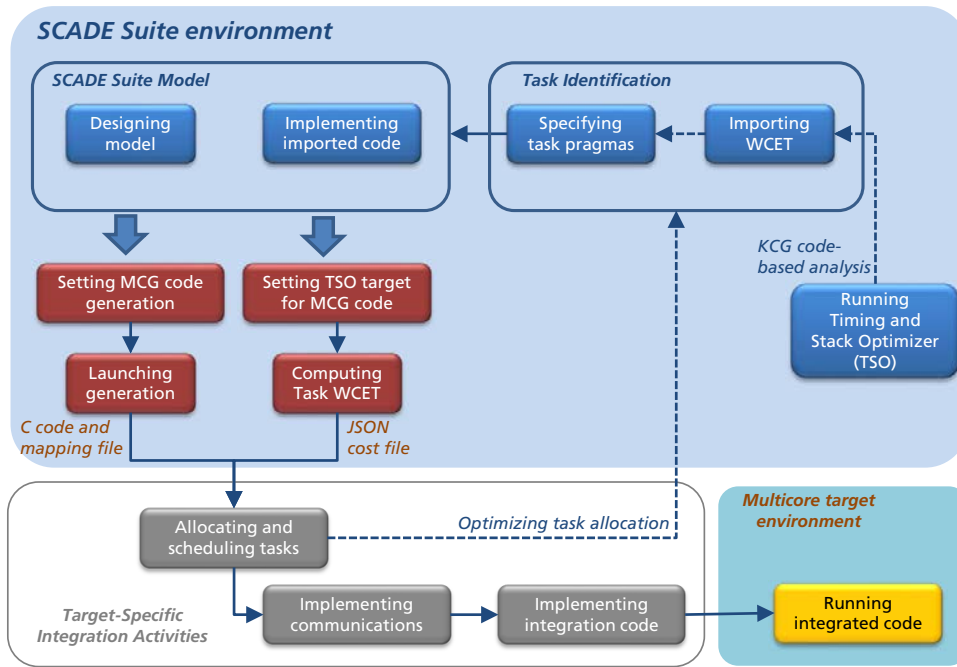


Figure 22.5: Task-based code generation workflow

The following user tasks are available for multicore code generation purposes:

- 1 Identifying tasks and setting other pragmas at model level: [“Specifying Task Pragmas”](#) on page 898 and [“Declaring Iterator Instances as Unrolled”](#) on page 904. Notice SCADE Suite models annotated for task-based code generation can contain imported C code (see Chapter 20 about [“Implementing Models with Imported Code”](#)).

- 2 Setting MCG generation options: [“Setting Task-Based Code Generation”](#) on page 905
- 3 Launching code generation: [“Launching MCG from User Interface”](#) on page 905 or [“Launching MCG Executable in Command Line”](#) on page 905.
- 4 Computing task weight to optimize task allocation: [“Computing Task WCET on MCG-Generated Code”](#) on page 907.
- 5 Allocating/scheduling tasks, implementing communications, and implementing integration code for target-specific needs: [“Implementing Code for Multicore Targets”](#) on page 910.

Annotating Models for MCG Code Generation

SCADE Suite models must be annotated with pragma information to identify explicitly which operator instances in the model are to be considered as separate tasks when generating code for multicore target integration.

- [“Specifying Task Pragmas”](#)
- [“Declaring Iterator Instances as Unrolled”](#)

Specifying Task Pragmas

The use of task pragmas is exclusively dedicated to task-based code generation. It is necessary for identifying operator instance to be generated as separate tasks.

- [“Identifying Operator Instances as Separate Tasks”](#)
- [“Specifying WCET Information from TSO Analysis on KCG-Generated Code”](#)
- [“Displaying Task Definition in Models”](#)

Identifying Operator Instances as Separate Tasks

You identify operator instances to be extracted as separate tasks during code generation by attaching a `C : task` pragma to each operator instance in the model. About pragma syntax, consult [“MCG-Specific Pragmas”](#) on page 256 in *SCADE Suite Technical Manual*.

The pragma can be defined on any operator instance in SCADE Suite models like

- instances of user-defined or imported operators
- instances inside an activate block or a state machine
- instances nested within another operator instance also defined as a task
- iterated instances (n tasks are generated for an iteration of size n)

It is not necessary to modify SCADE Suite models for defining task pragmas: they are saved in a dedicated `<ProjectName>.psc` file under the project directory. This allows adding task pragmas without having to modify the content of a model.

RESTRICTIONS ON PRAGMA USAGE

The usage of task pragma on instances has some restrictions in the following cases:

- The [Local variables as static](#) option (`-static`) is not compatible with tasks. Generating static local variables is not compatible with multicore execution.
- An instance with task pragma cannot have clocked inputs or outputs, for the same reasons that a non-expanded operator cannot have clocked inputs/outputs.

- It is not possible to extract tasks from an instance under a fold-like or partial iterator (either directly iterated or called inside an iterated operator). Only tasks from instances under unrolled **map** and **map1** iterators can be extracted.

```
-- Allowed
(#pragma kcg unroll #end #pragma kcg task F_task #end F <n> >(...))

-- Not allowed
function G (..) returns (..)
let
  = .. #pragma kcg task F_task #end F (..)
tel

...
(#pragma kcg unroll #end map G <n> >(...))
```

- An operator that contains instances with task pragma cannot be multi-instantiated unless it is expanded. This applies if the operator directly calls a task, or indirectly by calling another operator that calls a task. For instance, the following pragmas are rejected:

```
-- Case of a multi-instantiated task caller

function F(i: int32 ) returns (o: int32 )
  o = #pragma kcg C: task F1_task #end F1(i)
  + #pragma kcg C: task F2_task #end F2(i);
function root1 () returns (o: int32 )
  o = #F_inst1 F(1) + #F_inst2 F (0);

-- Case of a multi-instantiated indirect task caller

function G(i: int32 ) returns (o: int32 )
  o = F(i);
function root2 () returns (o: int32 )
  o = #G_inst1 G(1) + #G_inst2 G (0);
```

This can be solved by expanding all task callers that are multi-instantiated: F in the first example; F and G in the second one.

To attach task pragma information to operator instances

- 1 Right-click a user-defined operator instance in diagrams or select an operator instance in **Instance Graph** and select *Properties*.

- 2 Select **Task** on the left and set required properties on the right.

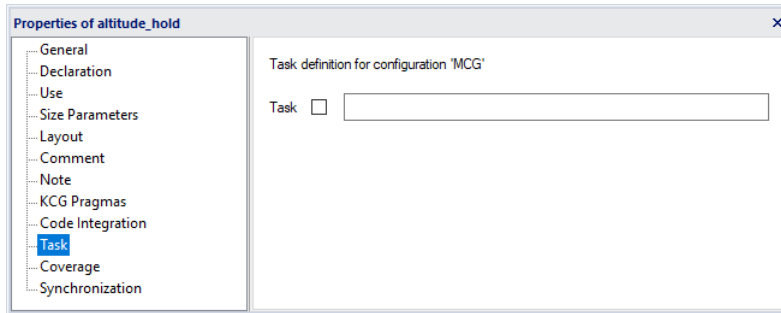


Figure 22.6: Setting task pragma on operator instances

- 3 Check **Task** to attach the task pragma to the selected instance. When the pragma is set, MCG generates the code of the instance in a dedicated C task. The mandatory value associated with the pragma is the name of the C task in the generated code. The pragma is associated with the current configuration indicated on the property page. Change the configuration by selecting a new one from **Code Generator** toolbar and set the pragma for the new configuration.

Specifying WCET Information from TSO Analysis on KCG-Generated Code

The WCET information obtained from Timing and Stack Optimizer analysis of the model can be used for code profiling by evaluating the weight of each operator. Such profiling based on WCET is a help for identifying operator instances as separate tasks and structuring generated code (see [“Task-Based Generated Code”](#) on page 891). The WCET information can be specified manually from the properties of each operator or by importing timing analysis results automatically into WCET properties.

To set WCET information at operator level

- 1 Right-click an operator in **Scade View** or an operator instance in diagrams and select *Properties*.
- 2 Select **Code Integration** and set required properties on the right.

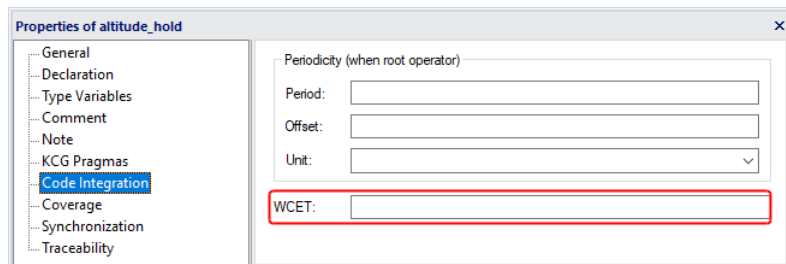


Figure 22.7: Setting WCET information to allow weighing operators

- 3 Enter the information in **WCET** and save.

To import WCET values from Timing analysis

- 1 Run Timing analysis with Timing and Stack Optimizer (TSO) analysis tool on the model to obtain WCET results. For detail about TSO settings and launch, see Chapter 17 about [“Verifying Timing and Stack Usage”](#).
- 2 Select *Project > Code Generator > Import WCET from Analysis Tool* to open the **Select Result** window.
- 3 Select a file containing WCET results in the window and click **OK**. All results are automatically applied to operator properties in the current model. Importing results over previous imports completes WCET information for operators with no information and does not change operator properties having WCET information already filled.

Displaying Task Definition in Models

A dedicated view facilitates the identification of tasks in the models.

To display task definition in models

- 1 Select *View > Instance Graph* to open **Instance Graph View** and identify instances defined as separate tasks from the call graph of the model. The content of this view is reflecting task definitions associated with the current configuration.
- 2 Expand the call graph and check where tasks are defined.
 - Operators identified as separate task for MCG are highlighted in bold
 - WCET information is displayed if specified for the operator (see [“Specifying WCET Information from TSO Analysis on KCG-Generated Code”](#) on page 901)

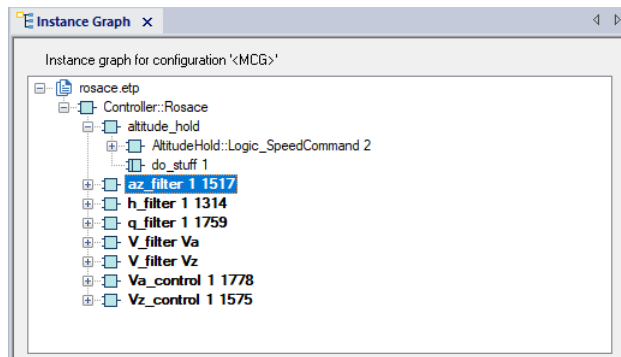


Figure 22.8: Displaying task identification in models from dedicated view

- 3 Display the definition of tasks associated with each configuration of the project by changing the active configuration in the **Code Generator** toolbar.

Declaring Iterator Instances as Unrolled

To generate tasks from an iterator, you need both an `unroll` pragma on the iterator instance and a `task` pragma on the operator instance. From an iteration of size n , MCG code generation produces n tasks. Notice that the `unroll` pragma can also be used separately for other purposes than task generation: it can replace a 'for' loop in the generated code with n calls (which can be more efficient for a small n). About pragma syntax, consult [“MCG-Specific Pragmas”](#) on page 256 in *SCADE Suite Technical Manual*.

To attach unroll pragma information to iterator instances

- 1 Right-click an iterator instance in diagrams and select *Properties*.
- 2 Select **KCG Pragmas** on the left and set required properties on the right.

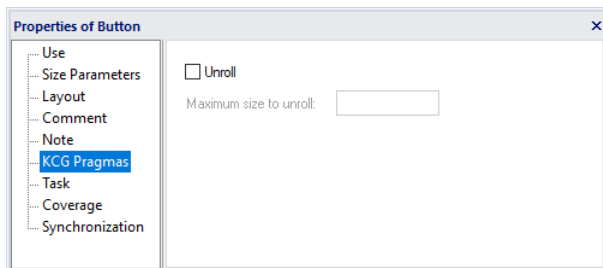


Figure 22.9: Setting unroll pragma on iterator instances

- 3 Check **Unroll** and specify optionally in **Maximum size to unroll** an integer value. If using unroll pragma with task pragma, do not set any value. If a value is set to replace 'for' loops, the unroll pragma applies to MCG code generation only if the size of the iterator is lower or equal to the value.

Multicore Code Generator also supports the same pragmas as SCADE Suite KCG 6.6.2 for C code generation. For setting other non MCG-specific pragmas, see [“Setting Pragmas on Model Elements”](#) on page 859. For syntax of non MCG-specific pragmas, see [“Pragma Syntax and Usage”](#) on page 239 in *SCADE Suite Technical Manual*.

Setting Task-Based Code Generation

Multicore Code Generator supports the same generation settings as SCADE Suite KCG 6.6.2 for C code generation. For details about options, see Chapter 19 about [“Setting and Launching Code Generation”](#)

Launching Task-Based Code Generation

You can launch code generation from the user interface or in command line.

- [“Launching MCG from User Interface”](#)
- [“Launching MCG Executable in Command Line”](#)

Most of the files generated by Multicore Code Generator are similar to the files generated by SCADE Suite KCG. For details about generated files, see Chapter 9 about [“Code Production Using SCADE Suite Code Generators”](#) in *SCADE Suite Technical Manual*.

All messages raised during code generation execution whether from user interface or in command line are available from Chapter 12 about [“Code Generation Messages”](#) in *SCADE Suite Technical Manual*.

Launching MCG from User Interface

You can launch task-based code generation with Multicore Code Generator (MCG) exactly as SCADE Suite KCG from SCADE Suite user interface. For details, see [“Launching KCG Code Generation from UI”](#) on page 864.

Launching MCG Executable in Command Line

MCG executable can be started by command lines taking as input a list of options or a configuration file. Such command lines can contain the same options applicable to SCADE Suite KCG 6.6.2 for C code generation. For details, see Chapter 9 about [“Code Production Using SCADE Suite Code Generators”](#) in *SCADE Suite Technical Manual*.

