Ansys

Technical Manual

SCADE Suite Technical 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 Technical Manual – Published April 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: 29/04/24 **Revision**: SCS-TM-24 - DOC/rev/63004-04

Technical Manual Overview

This manual provides advanced users with technical background focusing on the intricacies of SCADE Suite tools.

"Introduction"

Part 1: "Editor"

- Chapter 1: "Annotation Mechanism"
- Chapter 2: "Editor Save Format"
- Chapter 3: "Specific SCADE Suite Files"
- Chapter 4: "Scripting Capabilities in SCADE Environments"

Part 2: "Simulator"

- Chapter 5: "Dedicated Simulation Functions"
- Chapter 6: "SCADE Suite Simulation Batch"
- Chapter 7: "Data Format for SCADE Suite Simulation"
- Chapter 8: "Simulation Command Line Errors"

Part 3: "Code Generator"

- Chapter 9: "Code Production Using SCADE Suite Code Generators"
- Chapter 10: "Multicore Target Integration of Task-Based Code"
- Chapter 11: "Code Generators Command Line Options"
- Chapter 12: "Code Generation Messages"

"Index"

RELATED DOCUMENTS

- *SCADE Suite User Manual*
- *Scade Language Reference Manual*
- *SCADE Python API Guide*
- *SCADE Java/Tcl API Guide*

TYPOGRAPHICAL CONVENTIONS

Bold	GUI buttons, options, tabs, radio buttons, drop-down lists, window titles, panels.
Italics	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