To start MCG executable from list of options

Use the following command line format and parameters to launch MCG:

```
mcb [options] -root <NodeName> <FileName_1> ... <FileName_n> (MCG)
```

- **NodeName** is the fully-qualified path of the root operator or a comma-separated list of **NodeName**. It is **mandatory**.
- **FileName_n** are all files composing the input model with an absolute or relative path. The files may be in Scade 6 graphical syntax or in Scade textual syntax. At least one file must define the model. For details about model files used by MCG, consult [“Schema of Input/Output Files”](#) on page 181 in *SCADE Suite Technical Manual*. Pragma files (.pragmas) must also be specified in the command line.

To start MCG executable using a configuration file

Use the following command line format and parameters to launch MCG:

```
mcb -config <config_file_name>
```

- **config_file_name** is the name of the file that contains the command line on a single line. The file can be specified by an absolute or relative path. MCG behaves as if the content of **<config_file_name>** is directly written in the command line.

Using MCG command line options bypasses SCADE Suite Code Generator UI. It remains possible to use SCADE Suite Code Generator UI in command line as well (see [“Code Generator Command Line”](#) on page 920 in Appendix A about [“SCADE Suite in Command Line”](#)). The following options are available to run MCG code generation in command line:

MCG (C)	
[-h/-help]	[-separate_io, -no_separate_io, -separate_io_all]
[-exp, -noexp, -expall]	[-significance_length <n>]
[-debug, -observe, -no_observe]	[-short_circuit]
[-global_root_context]	[-state_vector]
[-globals_prefix <prefix>]	[-static]
[-header <filename>]	[-target <Language>]
[-input_threshold <n>]	[-target_dir <dir>]
[-keep_asserts]	[-user_config <filename>.h]
[-name_length <n>]	[-wrap_c_ops]
[-probes]	[-warn_error]
-root	[-xml_basename]

For option descriptions, see Chapter 11 about [“Code Generators Command Line Options”](#) in *SCADE Suite Technical Manual*.

Computing Task WCET on MCG-Generated Code

To optimize task allocation, users can associate a weight with each task and take it into account while allocating tasks and cores. Based on MCG-generated code, it is possible to weight each task by estimating the WCET using **Timing and Stack Optimizer** (TSO). This analysis refines the profiling performed with WCET at operator level by providing the WCET of each generated task at the level of their cycle, reset, and init functions. This WCET information at task level can later be used for scheduling tasks on multicore targets (see [“Allocating and Scheduling Generated Tasks”](#) on page 910).

To compute task WCET on MCG-generated code

- 1 Click **Settings**  in **Code Generator** toolbar to display **General** settings.
- 2 Set **Multicore Code Generator** from **Code Generator** drop-down list or select a configuration defined for MCG code generation.
- 3 Display **Code Integration**, set **Target** to enable **Timing and Stack Optimizer**.

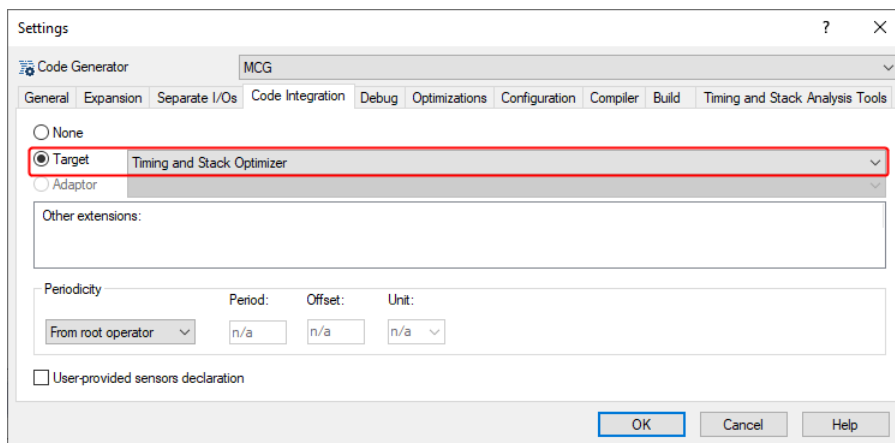


Figure 22.10: Enabling Timing and Stack Optimizer for MCG

- 4 Display **Timing and Stack Analysis Tools** to set code generation options that influence TSO computation. For details, see [option descriptions](#). Click **OK** to save.
- 5 Click **Build**  in **Code Generator** toolbar to produce all file resources (.c and .ais files) necessary for launching TSO analysis. For setting Timing and Stack analysis, see [“Setting Timing and Stack Analysis Tools”](#) on page 803 and [“Setting Advanced Timing and Stack Analysis”](#) on page 805.
- 6 Click **Analyze Timing**  in **Timing and Stack Analysis Tools** toolbar to launch analysis on generated files. Notice to select in the toolbar the same configuration as the one used for code generation.
- 7 After analysis completion, a temporary report displays automatically with all WCET results by cycle, init, and reset functions for each task. It also provides WCET results for instantiated operator contribution given as dependencies functions.

Timing Optimizer Analysis Result

MCG_30 Session

Date: 02/09/2021 16:27:26
 Project: D:\Dev Projects\2020 R1 Projects\Multicore TSO\rosace\rosace.etp
 Configuration: MCG
 Analyzer: Optimizer
 Max unroll: 2
 Default unroll: 2
 KCG version: MCG
 Keep probes: false
 Debug: false
 Significant length: 31
 Name length: 200
 Root inputs and outputs as global variables: false

Result Overview

Cycle functions

SCADE Critical Path #	Calls	WCET (sum)	WCET (max)	WCET (avg)	CWCET (sum)	CWCET (max)	CWCET (avg)
q_filter_1	1	559	559	559.00	1446	1446	1446.00
h_filter_1	1	435	435	435.00	1236	1236	1236.00
az_filter_1	1	441	441	441.00	1370	1370	1370.00
Vz_control_1	1	517	517	517.00	1338	1338	1338.00
Va_control_1	1	660	660	660.00	1712	1712	1712.00
V_filter_Vz	1	606	606	606.00	1608	1608	1608.00
V_filter_Va	1	525	525	525.00	1505	1505	1505.00
Rosace_3_Controller	1	391	391	391.00	391	391	391.00
Rosace_2_Controller	1	501	501	501.00	2540	2540	2540.00
Rosace_1_Controller	1	724	724	724.00	724	724	724.00

Figure 22.11: Timing analysis report on MCG code

A JSON file is also generated containing all task and WCET information for further use in task allocation and scheduling tool: the `<TSOSessionName>_cost.json` file is stored in the same directory as the project file. It can be used as an input of the API functions of the Multicore Code Integration Toolbox. For details and access to API documentation, see [“Multicore Code Integration API”](#) on page 72 in *SCADE Python API Guide*.

Implementing Code for Multicore Targets

The code generated by Multicore Code Generator contains tasks that communicate with other tasks through input/output channels. All communication information and mapping between code and input model are described in the traceability information produced by MCG. Target-specific integration activities consist mainly in allocating generated tasks, implementing communication channels, and writing the integration code.

- [“Allocating and Scheduling Generated Tasks”](#)
- [“Implementing Integration Code”](#)

Allocating and Scheduling Generated Tasks

Target-specific integration requires users to allocate and schedule the generated tasks. For developing critical software, it is essential that users keep full control on the scheduling/mapping. The workers must be allocated to OS tasks and cores. The integration code produced from MCG-generated code has to perform the following steps for each worker:

- *For each worker*, allocate the context of the tasks in the worker.
- Allocate the input/output structures of the root operator, if applicable (tasks generated from annotated instances have no inputs/outputs; they only use channels for communication).
- *For each worker*, initialize the contexts by calling their *init* methods
- *For each step*:
 - Set the inputs
 - For each worker, call methods in an order compatible with dependencies. For each method:
 - Receive data on input channels. The corresponding context fields must contain the received value before the execution of the method and the value should remain constant during execution.
 - Call the method.
 - Send data on output channels. The corresponding context fields must point to a valid memory location before the execution of the method and must contain the value to send after execution.
 - Read the outputs

The mapping file provided by MCG contains all information necessary for allocating tasks and scheduling the different threads. To help specify weights for each task, it is possible to estimate each task WCET using **Timing and Stack Optimizer** analysis on MCG-generated code (see [“Computing Task WCET on MCG-Generated Code”](#) on page 907).

Implementing Integration Code

The integration code for specific multicore targets must generally be written manually to describe all necessary information for OS task initialization, task allocation and scheduling within workers, communication channels between tasks, handling communications with external software components, and any other specific implementation cases. For details about specific implementation activities, see Chapter 10 about [“Multicore Target Integration of Task-Based Code”](#) in *SCADE Suite Technical Manual*.

The SCADE Multicore Integration Toolbox and its Python API functions can facilitate the full automation of code integration with the help of a wrapper example. For details about API functions, see [“Multicore Code Integration API”](#) on page 72 in *SCADE Python API Guide*.

Appendixes

- Appendix A: [“SCADE Suite in Command Line”](#)
- Appendix B: [“SCADE Suite Files in Configuration Management”](#)
- Appendix C: [“SCADE Suite and Source Code Control”](#)
- Appendix D: [“Compiler Configuration for SCADE Suite Code Generation”](#)
- [“Index”](#)

A / SCADE Suite in Command Line

This appendix lists what are the SCADE Suite functionalities available in command line and what syntax must be used for command line options.

- [“General Command Line Syntax”](#)
- [“Command Line Options”](#)

General Command Line Syntax

Several SCADE Suite modules and features can be launched using command lines. Each module is launched using the general command line where you specify the command that calls the module and the options.

The general syntax of SCADE Suite command lines is:

```
scade [-wait|-nowait] [-version [-name]] [-help| -?] [<command>] [<command options>]
```

Possible command options and parameters:

- **-wait** (default) and **-nowait** to specify if you wait or not the end of the command
- **-version** to retrieve the SCADE version. The **-name** option retrieves only the SCADE version name.
- **-help | - ?** to display the command usage
- **<command>** and **<command options>** to initialize SCADE Suite modules. The list of commands and options are described in table below. If no command is specified, SCADE Suite is launched.

Table A.1: List of SCADE Suite command line values

Command	Description	Syntax/Options
-studio	To launch SCADE Suite	“SCADE Suite Launch Command Line”
-viewer	To launch viewer mode	“Viewer Mode Command Line”
-check	To launch Checker	“Semantic Check Command Line”
-metrics	To launch Metric and Rules Checker	“Metrics and Rules Checker Command Line”
-rules_checker		

Table A.1: List of SCADE Suite command line values (Continued)

Command	Description	Syntax/Options
-verify_timing	To launch Timing Verifier	“Timing and Stack Verifiers Command Line”
-verify_stack	To launch Stack Verifier	
-verify	To launch Design Verifier	“Design Verifier Command Line”
-code	To generate code	“Code Generator Command Line”
-locate	To locate an object in models	“Locate Command Line”
-diff	To launch model comparison	“Model Diff Command Line”
-script	To execute a Python/Tcl script	“Scripts Command Line”
	To export/import archives from project	“Archive Export/Import Command Line”

For guidelines about SCADE Suite Simulation command lines, consult Chapter 6 about [“SCADE Suite Simulation Batch”](#) in *SCADE Suite Technical Manual*.

For guidelines about SCADE LifeCycle Reporter in command lines, consult Appendix A about [“SCADE LifeCycle in Command Line”](#) in *SCADE LifeCycle User Manual*.

For guidelines on SCADE Test Model Coverage in command lines, consult Appendix A about [“SCADE Test in Command Line”](#) in *SCADE Test User Manual*.

For guidelines about SCADE Architect command lines, consult Appendix A about [“SCADE Architect in Command Line”](#) in *SCADE Architect User Manual*.

Command Line Options

The syntax of the command options for each SCADE Suite module and feature available in command line is detailed in the following sections:

- [“SCADE Suite Launch Command Line”](#)
- [“Viewer Mode Command Line”](#)
- [“Locate Command Line”](#)
- [“Semantic Check Command Line”](#)
- [“Metrics and Rules Checker Command Line”](#)
- [“Timing and Stack Verifiers Command Line”](#)
- [“Design Verifier Command Line”](#)
- [“Code Generator Command Line”](#)
- [“Scripts Command Line”](#)
- [“Model Diff Command Line”](#)
- [“Archive Export/Import Command Line”](#)

SCADE Suite Launch Command Line

Use the following command to launch SCADE Suite and open a project or workspace:

```
scade -studio [-reset] [-binary] [<project>|<workspace>]
```

Possible command options and parameters:

- **-reset** to reset the user previous settings
- **-binary** to save text files in Unix mode (set the environment variable ETBINARY in Windows or start SCADE Suite with -binary option)
- **<project>** to specify the project path and the project file (.etp)
- **<workspace>** to specify the project path and the workspace file (.wsp)

Viewer Mode Command Line

Use the following command to launch SCADE Suite in viewer mode and open a project or workspace:

```
scade -viewer [<project>|<workspace>]
```

Possible command options and parameters:

- **<project>** to specify the project path and the project file (.etp)
- **<workspace>** to specify the project path and the workspace file (.wsp)

The Viewer mode requires no license. It allows opening any project or model, and accessing all features except actions implying Save or model file updating.

Locate Command Line

Use the following command to open SCADE Suite and locate an object:

```
scade -locate <project> <oid> <ProjectName>
```

Possible command options and parameters:

- <project> sets the project or workspace path to locate the object
- <oid> specifies the OID of the object to locate
- <ProjectName> specifies the name of the project (basename with extension) to locate

Semantic Check Command Line

Use the following command to run a semantic check on a SCADE Suite project:

```
scade -check <project> <configuration> [(-path|-p) <objectPath> | -model]
```

Possible command options and parameters:

- <project> to specify the path to the project file (.etp)
- <configuration> to specify the name of the configuration to use
- **-path** <objectPath> | **-model** to specify the path of the object to check in the model (optional).
Without **-path** and **-model**, the command checks the root operator of the configuration. With **-path**, it checks the object corresponding to the given path, even if the configuration has a root operator. With **-model**, it checks as if the model folder was selected, even if the configuration has a root operator.

The semantic check of SCADE Suite Code Generator verifies the correctness of the semantics used in models. In command line, the check runs on the entire project and cannot be run on subparts.

Metrics and Rules Checker Command Line

Use the following command to use SCADE Suite Metrics and Rules Checker:

```
scade -metrics|-rules_checker <project> -conf <configuration>
```

Possible command options and parameters:

- **-metrics** to compute metrics on specified project
- **-rules_checker** to check methodological rules on specified project
- <project> to specify the project path (absolute or relative to current directory)
- **-conf** <configuration> to specify the name of the configuration to use

Timing and Stack Verifiers Command Line

Use the following commands to launch Timing and Stack Verifier analysis:

```
scade -verify_timing <project> -conf|-c <configuration> [-b]
scade -verify_stack <project> -conf|-c <configuration> [-b]
```

Possible command options and parameters:

- **-verify_timing** to run Timing Verifier on specified project
- **-verify_stack** to run Timing Verifier on specified project
- **<project>** to specify the project path and the project file (.etp)
- **-conf <Configuration>** to specify the name of the configuration to use
- **-b** to launch cross-compiler build.

Design Verifier Command Line

Use the following command to run SCADE Suite Design Verifier on a project:

```
scade -verify <project> [-verbose] [-tasks <objective>*]
```

Possible command options and parameters:

- **<project>** to specify the path (absolute or relative to current directory) to the project file (.etp). Design Verifier analyzes all proof objectives defined in the project.
- **-verbose** to display messages and info during analysis
- **-tasks <objective>*** to specify a list of proof objectives or task folders (separated by blank space) to be verified in the specified project. Specifying task folders verifies the project with all proof objectives in task folder including task sub-folders.

Code Generator Command Line

Use the following command to generate code (executable or DLL for selected [Target](#) setting like FMU, Simulator, Timing and Stack Optimizer, or Simulink) from a SCADE Suite project:

```
scade -code <project> [-node <NodeName>] [-conf <Configuration>] [-sim] [-all]
```

Possible command options and parameters:

- **<project>** to specify the path to the project (.etp) to use as code generation input. The project file contains all info necessary for generation (see Chapter 19 about [“Setting and Launching Code Generation”](#)).
- **-node <NodeName>** to specify the root operator to generate code for the sub-tree of <NodeName>. This option is ignored if the root operator is already set in the Code Generator settings of the project.
- **-conf <Configuration>** to specify the name of the configuration to use for code generation. If option not set, the default configuration set in the project is used (see [“Enabling Configurations”](#) on page 119).
- **-sim** to generate simulation code and build the simulation executable
- **-all** to clean up the generation directory before generating code and building the simulation executable

Scripts Command Line

Use the following command to run a script (Python or Tcl) on a project:

```
scade -script [-project] <project>+ [-script] <script> [[-command] <command>]  
scade -script [-project] <project>+ [-script] <script> [-args <script_arg>+]
```

Possible command options and parameters:

- **<project>+** to specify the path to one or several projects to be loaded before executing scripts.
- **-script <script>** to specify the path of the Python or Tcl script to run.
- **-command <command>** to specify an optional command to be executed in the context of the script.
- **-arg <script_arg>** to specify one or several arguments to pass to a Python <script>. Not supported by Tcl script.

Using both **-command** and **-args** is not supported.

Model Diff Command Line

Use the following command to compute differences:

To compare two projects:

```
scade -diff <project1><project2> [-no_libraries]
      [-report <report>] [-format <format>] [-script <script>] [-GUI]
```

To compare two objects:

```
scade -diff <project1><object1> <project2><object2>
      [-report <report>] [-format <format>] [-script <script>] [-GUI]
```

To compare two XSCADE files:

```
scade -diff <xscade_file1> <xscade_file2>
      [-report <report>] [-format <format>] [-script <script>] [-GUI]
```

Possible command options and parameters:

- <project1> and <project2> specify respectively the path to the first and second project
- <object1> and <object2> specify respectively the object's path in the first and second project
- <xscade_file1> and <xscade_file2> specify respectively the path to the first and second file
- **-no_libraries** to prevent any library comparison. Applies only when comparing two projects. By default, both projects are compared including their libraries.
- **-report** <report> to specify the pathname of the generated report. By default, the report naming convention is: <project1/object1/file1>_diff_<project2/object2/file2>.htm. The report is located in the directory of the first project.
- **-format** <format> to set the report in RTF or HTML (default) format.
- **-script** <script> to specify the script used to compute the differences and generate the report (by default, the script is located at <install_dir>/Scripts/Diff/ScadeDiff.tcl).
- **-GUI** to specify opening diff results in **Diff View**. In that case, no report is produced and options (**-report**, **-format**, or **-script**) applying to the report are ignored.

Archive Export/Import Command Line

Use either of the following commands to export or import an archive:

```
Export: scade -script <project> <script_path>/exportarchive.py
Import: python <script_path>/extract.py -p <project>.zsp -d <target_dir>
```

Possible command options and parameters:

- <project> to specify the path and the project or workspace.
- <script_path> to specify the path of the archive export/import script to run.
- <target_dir> to specify the path to the directory where the archive is uncompressed.

B / SCADE Suite Files in Configuration Management

This appendix lists all files recommended for the Configuration Management of SCADE Suite projects. It remains the responsibility of users to determine what is needed for Configuration Management with regard to their project processes.

Note

Each file is described in details in [“SCADE Suite Files”](#) on page 60 and in [“SCADE LifeCycle ALM Gateway Files”](#) on page 90 in *SCADE LifeCycle User Manual*.

- [“Files Unnecessary for Configuration Management”](#)
- [“Files Necessary for Configuration Management”](#)

Files Unnecessary for Configuration Management

The following files can be ignored for Configuration Management.

Table B.1: Files ignored for configuration management with SCADE Suite

File extension	Description
*.vsw	Workspace file associated with a project or several projects. Automatically generated by SCADE Suite when creating new projects.
*.ewo	User preferences associated with a project workspace.
*.htm	Checker report
*.err	Error log
*.log	Log files from SCADE Suite tools

Storing `.err` and `.log` files may be useful to keep trace of the problems that may occur in the course of the development cycle.

Files Necessary for Configuration Management

The following files are recommended for Configuration Management.

Table B.2: Files for configuration management with SCADE Suite

File extension	Description
SCADE Suite project data:	
*.etp	Project files
Model definition data:	
.xscade/.scade	Description files associated with a project to store all graphical and textual descriptions declared in a model.
*.ann	Annotation files associated with an *.xscade file to store any annotation on model elements.
*.ssl	Symbol files associated with an operator to describe its graphical representation.
Shared data:	
*.aty	Description file of all annotation types available for model annotation.
*.psc	Description of task pragmas associated to operator instances for MCG code generation.
Tools options:	
*.tot	SCADE Suite options associated with a project.
Certification scope:	
*.h and *.c	All SCADE Suite KCG/MCG generated code.
SCADE Suite Simulation usage:	
.in/.sss	Input files describing simulation scenarios.
.sts/.sns	Input files describing application states.
.out/.csv/*.obs	Output files containing simulation results.
SCADE Suite Design Verifier usage:	
*.l4	File associated with Design Verifier activation and usage in projects.

Table B.2: Files for configuration management with SCADE Suite (Continued)

File extension	Description
Synchronization usage (between SCADE Suite and SCADE Architect models):	
*.synchro	Files containing model synchronization settings associated with the project.
SCADE Suite Simulink Importer usage:	
*.ucf	Configuration file completing the mapping rules and hardcoded definitions set in the MCF (Main Configuration File) with custom rules and definitions.
*.mdl, *.m, etc	Simulink input data used for translation into SCADE Suite model.
*_ANNmap.txt	File necessary to avoid losing OIDs at Simulink model re-import.
SCADE LifeCycle ALM Gateway usage:	
.trace	Files containing all traceability information when using Reqtify connections.
.almgp	Files containing connection configuration information for an ALM Gateway project.
.almgt	Files containing information about traceability links saved locally in SCADE projects but not yet exported to the ALM tool in use for requirements traceability.

C / SCADE Suite and Source Code Control

This appendix explains how Source Code Control (SCC) is integrated in SCADE Suite and how to run configuration management activities from projects.

Note

SCC integration is available on Windows platforms only.

- [“Introduction”](#)
- [“SCC Integration Principles”](#)
- [“Setting SCADE Suite Configuration Management”](#)
- [“Using Source Code Control from Projects”](#)

Introduction

SCADE Configuration Management Gateway is the SCC interface delivered to support several Configuration Management Systems (CMS). SCADE Suite users can use the most common configuration management commands from SCADE Suite user interface. SCADE Suite integration with CMS guarantees project consistency. The Configuration Management Gateway enables you to manage projects quickly, regardless of the different file types. Several users can share a project. Therefore, all users can access the latest version of any file, but only one can modify it and save a new version under SCC at once. If

installing an SCC provider on the local machine, SCADE Suite automatically detects it and connects to it provided you set SCADE Suite start-up as expected (see [“Setting SCADE Suite Configuration Management”](#) on page 929).

Warning

Please check CM tool's documentation and support to confirm that it fully supports the SCCI. If the entire SCCI is not supported by the CM tool, some limitations may have impact on SCADE Suite behavior or prevent the SCADE Configuration Management Gateway to work. If you encounter problems, please contact our support for technical guidance.

Selecting SCADE Suite project files for configuration management purposes is summarized in Appendix B about [“SCADE Suite Files in Configuration Management”](#).

SCC Integration Principles

SCC integration into SCADE Suite enables configuration management on projects, files and SCADE Suite objects.

The following SCADE files are managed under Source Code Control:

- Project file (.etp)
- SCADE Suite design files (.xscade and .scade)
- Annotation files (.ann)
- All additional files contained in the SCADE Suite projects.

SCADE Suite design files and annotation files are automatically managed together in order to ensure their consistency.

Source Code Control can manage automatically other files like SSL symbol files and ATY annotation type files, if they are explicitly inserted in the SCADE Suite project.

Warning

SCADE Suite guarantees the consistency of the models when SCC commands are launched on whole projects, but not when they are launched on a selection of files or objects. Consider that local modifications may impact many entities in the project. For example, renaming a type could impact the declaration of several node variables. A partial SCC operation (launched on part of a SCADE Suite model) endangers the consistency of the project.

- Launching SCC commands on a complete SCADE Suite project impacts the project file, all SCADE Suite design files of the project and any attached file like annotation files or symbol files.
- Launching SCC commands on a file(s) or a selection of files impacts the specified file(s).
- Launching SCC commands on a folder (from the File View) impacts all files contained in this folder.
- Launching SCC commands on a design element impacts the file where the element is saved.

Setting SCADE Suite Configuration Management

Configuration management with SCC tools is possible if you configure SCADE Suite application as appropriate. It is necessary to activate the gateway that enables SCC. Optionally, you may set the project to connect automatically to SCC each time you add new elements in the project.

- [“Enabling Source Code Control Integration”](#)
- [“Setting Options for Source Code Control”](#)

Enabling Source Code Control Integration

SCADE Suite can connect with CMS tools at startup provided you enable the gateway that runs Source Code Control.

To enable Source Code Control at startup

- 1 In SCADE Suite, select *Tools > Options* and display **Environment** options.
- 2 Check **Connect to the Source Code Control System at startup**. Restart SCADE Suite to enable this option.

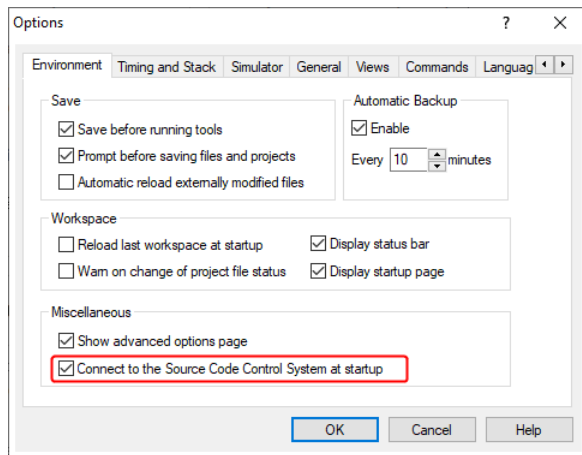


Figure C.1: Enabling connection between SCADE Suite and SCC

If you create a new SCADE Suite project after enabling connections with the SCC System, you are prompted by a confirmation message asking whether or not to add the new project under source control. Click **Yes** to insert the new project under the configuration management environment.

In the **Add to Source Control** window that pops up, check each project file you want to have under Source Control. Each library included in the new project can be checked-in or checked-out at this point of the project configuration for Source Control.

Setting Options for Source Code Control

You can set SCADA Suite projects so that each time you create an element such as an operator, a type, or a constant, SCADA Suite automatically adds the corresponding file to the Source Code Control System (if connected).

To insert new SCADA files automatically under SCC

- 1 In SCADA Suite, select *Tools > Options* and display **General** options.
- 2 Check **Add created files to Source Code Control**.

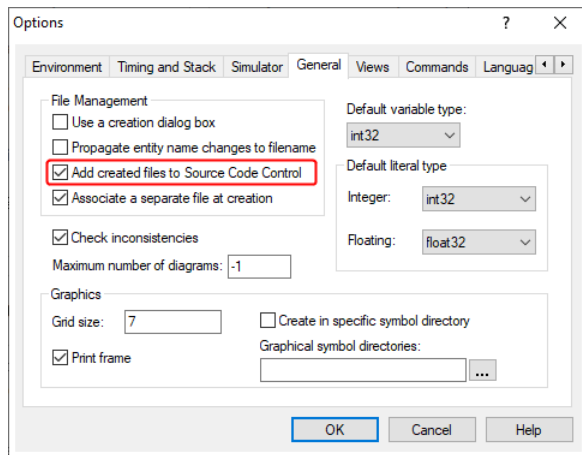


Figure C.2: Setting project files automatically under SCC

If you leave this option unselected, SCADA Suite prompts you each time it creates a new file after you insert any new element in a project connected with the SCC System. You are asked whether each new file must be inserted in the Source Code Control System. Click **Yes** to confirm insertion of the new project file under the configuration management environment. In the **Add to Source Control** window that pops up, check the file you want to include under Source Control.

Using Source Code Control from Projects

To use SCC commands from SCADE Suite, it is necessary that you have a SCC provider installed on the local machine and that SCADE Suite can connect successfully to it (see [“Enabling Source Code Control Integration”](#) on page 930).

Note

The *Project > Source Control* menu, the [“Source Control Toolbar”](#) and a contextual menu are automatically added to SCADE Suite once it is connected to a CMS tool.

- [“Adding/Removing from Source Control”](#)
- [“Checking-In Project Items under SCC”](#)
- [“Checking-Out Project Items under SCC”](#)
- [“Showing Configuration Management History”](#)
- [“Showing Differences between Project Versions”](#)
- [“Showing Source Control Properties”](#)
- [“Importing from Source Code Control”](#)

Source Control Toolbar

The **Source Control** toolbar includes buttons which allow sending commands to the Source Code Control System connected to SCADE Suite. The activation of SCC commands depends on selected SCADE Suite items and SCC status.

The commands of this toolbar are summarized in [Table C.1](#) below.

Table C.1: Commands of the Source Control toolbar

Command	Icons	Description
Get Latest Version		Gets the latest version of selected object, which must be under Source Control. The latest version file(s) replace(s) the previous project file(s). The project must be reloaded to take changes into account.
Check out		Checks out the selected object, which must be checked in. If a latest version file replaces the previous project file, the project must be reloaded to take changes into account.
Check in		Checks in the selected object, which must be checked out under Source Control.
Undo check out		Cancels the last check out done on single files, multiple selection or whole project and returns to previous status. Object must be checked out under Source Control.
Add to Source Control		Places an object under Source Control.
Remove from Source Control		Removes an object from Source Control.
Show History		Shows the configuration management history of an object. The object must be under Source Control.
Show Differences		Shows the differences between the current and previous versions of the selected object. The object must be under Source Control.

Table C.1: Commands of the Source Control toolbar (Continued)

Command	Icons	Description
Source Control Properties		Shows the Source Control properties of the selected object, which must be under Source Control.
Source Control		Launches the Source Control provider tool.
Import from Source Control		Imports an object from Source Control (e.g., project) even if the object is not part of the current workspace. The project must be loaded to take changes into account.

Note

Check out, Get latest version and Import from Source Control impact the files, but not the current project in SCADE Suite. It is necessary to reload it in order to obtain the right project configuration.

The **Scade View** and the **File View** of SCADE Suite display color marks as appropriate to indicate the SCC status of each object in a project:

- **Blue** checkmarks highlight checked-in items
- **Red** checkmarks highlight checked-out items
- **Green** checkmarks highlight indeterminate status (*i.e.*, files contained in the same project folder have different statuses)

Warning

You can modify a project only if its status is “checked-out”. In “checked-in” status, the project is read-only.

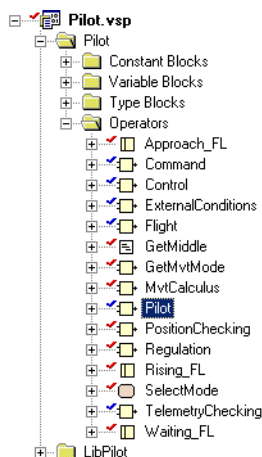


Figure C.3: Scade View highlighting SCC status marks in front of model elements

Adding/Removing from Source Control

You can add/remove SCADE Suite projects or parts to/from Source Control.

To add objects under Source Control

- 1 Select any file in **File View**, or any element in **Scade View**.
- 2 Click **Add to Source Control**  in **Source Control** toolbar.
- 3 In the **Add to Source Control** window that opens, make sure that all the files you want to include in Source Code Control are checked.
- 4 Click **OK**. The SCC tool may ask confirmation for any file that must be included especially if a file already exists in the SCC project. The selected element(s) is(are) displayed with the appropriate status mark.

Tip

You can also right-click on the selected item in File View and select *Add to SCC*.

If the project was previously connected to SCC, you get a confirmation message before reconnecting.

To remove objects from Source Control

- 1 Select a file in **File View** or any element in **Scade View** with a status mark.
- 2 Click **Remove from Source Control**  in the **Source Control** toolbar.
- 3 In the **Remove file from** window that opens, make sure that all the files you want to remove from Source Code Control are checked.
- 4 Click **OK** to confirm. The SCC tool may ask confirmation for any file that needs to be removed (consult its documentation for more details). The status mark(s) in front of the selected element(s) is(are) removed.

Checking-In Project Items under SCC

You can check-in the files corresponding to any item configured in SCADE Suite models. You select items one by one, or by multiple selection in **Scade View** or **File View**.

To check in under SCC

- 1 Select a checked-out item in **File View** or **Scade View**.
- 2 Click **Check in**  in **Source Control** toolbar. In the opened window, make sure to check all files you want to check in under Source Code Control.
- 3 Click **OK**. The status mark(s) in front of the selected element(s) turn(s) to blue.

Checking-Out Project Items under SCC

You can check-out the files corresponding to any item configured in SCADE Suite models. You select items one by one, or by multi-selection in **Scade View** or **File View**.

To check out from SCC

- 1 Select a checked-in item in **File View** or **Scade View**.

- 2 Click **Check out**  in **Source Control** toolbar. In the opened window, make sure to check all files you want to check out from Source Code Control.
- 3 Click **OK**. The status mark(s) in front of the selected element(s) turn(s) to red. Once files are checked out from SCC, you can undo the operation by clicking the Undo Check out button from **Source Control** toolbar.

Showing Configuration Management History

You can display individual file history and project history from **File View**.

To show history

- 1 Select any file under SCC in **File View**.
- 2 Click **Show History**  in **Source Control** toolbar.

For further user instructions, consult the documentation of the SCC tool you are using or click the **Help** button on the opened window.

Showing Differences between Project Versions

You can use this feature to observe the differences between two versions launching an application window specific to the CMS tool you have installed.

To show version differences

- 1 Select any file under SCC in the **File View**.
- 2 Click **Show Differences**  in **Source Control** toolbar.

For further user instructions, consult the documentation of the SCC tool you are using or click the **Help** button on the opened window.

Showing Source Control Properties

You can display the property window related to each project file.

To show file properties in Source Control

- 1 Select any file under SCC in the **File View**.
- 2 Click **Source Control Properties**  in **Source Control** toolbar.

For further user instructions, consult the documentation of the SCC tool you are using or click the **Help** button on the opened window.

Importing from Source Code Control

You can import files from SCC when you want to find files that are stored in the Source Code Control System, but which are not currently part of the SCADE Suite project, and add them to the SCADE Suite project.

Note

The SCC System is responsible for placing all the files into the user's directory.

To import files in Source Control

- 1 Select any file under SCC in the **File View**.
- 2 Click **Import from Source Control**  in **Source Control** toolbar.

For further user instructions, consult the documentation of the SCC tool you are using or click the **Help** button on the opened window.

D / Compiler Configuration for SCADE Suite Code Generation

It is possible to define cross-compilers or Windows native compilers by means of a configuration file. Such configuration mechanism can be set when a “Target” is selected in **Code Integration** settings from SCADE Suite Code Generator (esp. Simulator or Timing and Stack Verifiers) or when launching “**Execute Tests**” or “**Acquire Model Coverage**” with SCADE Test (configuration mechanism out of certification scope).

To start using a specific compiler, you first need to create a compiler configuration file. It is a Tcl file that allows defining all data necessary to create and execute a Makefile for a given compiler. This compiler configuration file must then be registered in SCADE Suite either:

- by saving the compiler configuration file in the `/SCADE/scripts/Compilers` directory of SCADE installation, or
- by declaring in a `SCADE_COMPILERS_DIRS` environment variable the directory where you saved the Tcl file(s). To declare several directories, separate them using columns ‘:’, commas ‘,’ , or semi-columns ‘;’.

A template of configuration file for defining one’s own specific compiler is available in SCADE installation at `/SCADE/scripts/Compilers`. Several examples of configuration files for cross-compilers usable with AbsInt aIT targets are also available in the same directory and can be used as models to help create one’s own compiler configuration file.

Read below how to create a compiler configuration file.

- [“Declaring Compilers”](#)
- [“Configuration File Example”](#)

Declaring Compilers

The Tcl file must define a Tcl namespace corresponding to the file name. In this namespace, the Tcl procedure `GetCompilersDeclaration` must be defined. This procedure has no argument and returns a list of strings.

Example:

```
namespace eval MinGW {
    proc GetCompilersDeclaration {} {
        set vMinGWCompilers [list]
        lappend vMinGWCompilers -Compilers MinGW64::GetCompilers
        lappend vMinGWCompilers -Kind Native
        ##TO COMPLETE ...
        return vMinGWCompilers
    }
}
```

The returned list must be a list of couples <option name> <option value>.

The available options are listed and described in the following sub-sections.

These options are all optional except when specified.

- [“Compiler Definition”](#)
- [“Compiler Default Options”](#)
- [“Compiler Forced Options”](#)
- [“Binary Information”](#)
- [“Extension Information”](#)
- [“Option names”](#)
- [“Rule Creation Options”](#)
- [“Environment Options”](#)

COMPILER DEFINITION

The table below summarizes the compiler definition options:

Table D.1: Compiler definition options

Option	Value
-Compilers	Get compilers procedure name
Description: • <i>Mandatory</i> option Allows naming the procedure that defines the names of the compilers created by the file (these names appear in Compiler drop-down list in Compiler settings of SCADE Suite Code Generator). The signature for this procedure is: () -> list of string Input: NA Output: list of compiler identification	
-Kind	Cross CustomCross Native Get kind procedure name
Description: • <i>Native</i> by default Allows defining the compiler kind for a target extension: Cross for compiler to be used with Timing and Stack Optimizer target extension, CustomCross for custom cross-compiler to be used with Timing and Stack Verifiers target extension, Native for compiler to be used especially with Simulator target extension.	
-CPUTypes	Get CPU types procedure name
Description: • Default CPU type list is empty Allows naming the procedure that gives the list of CPU types managed by the compiler (these types appear in the CPU Types drop-down list in Compiler settings of SCADE Suite Code Generator). The signature for this procedure is: (string) -> list of string Inputs: • ident: selected compiler identification • Input: selected compiler identification Output: list of CPU types supported by selected compiler	

Table D.1: Compiler definition options (Continued)

Option	Value
-aisNotes	Get .ais notes procedure name
Description: • Empty by default (useful only with compilers declared for Timing and Stack Verifiers) Allows naming the procedure that gives the content of .ais file required by aiT tool (Timing and Stack Verifiers / Optimizer) for selected compiler and CPU Type. The signature for this procedure is: (string, string, string) -> list of string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: "Optimizer" in the case of TSO, "Verifier" in the case of TSV Output: content of .ais file required by the aiT tool	

COMPILER DEFAULT OPTIONS

The table below summarizes the compiler default options:

Table D.2: Compiler default options

Option	Value
-DefaultCompilerFlags	Get default compiler flags procedure name
Description: • Empty by default Allows naming the procedure that gives the default compiler options of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments (the output of the procedure appears in the Additional compiler options field in Compiler settings of SCADE Suite Code Generator). The signature for this procedure is: (string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g., language as "ada" for ada and "c" for c) Output: string of default compiler flags	

Table D.2: Compiler default options (Continued)

Option	Value
-DefaultLinkerFlags	Get default linker flags procedure name
Description: • Empty by default Allows naming the procedure that gives the default linker options of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments (the output of the procedure appears in the Additional linker options field in Compiler settings of SCADE Suite Code Generator). The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: string of default linker flags	
-DefaultPreProcessorDef	Get default preprocessors procedure name
Description: • Empty by default Allows naming the procedure that gives the default preprocessor definitions of specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments (the output of the procedure appears in the Preprocessor definitions field in Compiler settings of SCADE Suite Code Generator). The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: string of default Pre Processor definitions	

COMPILER FORCED OPTIONS

The table below summarizes the compiler forced options:

Table D.3: Compiler forced options

Option	Value
-ForcedCompilerFlags	Get forced compiler flags procedure name
Description: • Empty by default Allows naming the procedure that gives some compiler options that are always present, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: string of forced compiler flags	
-ForcedLinkerFlags	Get forced linker flags procedure name
Description: • Empty by default Allows naming the procedure that gives some linker options that are always present, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: string of forced linker flags	

BINARY INFORMATION

The table below summarizes the options to get binary information:

Table D.4: Binary information options

Option	Value
-CompilerBinary	Get compiler binary procedure name
Description: • Empty by default Allows naming the procedure that gives the name of the compiler binary of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: a compiler binary as string	
-LinkerBinary	Get linker binary procedure name
Description: • Empty by default Allows naming the procedure that gives the name of the linker binary of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: a linker binary as string	

Table D.4: Binary information options (Continued)

Option	Value
-ExeBinary	Get executable binary procedure name
Description: • CompilerBinary value by default Allows naming the procedure that gives the name of the binary used to build an executable of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: a compiler binary as string	
-LibConversionBinary	Get linker binary procedure name
Description: • Empty by default Allows naming the procedure that gives the name of the library conversion binary of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments (the library conversion binary is the binary that allows building a static library from a DEF file). The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: a binary of conversion library as string	

EXTENSION INFORMATION

The table below summarizes the options to get extension information:

Table D.5: Extension information options

Option	Value
-ObjectExt	Get object extension procedure name
Description: • Default extension is .obj Allows naming the procedure that gives object file extension of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUtype: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: an object file extension as string	
-LibExt	Get static library extension procedure name
Description: • Default extension is .lib Allows naming the procedure that gives static library file extension of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUtype: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: an library file extension as string	

Table D.5: Extension information options (Continued)

Option	Value
-DllExt	Get dynamic library extension procedure name
Description: • Default extension is .dll Allows naming the procedure that gives dynamic library file extension of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: an dynamic library file extension as string	
-ExeExt	Get executable extension procedure name
Description: • Default extension is .exe Allows naming the procedure that gives executable file extension of the specific compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: an executable file extension as string	

OPTION NAMES

The table below summarizes the options to get option names:

Table D.6: Option names options

Option	Value
-IncludeDirOption	Get include directory option procedure name
Description: • Default option is - I Allows naming the procedure that gives the way to specify an include directory to the compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the include option as string	
-PreProcessorOption	Get preprocessor option procedure name
Description: • Default option is - D Allows naming the procedure that gives the way to specify a preprocessor to the compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the PreProcessor option as string	

Table D.6: Option names options (Continued)

Option	Value
-CompilerOutputOption	Get compiler output option procedure name
Description: • Default option is - o Allows naming the procedure that gives the way to specify the output file of the compiler, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the compiler output option as string	
-LinkerOutputOption	Get linker output option procedure name
Description: • Default option is - o Allows naming the procedure that gives the way to specify the output file of the linker, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the linker output option as string	

Table D.6: Option names options (Continued)

Option	Value
-ExeOutputOption	Get exe output option procedure name
Description: • Default option is -o Allows naming the procedure that gives the way to specify the output file of the binary that build an executable, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the executable output option as string	
-DefFileOption	Get definition file option procedure name
Description: • Default option is --def Allows naming the procedure that gives the way to specify a definition file to the linker, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: the define option as string	

RULE CREATION OPTIONS

The table below summarizes rule creation options:

Table D.7: Rules creation options

Option	Value
-RulePath	Get rule path procedure name
Description: • Empty by default Allows naming the procedure that formats a string used in a rule of a Makefile. If the input gets the default value (empty), nothing changes in the path. The default format of the rule path is a Unix path format: the separator is "/" instead of "\". The signature for this procedure is: (string) -> string Input: a rule path • path Output: Makefile formatted rule	
-SystemPath	Get system path procedure name
Description: • Empty by default Allows naming the procedure that formats a string used in a system command of a Makefile. The signature for this procedure is: (string) -> string Input: a path • path Output: system formatted rule	

ENVIRONMENT OPTIONS

The table below summarizes the environment options:

Table D.8: Environment options

Option	Value
-ObjectDirectory	Get object directory procedure name
Description: • Empty by default Allows naming the procedure that gives the name of the directory created to store the object files, using the compiler identifier, the CPU type, and the C/Ada language as arguments. The signature for this procedure is: (string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: an object directory as string	
-BeforeMakeCommand	Get before make command procedure name
Description: • Empty by default Allows naming the procedure that prepares the environment, config files, and scripts files used to launch the make command, using the compiler identifier, the CPU type, and the target directory path as arguments. The signature for this procedure is: (string,string,string) -> 0 1 (0=OK, 1=KO) Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • Args: other arguments (e.g language as "ada" for ada and "c" for c) Output: output flag (integer: 0 1)	

Table D.8: Environment options (Continued)

Option	Value
-MakeCommand	Get make command procedure name.
Description: • Empty by default Allows naming the procedure that allows getting the command to launch the make process, using the compiler identifier, the CPU type, the rebuild all options (0 1) and the Makefile file name as arguments. The signature for this procedure is: (string,string,0 1,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • reBuildAll: 0 if specify not rebuild all, 1 if any • file: Makefile path Output: Make command	
-MakeWarning	Get make warning procedure name.
Description: • Empty by default Allows naming the procedure that checks if a line of a make command log is a warning and converts it into SCADE Suite logable warning, using as arguments, the compiler identifier, the CPU type, the target directory path, and the line of the make command log file. The signature for this procedure is: (string,string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • targetDir: target directory • line: a line of make command output Output: list of warning messages (if any) or empty string otherwise	

Table D.8: Environment options (Continued)

Option	Value
-MakeError	Get make error procedure name.
Description: • Empty by default Allows naming the procedure that checks if a line of a make command log is an error and converts it into SCADE Suite logable error, using as arguments, the compiler identifier, the CPU type, the target directory path, and the line of the make command log file. The signature for this procedure is: (string,string,string,string) -> string Inputs: • ident: selected compiler identification • CPUType: selected type of the CPU • targetDir: target directory • line: a line of make command output Output: list of error messages (if any) or empty string otherwise	

Configuration File Example

The `MyCompiler.tcl.template` below is an example of configuration file for use with Timing and Stack Verifiers. It is located in SCAD installation folders at `\SCADE\SCADE\scripts\Compilers`. Other scripts `*.tcl.example` are examples that can be renamed and customized to be used with TSV.

```
# Copyright (C) 2021 ANSYS, Inc. ALL RIGHTS RESERVED.
# This source file may be used and distributed without restriction provided that this copyright statement is not removed from the file and
# that any derivative work contains this copyright notice.
#
# Warranty
# ANSYS, Inc makes no warranty of any kind with regard to the use of this Software, either expressed or implied, including, but not limited
# to the fitness for a particular purpose.

global ScriptsDirectory
source [file join ${ScriptsDirectory} Compilers aIsNotes GNUCAIsNotes.tcl]

# Define a TCL namespace with the same name than the current script file name
namespace eval MyCompiler {

    # Define procedure named 'GetCompilersDeclaration' returning a list of couples of options values.
    proc GetCompilersDeclaration {} {
        set vDeclaration [list]
        ## Definition
        lappend vDeclaration -Compilers MyCompiler::GetCompilers
        # Define the compiler kind. Allows knowing if a target extension can use it or not:
        # - 'Cross' will be available for target Timing and Stack Optimizer,
        # - 'CustomCross' for target Timing and Stack Verifiers,
        # - 'Native' especially for target Simulator.
        lappend vDeclaration -Kind CustomCross
        lappend vDeclaration -CPUTypes MyCompiler::GetCPUTypes
        ##
        ## Default options values
        lappend vDeclaration -DefaultCompilerFlags MyCompiler::GetDefaultCompilerFlags
        lappend vDeclaration -DefaultLinkerFlags MyCompiler::GetDefaultLinkerFlags
        lappend vDeclaration -DefaultPreProcessorDef MyCompiler::GetDefaultPreProcessorDef
        ##
        ## Forced options values
        lappend vDeclaration -ForcedCompilerFlags MyCompiler::GetForcedCompilerFlags
        lappend vDeclaration -ForcedLinkerFlags MyCompiler::GetForcedLinkerFlags
        ##
        ## Binaries info
        lappend vDeclaration -CompilerBinary MyCompiler::GetCompilerBinary
        lappend vDeclaration -ExeBinary MyCompiler::GetExeBinary
        ##
        ## Extension info
        lappend vDeclaration -ObjectExt MyCompiler::GetObjectExtension
        lappend vDeclaration -ExeExt MyCompiler::GetExecutableExtension
        ##
        ## Options
        lappend vDeclaration -IncludeDirOption MyCompiler::GetIncludeDirOption
        lappend vDeclaration -CompilerOutputOption MyCompiler::GetCompilerOutputOption
        lappend vDeclaration -LinkerOutputOption MyCompiler::GetLinkerOutputOption
        ##
        ## Rules creation
        lappend vDeclaration -RulePath MyCompiler::GetRulePath
        lappend vDeclaration -ResponseFile MyCompiler::GetResponseFile
        ##
        ## Environment
        lappend vDeclaration -ObjectDirectory MyCompiler::GetObjectDirectory
        lappend vDeclaration -BeforeMakeCommand MyCompiler::BeforeMakeCommand
        lappend vDeclaration -MakeCommand MyCompiler::GetMakeCommand
        lappend vDeclaration -MakeWarning MyCompiler::GetWarning
        lappend vDeclaration -MakeError MyCompiler::GetError
        ## Specific aiT Custom command
        lappend vDeclaration -AIsNotes MyCompiler::AIsNotes
        return $vDeclaration
    }

    proc GetCompilers {} {
        # Return the names of the compilers created by the file
        # These names appear in the 'Compiler' drop-down list
        # on the 'Compiler' tab in SCAD Suite Code Generator settings
        return [list "My Compiler"]
    }

    proc GetCPUTypes { ident } {
        # Return the list of CPU types managed by the selected compiler
        # These types appear in the 'CPU Types' drop-down list
        # on the 'Compiler' tab in SCAD Suite Code Generator settings
        return [list MPC555 MPC565 e300 MPC755s]
    }
}
```

```

#####
## Custom Procedures Specific for aiT
#####

proc AisNotes { ident CPUType arguments } {
    # Return the content of .ais file required by aiT tool (Timing and Stack Verifiers / Optimizer)
    # for selected compiler, CPU Type and analyzer tool (arguments = 'Verifier' | 'Optimizer')
    return [GNUCL::ais::GetNotes $CPUType $arguments]
}

#####
## Procedures To Get Default Value For Compiler Options
#####

proc GetDefaultCompilerFlags { ident CPUType args } {
    # Return the default compiler options of the selected compiler and CPU Type
    # The output of the procedure appears in the 'Additional compiler options' text field
    # on the 'Compiler' tab in SCADE Suite Code Generator settings
    return ""
}

proc GetDefaultLinkerFlags { ident CPUType args } {
    # Return the default linker options of the selected compiler and CPU Type
    # The output of the procedure appears in the 'Additional linker options' text field
    # on the 'Compiler' tab in SCADE Suite Code Generator settings
    return ""
}

proc GetDefaultPreProcessorDef { ident CPUType args } {
    # Return the default preprocessor definitions of the selected compiler and CPU Type
    # The output of the procedure appears in the 'Preprocessor definitions' text field
    # on the 'Compiler' tab in SCADE Suite Code Generator settings
    return ""
}

#####
## Procedures To Get Forced Value For Compiler Options
#####

proc GetForcedCompilerFlags { ident CPUType args } {
    # Return some compiler options that will be always present for the selected compiler and CPU Type
    set vFlags "-c -g"
    return $vFlags
}

proc GetDLDFilename { CPUType } {
    set vFileName [string tolower $CPUType]
    append vFileName ".dld"
    return $vFileName
}

proc GetForcedLinkerFlags { ident CPUType args } {
    # Return some linker options that will be always present for the selected compiler and CPU Type
    # These options can reference an external linker script (.dld, .ld, .cmd...)
    global ScriptsDirectory
    set vFileName [GetDLDFilename $CPUType]
    set dldFile [file join ${ScriptsDirectory} Compilers Etc mingwgcc ${vFileName}]
    set vFlags "-g -Wl,-T\"${dldFile}\" -static -lm -lc"
    return $vFlags
}

#####
## Procedures To Get Extensions
#####

proc GetObjectExtension { ident CPUType args } {
    # Return object file extension for the selected compiler and CPU Type
    return ".o"
}

proc GetStaticLibraryExtension { ident CPUType args } {
    # Return static library file extension for the selected compiler and CPU Type
    return ".a"
}

proc GetExecutableExtension { ident CPUType args } {
    # Return executable file extension for the selected compiler and CPU Type
    return ".elf"
}

```


Index

Symbols

- . 60
- .ann 60
- .aty 60
- .csv 60
- .err 60
- .etp 60
- .ewo 60
- .ewo files (Suite) 60
- .ffonts 60
- .fonts 60
- .htm 60
- .in 60
- .log 60
- .out 60
- .scade 60, 62
 - inserting textual definitions 270
- .sns 60
- .ssl 60
- .sss 60
- .sts 60
- .tot 60
- .vsw 60
- .xscade 60, 62
- \$(CG) 837
- \$(Configuration) 837
 - environment variable for TSV/TSO 804
- \$(NodeName) 837
 - environment variable for TSV/TSO 804

- \$(NodePath)
 - environment variable for TSV/TSO 804
- \$(SCADE)
 - environment variable for project library 116
- \$(TargetDir) 844

A

- Action
 - style category 137
- Action display
 - setting in embedded mode 201
 - setting in hidden mode 202
 - switching graphical/textual mode 266
- Action on transitions
 - specifying 179
- Actions
 - editing textual mode 268
 - fundamentals about 23
- activate every 402, 405
- Activation
 - on condition 402
 - see Higher-order patterns 402
 - style category 137
- Active views
 - zooming 148
- Add 131
- Add an Input to a Predefined Operator 235
- Add Configuration 121
- Add created files to Source Code Control 931
 - tools options 78
- Add Instance Path
 - simulation 610
- Add to current workspace 97
- Add to Equation Set 254
- Add to Favorites 131
- Add to Source Control 933, 935
- Add/Remove Point 292
 - on fork transitions 175
 - on state transitions 172
- Additional compiler options 590
- Additional Include Directories 855
- Additional include directories
 - code build options 591
- Additional linker options 590
- Additional mex options
 - Simulink target options 662
- Align Bottom
 - design elements 284
 - straightening vertical wires 293
- Align Center
 - design elements 284
- Align Left
 - design elements 284
 - straightening horizontal wires 293
- Align Middle
 - design elements 284
- Align Right
 - design elements 284
 - straightening horizontal wires 293

Index

- Align Top
 - design elements 284
 - straightening vertical wires 293
 - Aligning
 - components bounding box 285
 - components referent element 286
 - selection of design elements 284
 - All
 - expansion mode 847
 - I/O separation mode 857
 - All except selected
 - debug options 841
 - expansion mode 848
 - I/O separation mode 858
 - All Styles
 - creating styles 136
 - Analysis messages
 - fundamentals 758
 - Analysis reports
 - Design Verifier 724
 - Analysis scenarios
 - fundamentals 726
 - Analyze 756
 - Analyze One By One 756
 - Analyze Stack 809
 - Analyze Timing 809
 - Analyzing models
 - about semantic differences 529
 - Angle snap 152
 - Annotation files
 - for SCADE Suite 64
 - Annotation type files
 - for SCADE Suite 63
 - Annotation types
 - files (Suite) 63
 - Annotations
 - external fonts 65
 - files (Suite) 64
 - Apply 265
 - for changes in textual nodes 261
 - in Symbol Editor 296, 297, 298, 299, 300, 301
 - Apply Higher Order
 - creating Boolean activation 403
 - creating clocked activation 405
 - creating mapfold iterator 400
 - creating resetable node 408
 - Applying iteration pattern 400
 - Applying styles
 - by design element selection 138
 - Arcs
 - drawing 298
 - As Toolbar 131
 - Assertions
 - in property verification context 739
 - Assign
 - style category 137
 - Associate a separate file at creation
 - tools options 78
 - Assume
 - defining contract 242
 - Authors
 - Project properties (Suite) 425
 - Automatic Backup options 75
 - Automatic mapping
 - setting Simulink target 662
 - Automatic reload externally modified files
 - tools options 75
- ## B
- Back 420
 - Back to Main
 - contextual menu 147
 - Background
 - design layout 301
 - Backward cycles limit
 - simulation option 584
 - Basic clock 669
 - Bezier style
 - transitions 173, 176
 - Bitwise operators
 - toolbar 47
 - Bitwise toolbar 47
 - Black-box co-simulation
 - launching 665
 - setting FMU target 635
 - setting Twin Builder target 653
 - Body
 - textual action property (If Node) 466
 - textual action property (When Branch) 469
 - textual state properties 473

Index

Border
 embedded design area 143
Breakpoint
 fundamentals 573
Bring Forward 301
Bring to Front 301
Browse References 413
Browse toolbar 49

Build
 exporting SCADE Suite model
 to FMU 636
 exporting SCADE Suite model
 to Twin Builder 654

C

C identifiers
 setting 860
Call Graph 35
Call Graph Browser
 presentation 30
Callers 416
Calls 416
Change control
 in SCADE Suite models 525
Char array values
 defining 210
Char type
 defining array value 210
Check 276
Check in 933, 936
Check out 933, 937
Check Rules 505
Checker 274

 command line 918
Checker report 277
Choice toolbar 48
Choose Font 299
Clear
 removing output content in
 Suite 39
Close All Tabs
 contextual menu 148
Close Other Tabs
 contextual menu 148
Close Tab
 contextual menu 148
Closed curves
 drawing 298
Closing active views 148
CM tool 928
Code expansion
 setting mode 847
Code Extension
 Simulink target 659
Code generation
 enabling FMU target 633
 enabling Simulink target 659
 enabling Twin Builder
 target 651
 setting user pragmas 859
Code generation settings
 CPU type 854
Code Generator
 command line 920
 setting configuration 120

Code Generator-Simulator
 toolbar 829
Color
 applying to design
 elements 134
Colored graphs
 displaying in simulation 584
Command lines
 using SCADE Suite modules
 in 915
Company
 Project properties (Suite) 425
Comparing models
 as separate projects 529
 in same workspace 530
Comparison toolbar 47
Compiler
 setting simulation code
 build 589
Complete Word 262
Compose 237
Compute Metrics 503
Compute Types 247
Condition activation 402
 setting with clock
 expression 405
 setting with default values 402
Condition on transitions
 specifying 179
Conditional signal
 designing 241
Configuration for FMU
 creating 633
Configuration for Twin Builder

Index

- creating 651
- Configuration Management 927
 - multifile in projects (Suite) 66
 - setting SCADE Suite 929
 - showing history 937
 - tools 927
- Configuration Management Gateway
 - in SCADE Suite 6
- Configurations
 - activating (Suite) 120
 - associating with tools (Suite) 122
 - creating custom (Suite) 121
 - creating new (Suite) 121
 - deleting 121
 - setting tool parameters 123
- Configurations Filter
 - in Suite 122
- Connect by Name 248
- Connect by Rank 249
- Connect probes to
 - Simulink target options 662
- Connect to Source Code
- Control System at startup 930
- Connect to the Source Code
- Control System at startup 76
- Connection wires
 - adding segments 292
 - distributing 288
 - optimizing 291
 - removing segments 292
- Connections
 - drawing multiconnections by name 248
 - drawing multiconnections by rank 249
 - straightening
 - multiselection 293
- Consistency verification 745
- Consistent message
 - reading result 759
- Const
 - pragma 863
- Constant 218
- Constants
 - setting pragmas 863
- Constants View
 - using 41
- Constants Views
 - purpose and user tasks 41
- Consumers 415
- Context
 - searching in models (Suite) 410
- Contextual help
 - in SCADE Suite 12
- Contracts
 - defining assume/ guarantee 242
- Contradictory
 - fundamentals 726
- Control flow
 - signals interaction 24
- Convert to Graphical 267
- Convert to Textual 267, 466, 469, 473
- Convert to textual 261
- Copy Instance Path
 - simulation 610
- Cosimulation
 - main API functions 688
 - setting application with API 688
- Co-simulation host
 - setting FMU target options 635
 - setting Twin Builder target options 653
 - Simulink target options 661
- Cosimulation mode 687
- Counter example 727
 - analysis scenarios 726
- Counter-example 719
- CPU type 854
- Create Counterpart Plug 311
- Create Counterpart Variable 311
- Create in specific symbol directory 295
 - in Tools options 78
- Create new workspace 97
- Create toolbar 47
- Creating fork transitions 175
- Creating parallel State Machines 169
- Creating State Machines 168
- Creating state transitions 172
- Cut/Paste 279, 280
- Cycle
 - simulation 572

Index

- D**
 - Data flows
 - connecting design elements 246
 - type propagation
 - Type propagation 247
 - Data typing
 - flow composition/
decomposition 236
 - Date
 - Project properties (Suite) 426
 - Debug All 840
 - Debug options
 - selecting non-observed operators 841
 - selecting observed operators 841
 - setting all operators observable 840
 - Decisional operators
 - toolbar 48
 - Declarations
 - deleting from model 282
 - private versus public 130
 - Decompose 238
 - Deep Consumers 415
 - Deep Producers 413
 - Default
 - pragma 861
 - Default kind
 - transition creation 178
 - Default layout
 - transition creation 178
 - Default model file
 - project creation 99
 - Default Page Format
 - Diagram properties (Suite) 437
 - Default size 289
 - Default transitions
 - creation modes 178
 - Default type
 - tools options 78
 - Definitions 412
 - modifying in design 281
 - Delete from Model 282
 - Delete Hidden Values View 252
 - Description
 - Project properties (Suite) 425
 - Design
 - creating symmetric instances 234
 - Design elements
 - applying font style and color 134
 - associating styles 134
 - distributing 288
 - drawing multiple connections 248
 - editing wire and transition lines 292
 - moving (Suite) 279
 - multiple wires 246
 - nudging position 291
 - optimizing connection wires 291
 - optimizing size 289
 - position and size 482
 - resizing 289
 - resizing manually 290
 - rotating 290
 - wires one by one 246
- Design help
 - setting grid 152
 - Design layout
 - optimizing tools 283
 - Design Verifier
 - Analysis reports 724
 - command line 919
 - degraded mode analysis 720
 - introduction 718
 - Design Verifier toolbar 722
 - Design Verifier View
 - presentation 37
 - Designing
 - conditional signal emission 241
 - quick multiconnections 248
 - references to local data 240
 - Diagram Kind
 - Project properties (Suite) 427
 - Diagram Simulation View 576
 - Diagram View
 - using 44
 - Diagram Views
 - enabling/disabling display options 143
 - grabbing design 148
 - purpose and user tasks 44
 - selecting display options 73
 - setting grid 152
 - zooming on design 149
 - Diagrams
 - creating additional 204

Index

- editing style layout 134
- maximum number when splitting 204
- naming rule when splitting 206
- opening multiple 41
- splitting 205
- Diff Analyzer 526
- Diff report
 - fundamentals 528
- Diff tab
 - examining model comparison 532
- Diff View
 - fundamentals 527
 - in SCADE Suite 40
 - locating differences in models 534
 - showing/hiding 527
- DiffAnalyzer tab 530
- Digits after decimal point
 - project simulation property 585
- Discrete-Time Integrators 669
- Display
 - changing format in Watch window 611
- Display arrays as 3.1
 - tools option 81
- Display Border
 - display options 203
- Display border
 - tools option 80
- Display char arrays as string
 - tools option 80
- Display effect on truncated text
 - tools option 80
 - tools options 150
- Display generated pragmas
 - tools option 80
- Display grid
 - tools option 80
- Display grid (editing symbols)
 - tools option 81
- Display Hidden Notes
 - enabling in diagrams 143
- Display Internal Types
 - enabling in diagrams 143
- Display internal types
 - tools option 79
- Display Internal Variables
 - enabling in diagrams 143
- Display internal variables
 - tools option 79
- Display Local Types
 - enabling in diagrams 143
- Display local types
 - tools option 79
- Display Local Variables
 - enabling in diagrams 143
- Display local variables
 - tools option 80
- Display modes
 - content of states and actions 199
- Display Options 143
- Display options
 - in diagrams 73
- Display page boundary
 - tools option 80
- Display page boundary (editing symbols)
 - tools option 81
- Display Pin Names
 - enabling in diagrams 143
- Display pin names
 - tools option 80
- Display Pin Ranks
 - enabling in diagrams 143
- Display pin ranks
 - tools option 80
- Display Pragmas
 - hiding in Text Diagrams 264
 - showing in Text Diagrams 264
- Display startup page
 - tools options 76
- Display status bar
 - tools options 76
- Display Views as Columns 310
- Display Views as Rows 310
- Distribute 288
- Docking Windows
 - showing/hiding 29
 - viewing 29
- Docking windows
 - Diff View in SCADE Suite 40
 - Output 38
 - Properties in SCADE Suite 37
 - Requirements in SCADE Suite 38
 - Shortcuts in SCADE Suite 39

Index

Documenting system 755

Down

Go To toolbar 417

Duplicating active views 147

DV proofs

consistency check 745

dynamic access detection 745

float infinity detection 745

Float NaN detection 745

integer division-by-zero
detection 745

integer overflow detection 745

Dynamic access verification 745

E

Edit Justifications 509

Edit Symbol 295

Editing views

opening in SCADE Suite 41

Editor

viewer mode command line
(Suite) 917

Effect

see transition action 23

Ellipses

drawing 298

Ellipsis

truncated text mark 150

Else branch 177

Embed contents

on states 200

Embedded display

showing action content 201

Embedded mode

states and actions 199

Enable automatic backup

tools options 75

Enable colored graphs

simulation display options 584

Enable white box

FMU target options 635

Simulink target options 661

Twin Builder target options 653

Entity

style category 137

Enum Value

pragma 861

Enumeration

setting pragmas on value 861

Enumeration value

setting pragmas 861

Equation sets

creating 254

Error

files (Suite) 63

Error messages 50

ETP files (Suite) 60

Excluded Files 855

Excluded files

code build options 591

Execute MATLAB M-File 666

Existing Item 206

Expand

pragma 861

Expanded operators 847

Expansion mode

definition of expanded
operators 847

definition of non-expanded
operators 847

expanding all operators 847

expanding selected
operators 848

setting non-expanded
operators 848

Export probes

FMU target options 636

Export sources

FMU target options 636

Expressing properties

observer operator 730

Expressing properties context

top-level operator 730

External Ada code

simulation 685

External C code

simulation 673

External code

specifying Ada file type 685

specifying C file type 674

External fonts

in .fonts files 65

F

Falsifiable 727

Falsifiable message

fundamentals 726

reading result 759

Favorites

access to toolbar list 131

Index

- adding in list 131
- managing 73
- removing from list 132
- File extensions
 - File folder properties (Suite) 430
- File management
 - customizing (Suite) 68
 - default project storing (Suite) 66
 - options 77
- File type
 - file properties for simulation (Suite) 432
- File View
 - in SCADE Suite 31
 - modifying definitions 281
 - presentation 30
- Files
 - _Sfct.m 664
 - .ann file (Suite) 64
 - .aty file (Suite) 63
 - .etp (Suite) 60
 - .ewo (Suite) 60
 - .font file (Suite) 65
 - .htm file (Suite) 65
 - .log file (Suite) 65
 - .scade 62
 - .tot file (Suite) 65
 - .vsw (Suite) 60
 - .xscade 62
 - default .font file (Suite) 65
 - ERR file (Suite) 63

- generated files for co-simulation in Twin Builder 654
- generated files for FMI co-simulation 636
- libraries (Suite) 61
- saving in UNIX text format 114
- SSL file (Suite) 64
- Files (Suite)
 - list of 60
- Fill end of string with padding character
 - Project properties (Suite) 429
- Filtering content
 - in model browsers (Suite) 421
- Final states
 - in State Machines 172
- Find
 - in Textual Views 262
 - searching in models (Suite) 410
- Find Next
 - in Text Diagrams 262
- Find toolbar (Suite) 49
- Find what
 - in Text Diagrams 262
- Flatten operator
 - defining 363
- Float
 - contextual menu 147
- Float Infinity verification 745
- Float Not a Number verification 745
- Flow composition 236
- Flow decomposition 236
- FMU

- generated files for FMI co-simulation 636
 - setting target options 634
- Focus on Selection (Suite) 421
- Folder
 - creating new 114
- Folder extensions 114
- Folder name 114
 - File folder properties (Suite) 430
- Font 299
- Font files 65
- Font style
 - applying to design elements 134
- Foreground
 - design layout 301
- Fork transitions
 - creating 175
- Format
 - Diagram properties (Suite) 437
 - Project properties (Suite) 428
 - project simulation property 585
- Forward 420
- From root operator
 - code execution option 852
- Full Screen
 - opening views in 151

G

- Generate context as global 842
- Generate data type conversion blocks

Index

- Simulink target options 663
- Generate loop bounds annotations 801
- Generate Observability File from Watch List 607
- Generate system memory state management 843
 - for FMUs 636
- Generate type copy functions 801
- Generating reports
 - about model comparison 536
- Generation name
 - Simulink target options 662
- Generic types
 - setting 216
- Get Latest Version 933
- Global prefix 843
- Go to line 264
- Go To toolbar (Suite) 49
- Grabbing design boards 148
- Graph resolution
 - simulation display options 584
- Graph visible values
 - simulation display options 584
- Graph window 578
- Graphical 295
- Graphical design
 - cut/paste elements 279
- Graphical layout
 - see Styles 134
- Graphical operators
 - in Diagram Views 44

- Graphical panel
 - inserting in Suite project 305
 - simulation workflow 625
- Graphical symbol directories 78
- Graphical symbols
 - adding text boxes 299
 - drawing 298
 - drawing arcs 298
 - drawing ellipses 298
 - drawing line 298
 - drawing polycurves 298
 - drawing polygons 298
 - drawing rectangles 298
 - editing layout 295
 - editing/adding text 299
 - input/output name 300
 - ordering layout 301
 - shaping 297

- Graphics options 78
- Grid
 - setting properties 152
- Grid options 81
- Grid Properties 152
- Grid properties
 - color 152
 - line spacing 152
- Grid size 78
- Grid visible 152
- Guarantee
 - defining contract 242

H

- Hatch background in views
 - highlighting truncated text 150

- Header file 844
 - Ada code generation 846
- Help
 - launching from SCADE Suite 10
- Help about SCADE Suite
 - contextual access 12
 - contextual tooltip 12
- Hidden
 - insert 163
- Hidden display
 - hiding action content 202
- Hidden mode
 - accessing state/action content 203
 - states and actions 199
- Hidden state display
 - showing border 143
- Hide
 - pragmas in Text Diagrams 264
- Hide content
 - on states 200
- Hierarchy 22
- High order patterns
 - toolbar 48
- Higher Order
 - no pattern 449
- Higher Order property
 - Boolean activate pattern 450
 - clocked activate pattern 450
 - fold pattern 450
 - foldi pattern 450
 - foldw pattern 450
 - foldwi pattern 450

Index

- map pattern 450
- mapfold pattern 450
- mapfoldi pattern 450
- mapfoldwi pattern 450
- mapi pattern 450
- mapw pattern 450
- mapwi pattern 450
- restart pattern 450
- Higher Order toolbar 48
- Higher-order patterns
 - setting operator call activation 402
 - setting operators into 378
- I**
- I/O connections
 - types compatibility 306
- If blocks
 - creating 180
 - defining textual actions 266
 - designing 180
 - moving 185
- If Node actions
 - defining textual body 465
 - setting display mode 465
- If nodes
 - creating 182
 - defining actions of 185
 - deleting 187
 - designing 182
 - removing else branch 187
 - removing then branch 187
 - setting condition of 184
 - switching branches of 187
- Ignore
 - failed assertion option 584
- Import Connections 311
- Import from Source Control 934, 938
- Import WCET from Analysis Tool 902
- Imported code
 - inserting include directories 591, 855
- Imported nodes
 - specifying Ada file type 685
 - specifying C file type 674
- Imported Operators
 - specializing 196
- Imported operators (Ada)
 - simulation with 685
- Imported operators (C)
 - simulation with 673
- Imported types
 - setting pragmas 863
 - simulating 675
 - simulation 675
- Inconsistent 759
- Inconsistent message
 - reading result 759
- Indeterminate message
 - fundamentals 726
 - reading result 759
- Index
 - Project properties (Suite) 426
- Info
 - in operator Twin Builder properties 650
- Initial states
 - in State Machines 172
- Input
 - insert 162
- Input threshold 839
- Input vector scenarios
 - creating 614
- Insert Folder 114
- Insert Typed Probe 241
- Instance
 - style category 137
- Instance Graph 903
- Instances Model Browser 575
 - observable variables 575
- Instrumenter
 - setting configuration 120
- Integer Division by Zero verification 745
- Integer Overflow verification 745
- Integration with SCADE Display 5
- Interactive simulation
 - launching 596
- Interrupted message
 - fundamentals 726
 - reading result 759
- Item
 - inserting into workspace 206
- Iterators
 - applying pattern 400
 - see Higher-order patterns 378

Index

J

Joint Design

- creating Rapid Prototyper panel in project 305
- inserting graphical panel in Suite project 305
- workflow 304

Joint model-based design

- SCADE Suite toolset 5

K

Keep

- pragma 861

Keep assertions (Ada code) 845

Keep assertions (C code) 843

Keep probes in generated code

- Ada code generation 845
- C code generation 843

Key symbol

- meaning 34, 130

L

Landscape

- Diagram properties (Suite) 437
- Project properties (Suite) 428

Language version

- Project properties (Suite) 426

Large Icons

- in shortcuts list 132

Large models

- filtering content (Suite) 421

Layout

- editing style 134
- setting grid 152

Layout toolbar 48

- use for connection straightening 293
- use for distributing 288
- use for resizing 289

Library

- files 61

Lines

- drawing 298

Link labels margin 81

Load Watch List from Observability File 610

Local

- insert 165

Local variables

- declaring in transition scope 179
- designing input reference 240
- designing output reference 240

Local variables as static 839

Locate

- command line 918
- connected I/Os in graphical panel model 312

log files 65

Log only

- failed assertion option 584

Logical operators

- toolbar 47

Logical toolbar 47

M

Macro File 433

Macro user definition

implementing 676

Make operator

- defining 363

Makefile compilation

- excluding files 591

Manage Metrics and Rules 501

Managing perspectives 92

Managing projects 73

Manifest

- pragma 862

Mark All

- lines in Text Diagrams 264

Mathematical comparison

- toolbar 47

Mathematical operators

- toolbar 47

Mathematical toolbar 47

Maximum number of diagrams 204

- tools options 78

Maximum text size

- Simulator display options 584

MCG

- Multicore Code Generator 4

Menus

- customizing 73

Merge Declaration only to Left 538

Merge Declaration only to Right 538

Merge to Left 538

Merge to Right 538

Metrics and Rules Checker

Index

- command line 918
- managing metrics and rules 501
- Metrics and Rules Checker toolbar 497
- M-file
 - executing file 663
- Model
 - change control 525
 - checking semantics 274
- Model Browsers
 - viewing 29
- Model browsers
 - filtering content (Suite) 421
- Model comparison
 - analysis 526
 - analysis reports 528
 - analyzing differences 529
 - Diff Analyzer 526
 - generating analysis reports 536
 - locating differences 534
- Model designing
 - cut/paste 279, 280
 - in full screen 151
- Model Diff
 - command line 921
 - examining differences 532
 - fundamentals 525
 - generating analysis reports 536
 - launching analysis 529
 - locating model differences 534
 - reports 528
- Model Diff Analyze 529, 530
- Model Diff Report 536
- Model editing
 - arranging views 73
- Model organization
 - packages 73
- Move to Next Tab Group 146
- Move to Previous Tab Group 146
- multiconnections
 - drawing quickly 248
- Multicore code generation
 - enabling Timing and Stack Optimizer target 907
- Multiple diagrams
 - opening 41
- N**
- Name
 - pragma 860
- Name length 856
 - C code generation 843
- Name Rules Definition
 - for I/O connection 314
- Navigate
 - from Text diagrams 265
- Navigate toolbar (Suite) 49
- Navigating
 - to type definition 419
- Network Elements
 - Symbol Editor 50
- New Equation Set 254
- New Fork Symbol 175
- New Group 131
- New Horizontal Tab Group 146
- New If Block 180
- New If Node 182
- New State 171
- New State Machine 168, 169
- New Terminator 243
- New Textual Expression 272
- New variable symbols
 - Project properties (Suite) 429
- New Vertical Tab Group 146
- New When Block 188
- New When Branch 190
- New Window 147
- Next 532
 - Go To toolbar 417
- Next Item From Browser 419
- Next Item From Output 418
- No Expand
 - pragma 861
- No periodicity
 - code execution option 852
- No Separate I/Os
 - pragma 862
- Non-expanded operators 847
- Note type files
 - project creation 99
 - Project properties (Suite) 426
- Nudge Down 291
- Nudge Left 291
- Nudge Right 291
- Nudge toolbar 48
 - using 291
- Nudge Up 291

Index

Number of inputs 449

O

Object File 433

Observability File

setting 607

Observable variable 575

Observable variables

fundamentals 571

generating OBS file 607

saving simulation output 606

Observer operator 718, 730

Offset

code execution option 852

root operator periodicity 447

Open 32

Open Hidden Values View 251

Open Workspace 54

Operator

encapsulating 252

Operator calls

substituting in design 238

Operator iteration

applying iteration pattern 400

Operators

graphical outlook 50

setting pragma for I/O
separation 862

setting pragma for no I/O
separation 862

setting pragmas 861, 863

simulating partially 593

straightening connections 293

Optimization

fundamentals 826

Optimize

adjusting state/action symbol
to text 269

Connection 291

Size 289

Option files 65

Order 301

Organize Favorites 132

Orientation

Diagram properties (Suite) 437

Project properties (Suite) 428

Output

insert 162

Output File

setting 606, 607

Output file

disabling 607

extending observability 607

Output Value File 606

Output Window 38

showing/hiding 29

Override

code execution option 852

P

Pack in Operator 252

Pack in State 253

Package 127

Packages

creating new 125

creating sub-packages 127

model organization 73

opening namespace 128

public declaration 128

setting pragmas 863

setting private visibility
(Suite) 130

setting public visibility
(Suite) 130

setting visibility rules 130

Page Bounds 205

Pan

grabbing design 148

Parallelism 22

fundamentals about 25

Period

code execution option 852

root operator periodicity 447

Periodicity

setting Simulink target 660

Persist As

project file properties
(Suite) 431

Perspective toolbar 46

Perspectives

default (Suite) 92

managing 92

switching 93

Pfproperty analysis report

fundamentals 726

Physical name

in operator Twin Builder
properties 650

PinRank

style category 137

Polycurves

drawing 298

Index

- Polygons
 - drawing 298
- Polyline style
 - transitions 173, 176
- Port Number
 - project simulation
 - property 586
- Portrait
 - Diagram properties (Suite) 437
 - Project properties (Suite) 428
- Position and size 482
- Pragma
 - pragma 863
- Pragmas
 - setting on objects 859
- Predefined operators
 - dedicated toolbars 47
 - symmetric instances 234
 - symmetric use 449
- Predefined proof objective 745
- Preemption 22
- Preprocessor 590
- Previous
 - Go To toolbar 417
- Previous Difference 532
- Previous Item From Browser 419
- Previous Item From Output 418
- Print
 - active views 153
- Print frame
 - in Tools options 78
- Print Preview
 - active views 153
- Print Setup 153
- Print Zoom
 - Diagram properties (Suite) 437
 - Project properties (Suite) 428
- Priority
 - style category 137
- Private
 - visibility inside packages (Suite) 130
- Private constants
 - key symbol 34, 130
- Private operators
 - key symbol 34, 130
- Private packages
 - key symbol 34, 130
- Private types
 - key symbol 34, 130
- Probe
 - insert 166
- Probes
 - designing reference 241
- Producers 413
- Project
 - creating new 96
 - renaming 102
 - setting simulation options 585
 - sharing 927
 - switching as active 102
- Project creation
 - in active workspace 97
 - in new workspace 97
- Project files
 - check-in under SCC 936
 - check-out under SCC 936
 - opening 54
- Project files (Suite) 60
- Project folders
 - creating 114
- Project into Workspace 112
- Project management 73
 - creating configuration (Suite) 121
- Project pathname
 - FMU target options 635
 - Simulink target options 661
 - Twin Builder target options 653
- Project toolbar (Suite) 46
- Projects
 - consistency with SCC 929
 - inserting into workspace 112
- Projects (Suite)
 - providing project description 425
 - specifying company information 425
 - specifying index information 426
 - specifying list of authors 425
 - specifying project date 426
 - specifying project subtitle 425
 - specifying project title 425
 - specifying reference information 425
- Prompt before saving files and projects
 - tools options 75

Index

- Proof objectives (predefined)
 - fundamentals 734
- Proof objectives (user defined)
 - fundamentals 732
- Propagate entity name changes to filename 77
- Propagation
 - of types 247
- Properties window
 - in SCADE Suite 37
- Property 718
- Proving concept 718
- Public
 - visibility outside packages (Suite) 130

R

- Raised an Error message
 - fundamentals 726
 - reading result 759
- Rapid Prototyper panel
 - creating from Suite project 305
- Recent Files 59
- Recent Workspaces 59
- Record Scenario
 - FMU target options 634
 - Twin Builder target options 652
- Record scenario
 - Simulink target options 661
- Rectangles
 - drawing 298
- Redefined types
 - simulating 684

- Redo 59
- Reference
 - Project properties (Suite) 425
- Refresh 533
- Reload last workspace at startup
 - tools options 76
- Remove an Input from a Predefined Operator 236
- Remove else 187
- Remove Focus (Suite) 421
- Remove from Equation Set 255
- Remove from Source Control 933, 936
- Remove then 187
- Renaming projects 102
- Replace
 - in Text Diagrams 263
 - in Textual Views 263
- Replace All
 - in Text Diagrams 263
- Replace in
 - Text Diagrams 263
- Replace with
 - in Text Diagrams 263
- Report files 65
- Reporter
 - setting active configuration 120
- Reports
 - Design Verifier analysis 724
- Requirements window
 - in SCADE Suite 38

- restart every 408
- Resume transition 174
- Root operator 837
 - for simulation code generation 587
- Rotate Left 290
- Rotate Right 290
- Rotate toolbar 48
 - using 290
- Rounded rectangles
 - drawing 298
- RTOS Adaptors 6
- Run-time checks
 - setting for simulation 584
- Run-time exceptions 597

S

- Same Height 289
- Same Size 289
- Same Width 289
- Sample time 669
- Save 57
- Save All 57
- Save As 57
- Save as Template 58
- Save before running tools
 - tools options 75
- Save options 75
- Save Workspace 57
- SCADE
 - setting tools options 74
- SCADE files 62
- SCADE Suite

Index

- command lines 915
- creating new project 96
- launching on Windows 8
- organizing in packages 73
- project management 73
- SCADE Display integration 5
- used files 60
- Viewer mode 8
- SCADE Suite Design Verifier 3
- SCADE Suite Editor 2
 - Diagram Views 44
 - Editing views 40
 - packages 124
 - status bar 50
 - Textual Views 43
 - toolbars 45
 - user interface overview 28
- SCADE Suite environment
 - toolset content 2
- SCADE Suite environment variables
 - \$(CG) 837
 - \$(Configuration) 837
 - \$(NodeName) 837
 - \$(TargetDir) 844
- SCADE Suite files
 - adding automatically under SCC 931
 - in Source Code Control 928
- SCADE Suite Gateways
 - for VeriStand 5
- SCADE Suite interface
 - Diff View 40
 - Output Window 38
- Properties window 37
- Requirements window 38
- Shortcuts window 39
- SCADE Suite KCG
 - code generator for certification activities 4
- SCADE Suite launch
 - command line 917
- SCADE Suite models
 - change control 525
 - project and workspace files 60
- SCADE Suite projects
 - inserting textual definitions 270
- SCADE Suite Simulation 3
 - schema of operating procedures 570
- SCADE Suite Stack Verifier 4
- SCADE Suite Timing and Stack Optimizer 3
- SCADE Suite Timing Verifier 4
- SCADE Suite tools
 - showing advanced options 76
- SCADE Suite toolset
 - Configuration Management Gateway 6
- Scade View 34
 - cut/paste elements 280
 - deleting declarations 282
 - key symbol meaning 34, 130
 - opening editing views 41
 - presentation 30
- Scalar
 - pragma 863
- SCC 76, 927
 - checked-in item 934
 - checked-out item 934
 - file history 937
 - importing files stored in SCCS 938
 - showing file properties 938
 - showing version differences 937
- SCC commands
 - using from SCADE Suite 932
- SCC status
 - color checkmarks 934
- SCC System (Suite)
 - sending commands 933
- Scenario pathname
 - FMU target options 634
 - Simulink target options 661
 - Twin Builder target options 652
- Scenario Viewer 727
- Scripts
 - command line 920
- Search
 - in models (Suite) 410
- Select All 283
- Select Mode 283
- Selected
 - debug options 841
 - expansion mode 848
 - I/O separation mode 858
- Selection
 - area 283
 - by adding 283
 - by intersection 283

Index

- small symbols 283
- Semantic check
 - error files 63
- Semantic checks 828
- Semantic differences
 - comparing models 525
 - examining 532
 - reporting of analysis 536
- Semantic error file
 - project creation 99
 - Project properties (Suite) 426
- Semantic error files 63
- Semantics checking 274
- Send Backward 301
- Send to Back 301
- Sensor 230
- Sensors View
 - using 42
- Sensors Views
 - purpose and user tasks 42
- Separate I/Os
 - pragma 862
- Separation mode
 - separating I/O for all operators 857
 - separating I/O of selected operators 858
 - setting operators not marked for I/O separation 858
- Set Active 120
- Set Active Configuration 120
- Set as Active Project 102
- Set symmetric 235
- Set transition kind as strong 173, 176
- Set transition kind as synchro 173, 176
- Set transition kind as weak 173, 176
- Set transition layout as Bezier 173, 176
- Set transition layout as Polyline 173, 176
- Set warnings as errors 838
- Setting SCADE Suite environment 73
- S-Function 667
- S-Function creation 663
- S-Function dynamic library
 - creating 664
- Short circuit operators (Ada code) 845
- Short circuit operators (C code) 843
- Shortcuts
 - use of 73
- Shortcuts Window
 - showing/hiding 29
 - using 131
- Shortcuts window
 - in SCADE Suite 39
- Show
 - pragmas in Text Diagrams 264
- Show advanced options page
 - tools options 76
- Show Differences 933, 937
- Show Grid 293
- Show History 933, 937
- Show in Browser 419
 - I/Os connected to graphical panels 312
- Show in Folder 832
- Signal
 - insert 167
- Signals
 - creating presence test 241
 - declaring in transition scope 179
 - designing conditional emission 241
 - emission of 26
 - emitting 241
 - fundamentals about 24
- Significance length 856
 - C code generation 843
- Significant digits
 - project simulation property 585
- Simulating in FMI-compliant tools
 - setting black-box co-simulation 635
 - setting source code export 636
 - setting white-box co-simulation 635
- Simulating in Twin Builder
 - setting black-box co-simulation 653
 - setting white-box co-simulation 653
- Simulation

Index

- breakpoint definition 573
- data-driven 571
- displaying colored graphs 584
- enabling partial simulation 593
- executables 572
- GUI-driven 571
- ignoring violated assertions 584
- interactive scenario-driven 571
- logging violated assertions 584
- observable variable 575
- optimizing graph display 584
- SCADE Suite architecture 570
- scenario-driven 571
- setting backward limit 584
- setting observability file 607
- setting output file 606, 607
- setting project options 585
- setting run-time checks 584
- setting tool options 583
- stopping on violated assertions 584
- traceability and observability 571
- using imported types 675
- workflow with graphical panels 625
- Simulation (Ada)
 - using imported operators 685
- Simulation (C)
 - using imported operators 673
- Simulation API
 - main functions 688
- Simulation architecture
 - fundamentals 570
- Simulation code
 - setting basic generation options 587
 - setting compiling options 589
- Simulation cycle
 - fundamentals 572
- Simulation executable
 - fundamentals 572
 - generating 597
- Simulation scenarios
 - IN files 614
 - SNS files 614
 - SSS files 614
 - STS files 614
- Simulation Views 576
- Simulation window 577
- Simulation with Simulink
 - launching in black-box 665
 - launching in white-box 666
- Simulator
 - cosimulation mode 687
 - Diagram Simulation View 576
- Simulator Extension toolbar 579
- Simulator scenarios
 - creating 614
- Simulator toolbar 579
- Simulink
 - setting target options 660
- Simulink code wrapping
 - setting 659
- Skip disabled model objects 593, 838
- Skip unused model objects 838
- Skip unused operators 593
- Small Icons
 - in shortcuts list 132
- Snap to Grid 293, 296
- Snap to grid 152
- Snapshot scenarios
 - creating 614
- Source Code Control 927
 - and SCADE files 928
 - enabling (Suite) 930
 - integration principles in SCADE Suite 928
 - introduction 927
 - setting options 931
- Source Code Control System 76
 - connecting option 76
- Source Control 934
 - adding/removing 935
- Source Control Properties 934, 938
- Source Control toolbar
 - in SCADE Suite 933
- Source File for Ada 433
- Source File for C 433
- Source Code export
 - setting FMU target 636
- Space Across 288
- Space Down 288
- SPARK Hide
 - pragma 863

Index

- Specialization Operators
 - declaring 197
 - Specialize
 - declaring specialization operators 197
 - declaring specialized operators 197
 - Specialized Operators
 - declaring 197
 - Split 205
 - Standard toolbar (Suite) 46
 - Star symbol (*) 727
 - Staretegies
 - fundamentals 734
 - State
 - border of hidden display 143
 - setting display mode 472
 - setting pragmas 861
 - State actions
 - defining textual body 473
 - State display
 - setting in embedded mode 200
 - setting in hidden mode 200
 - switching graphical/textual mode 266
 - State Machine toolbar 47
 - State Machines
 - defining textual states 266
 - designing 168
 - State machines
 - in parallel 169
 - State scenarios
 - creating 614
 - States
 - creating transitions 172
 - editing textual mode 268
 - embedding display 200
 - encapsulating 252
 - freezing execution of 24
 - fundamentals 22
 - hiding display 200
 - set as final 172
 - set as initial 172
 - setting as initial/final 171
 - Status bar 50
 - Status messages
 - Consistent 759
 - Falsifiable 759
 - Inconsistent 759
 - Indeterminate 759
 - Interrupted 759
 - Raised an Error 759
 - Valid 759
 - Stop simulation
 - failed assertion option 584
 - Strategy
 - fundamentals 734
 - Strong transition
 - setting 173, 176
 - Structure and array operators
 - toolbar 47
 - Structure/Array toolbar 47
 - Structured types
 - composing/decomposing 236
 - Style toolbar 48
 - Styles
 - applying from properties 482
 - applying from toolbar 139
 - applying in diagrams 138
 - creating 73
 - creating own styles 135
 - default or user-defined 134
 - deleting user-defined 142
 - modifying 73, 141
 - Sub-title
 - Project properties (Suite) 425
 - Switching perspectives 93
 - Symbol Drawing 298
 - Symbol Drawing toolbar (Suite) 48
 - Symbol Editor
 - introduction 50
 - using 294
 - Symbol files 64
 - Symbol Views
 - setting grid 152
 - Symbols
 - external fonts 65
 - Symmetric
 - operator call properties 449
 - Synchro transition
 - setting 173, 176
 - Syntax check
 - in text Diagrams 264
 - System Synchronization
 - setting configuration 120
- ## T
- Target
 - selecting FMU 633
 - selecting Simulink 659

Index

- selecting Timing and Stack Optimizer (for MCG) 907
- selecting Twin Builder 651
- Target directory 837
- Task
 - pragma 863, 900
 - setting pragmas 863
- TCL Script toolbar (Suite) 49
- Temporal properties
 - fundamentals 730
- Text Box 299
- Text box
 - editing 299
 - inserting 299
- Text Diagram 260
- Text diagrams
 - navigating 265
- Textual actions
 - editing definition for If Nodes 465
 - editing definition for When Branch 469
- Textual definitions
 - inserting SCADE file 270
- Textual diagrams
 - applying changes 265
 - checking syntax 264
 - creating 260
 - editing 261
 - jumping to lines 264
 - marking lines 263
 - navigating from 265
 - replacing strings 263
 - searching strings 262
- Textual nodes
 - in Textual Views 43
- Textual states
 - editing definition 473
- Textual states/actions
 - defining 266
 - editing 268
- Textual Views
 - purpose and user tasks 43
- Time toolbar 48
- Time-related operators
 - toolbar 48
- Timing and Stack 118
- Timing and Stack Tools toolbar 788
- Timing and Stack Verifiers
 - setting active configuration 120
- Title
 - Project properties (Suite) 425
- To Shortcut 131
- Tool execution
 - status 50
- Tool Options
 - Common options - Source Code Control System 76
- Toolbar (Suite)
 - Higher Order 48
 - Structure and Array 47
- Toolbar buttons
 - help 50
- Toolbars
 - adding in shortcuts list 131
- Browse 49
- CG-Simulator 829
- Choice 48
- customizing 73
- Design Verifier 722
- display in shortcuts 131
- removing from shortcuts list 131
- shortcuts list in large icons 132
- shortcuts list in small icons 132
- showing/hiding 45
- Simulator 579
- Simulator Extension 579
- Timing and Stack Tools 788
- Toolbars (Suite)
 - Bitwise 47
 - Comparison 47
 - Create 47
 - Find 49
 - Go To 49
 - Layout 48
 - Logical 47
 - Mathematical 47
 - Metrics and Rules Checker 497
 - Navigate 49
 - Nudge 48
 - Perspective 46
 - Project 46
 - Rotate 48
 - Source Control 933
 - Standard 46
 - State Machine 47
 - Style 48
 - Symbol Drawing 48

Index

- TCL Script 49
- Time 48
- Zoom/Pan 48
- Tools
 - setting options 74
- Tooltip help
 - in SCADA Suite 12
- Top level package 845
- Top-level imported package 845
- Top-level operator 730
- Traceability
 - observable variables in simulation 571
- Transitions
 - adding tangents 292
 - creating 172
 - creating fork transitions 175
 - defining action 179
 - defining condition 179
 - fork set as else branch 177
 - fundamentals 23
 - moving 174
 - reconnecting 174
 - removing tangents 292
 - resuming 174
 - set as strong 173, 176
 - set as synchro 173, 176
 - set as weak 173, 176
 - setting default creation modes 178
 - setting layout 173, 176
- Truncated info
 - underlining 149
- Twin Builder
 - generated files for co-simulation 654
 - setting operator properties 650
 - setting target options 652
- Twin Builder version
 - Twin Builder target options 652
- Type 208
- Type definition
 - navigating to declaration 419
- Type Definition File 433
- Type initializer
 - pragma 862
- Type redefinition
 - simulating 684
- Typed flows
 - composing from output variables 237
 - decomposing from input variables 238
- Types
 - propagating along data flow 247
 - setting generic 216
 - setting init pragmas 862
 - setting pragmas 862, 863
- Types View
 - using 43
- Types Views
 - purpose and user tasks 43
- U**
- Undo 59
- Undo check out 933
- Unit
 - in operator Twin Builder properties 650
 - root operator periodicity 447
 - simulation execution option 852
- UNIX text format
 - saving files 114
- Unroll
 - pragma 863, 904
 - setting pragmas 863
- Unset symmetric 235
- Up
 - Go To toolbar 417
- Use a creation dialog box
 - tools options 77
- Use properties
 - higher-order patterns 449
 - symmetric 449
- Use type clauses 846
- User config 682, 844
 - for mobile generation 709
- User preferences
 - in EWO file (Suite) 60
- User Styles
 - creating styles 136
- User-defined operators
 - in Shortcuts lists 131
- User-provided sensors
 - declaration 851
 - Simulink target options 660
- V**
- Valid message

Index

- fundamentals 726
- reading result 759
- Variables
 - creating local references 240
 - highest visibility 594
 - simulation observability 571
- Verification 724
- Verification fundamentals
 - predefined proof objectives 734
 - property expression 730
 - status messages 758
 - strategies 734
 - user proof objectives 732
- Viewer mode
 - in SCADE Suite 8
- Views
 - char array representation 210
 - closing active view 148
 - displaying and setting 73
 - duplicating 147
 - zooming content 148
- Visibility
 - outside packages 130
- VSW files (Suite) 60

W

- Warn on change of project file status
 - tools options 76
- Watch window 577
- Weak transition
 - setting 173, 176
- When blocks
 - defining textual actions 266
 - designing 188
 - moving 192
 - setting clock of 190
- When branch
 - creating 190
 - deleting 194
 - setting 191
 - setting action of 192
- When Branch actions
 - defining textual body 469
 - setting display mode 468
- White-box co-simulation
 - launching 666
- White-box simulation
 - setting FMU target 635
 - setting Twin Builder target 653

Workspace

- inserting existing item 206
- inserting existing projects 112
- opening projects 54
- searching in models (Suite) 410
- switching active project 102

Workspace files (Suite) 60

Workspace options 76

Wrap C operators 683, 843

X

XSCADE file 31

Z

Zoom 149

- zooming on design 149

Zoom to Fit 149

Zoom to Selection 149

Zoom/Pan toolbar 48

Zooming

- continuous 149
- in steps 149
- magnification factor 149
- on specific element 149
- to fit content 149

Zooming design pages 149