Technical Manual Overview

Introduction 1

Part 1: Editor

1. Annotation Mechanism 1 - 5

Fundamentals about Annotations 1 - 5

Objects Supporting Annotations 1 - 6

Inlaid Annotations 1 - 11

Propagation in Generated Code 1 - 12

Data Identification 1 - 12

Annotation Storage Principles 1 - 13

Annotation Service Functions 1 - 14

Note Handling Functions 1 - 14

Note Type Handling Functions 1 - 15

Checking Functions 1 - 17

Loading/Saving Functions 1 - 17

Storage File Formalism 1 - 18

Annotation Type Files (ATY) 1 - 20

Annotation Files (ANN) 1 - 25

List of Predefined Properties 1 - 27

Kernel Recognized Properties 1 - 27

Properties Recognized in Annotation Editing Window 1 - 28

Inlayable Annotations Predefined Attributes 1 - 29

Printable Annotation Predefined Attributes 1 - 31

2. Editor Save Format 2 - 33

Save File Format 2 - 34

Pragmas 2 - 35

Table of Contents

UML Diagram 2 - 36

Graphical Information Description 2 - 38

Data Flow Graphical Representation 2 - 42

Equation (Box) 2 - 42

Assertion (Box) 2 - 45

Data Flow (Edge) 2 - 45

Control Flow Graphical Representation 2 - 47

State Machine (Box) 2 - 47

State (Box) 2 - 47

Fork (Box) 2 - 48

Transition (Curve) 2 - 48

If Block (Box) 2 - 50

If Node (Box) 2 - 50

If Action (Box) 2 - 50

When Block (Box) 2 - 50

When Branch (Box) 2 - 50

When Action (Box) 2 - 50

XML Save Format of Graphical Elements 2 - 51

Data Flow Graphical Elements 2 - 52

Control Flow Graphical Elements 2 - 55

Specific Information 2 - 64

3. Specific SCADE Suite Files 3 - 65

Symbol Files 3 - 65

Lexical Aspects and Conventions 3 - 66

Language Syntax and Graphical Representation 3 - 69

Naming Rules 3 - 70

Search Rules 3 - 70

Creation Rules 3 - 71

Option Files 3 - 71

Table of Contents

4. Scripting Capabilities in SCADE Environments 4 - 73

SCADE Scripting Capabilities 4 - 73

Registering Customization Scripts 4 - 77

Registering Scripts from User Interface 4 - 77

Registering Scripts in Batch Command 4 - 78

Registering Customization Scripts from File 4 - 78

Removing Customization Scripts 4 - 78

Custom Script Example 4 - 79

Part 2: Simulator

5. Dedicated Simulation Functions 5 - 83

Simulator Commands 5 - 83

Simulation-Specific Commands 5 - 84

Simulation-Specific Variables 5 - 105

Simulator Tcl Extension 5 - 106

Conversion and Comparison Functions for Types 5 - 107

Basic Use Cases 5 - 108

Structure of VTable for C Code 5 - 109

Structure of VTable for Ada Code 5 - 111

Definition of SimConvType (C only) 5 - 112

Defining VTables in C Source File 5 - 113

Defining VTables in Ada Source File 5 - 118

Examples of Function Definitions 5 - 121

Conversion Functions for Imported Operators 5 - 129

Structure of VTable for C Code 5 - 130

Structure of VTable for Ada Code 5 - 131

Defining VTables in C Source File 5 - 131

Defining VTables in Ada Source File 5 - 133

Examples of Function Definitions 5 - 135

Table of Contents

Public API Functions for Co-Simulation Mode 5 - 138

Description of API Functions 5 - 138

API Function Errors 5 - 143

Example of Enriched Main Function 5 - 144

6. SCADE Suite Simulation Batch 6 - 149

Simulation Modes 6 - 149

Running Simulation Executable 6 - 151

7. Data Format for SCADE Suite Simulation 7 - 155

Simulation Value Format 7 - 156

Simulation Path Format 7 - 159

Simulation Input Files 7 - 162

Format of SNS Files 7 - 163

Format of STS Files 7 - 163

Format of SSS Files 7 - 163

Format of IN Files 7 - 164

Simulation Output Files 7 - 166

Format of OUT Files 7 - 166

Format of CSV Files 7 - 167

Format of OBS Files 7 - 168

Example with Simulation Scenario Files 7 - 170

8. Simulation Command Line Errors 8 - 173

Error Codes 8 - 173

Error Messages 8 - 176

Part 3: Code Generator

9. Code Production Using SCADE Suite Code Generators 9 - 179

General Overview 9 - 180

Table of Contents

Input and Output Files 9 - 182
Schema of Input/Output Files 9 - 183
User-Provided Input Files 9 - 185
Generated Files 9 - 187
Description of Generated Information Files 9 - 191
Description of Generated Files for C Code 9 - 194
Standard Generated Files in C 9 - 195
Additional Generated Files in C 9 - 215
File Inclusion Protection 9 - 220
Description of External Files for C Code 9 - 221
Description of Generated Files for Ada Code 9 - 222
Standard Generated Files in Ada 9 - 223
Additional Generated Files in Ada 9 - 227
Description of External Files for Ada Code 9 - 228
Traceability from Specification Models 9 - 229
Traceability Information 9 - 229
Package Hierarchy and SPARK Compliance 9 - 234
Mapping File 9 - 239
Pragma Syntax and Usage 9 - 241
General Pragmas 9 - 241
Ada-Specific Pragmas 9 - 255
C-Specific Pragmas 9 - 257
MCG-Specific Pragmas 9 - 258
Sequence of Pragmas 9 - 259
Pragmas by Code Generator Version 9 - 260

10. Multicore Target Integration of Task-Based Code 10 - 261
Traceability and Mapping File 10 - 262
Method Splitting 10 - 263

Table of Contents

Implementation of Communication Channels 10 - 265
Implementation with Semaphores 10 - 266
Implementation with Condition Variables 10 - 267
Special Implementation Cases 10 - 268
Implementation with Initial Token 10 - 269
Local Communications 10 - 270
Advanced Multicore Implementation 10 - 271
Clocked Tasks 10 - 271
Tasks and Code Expansion 10 - 274
Channel Optimization 10 - 274
Pipelining 10 - 275
Examples of Integration 10 - 276

11. Code Generators Command Line Options 11 - 283
KCG/MCG Execution and Compatible Options 11 - 284
List of Available KCG/MCG Options 11 - 286
Description of General Options 11 - 290
Description of Optimization Options 11 - 294
Description of Debug Options 11 - 297
Description of Expansion Options 11 - 301
Description of I/O Separation Options 11 - 308
Description of Configuration Options 11 - 311
Description of Additional Options 11 - 332

12. Code Generation Messages 12 - 333
Messages from Model Checks 12 - 333
Syntactic/Post-Synctactic Check Errors/Warnings 12 - 334
Namespace Check Errors 12 - 335
Type Check Errors/Warnings 12 - 337
Clock Check Errors 12 - 340

Table of Contents

Causality Check Errors/Warnings 12 - 341
Initialization Check Errors 12 - 341
Monomorphisation Check Errors/Warnings 12 - 342
Messages from Additional Checks 12 - 342
Messages from XML Checks 12 - 347
Failure Messages 12 - 347
Command Line Messages 12 - 348

INDEX 353

Introduction

This manual provides you with technical information about SCADE Suite ® customization, file formats and other useful back-end data.

It is intended for the advance use and customization of SCADE Suite environment. For any basic feature description, user instructions or tips, please consult the contents of *SCADE Suite User Manual* from "User Manual Overview".

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

- "Editor"
- "Simulator"
- "Code Generator"

For a technical introduction to each of these modules, read about "SCADE Suite Environment Toolset" on page 2 in *SCADE Suite User Manual*. Additional manuals and guidelines cover the rest of SCADE Suite modules and more.

Ansys

Technical Manual

Part 1

Editor

- Chapter 1: "Annotation Mechanism"
- Chapter 2: "Editor Save Format"
- Chapter 3: "Specific SCADE Suite Files"
- Chapter 4: "Scripting Capabilities in SCADE Environments"

1 / Annotation Mechanism

This chapter defines and explains the principles, the mechanisms and the formalism used by SCADE Suite to help annotate models. You can also check all the properties that are defined in dedicated files to support annotation service.

- "Fundamentals about Annotations"
- "Annotation Storage Principles"
- "Annotation Service Functions"
- "Storage File Formalism"
- "List of Predefined Properties"
- "Inlayable Annotations Predefined Attributes"
- "Printable Annotation Predefined Attributes"

Fundamentals about Annotations

An annotation is a note associated with an object of the model and is used to add extra information to the model. Annotations are structured information. Their structure is defined by note types. Some annotations can be inlaid in the graphical views: they are called inlaid annotations. The list of annotable objects is provided in "Annotable objects" on page 6.

The content of annotations can be created and edited from the **Note** property page as explained in "Adding and Editing Notes" on page 482 of *SCADE Suite User Manual*.

- "Objects Supporting Annotations"
- "Inlaid Annotations"
- "Propagation in Generated Code"
- "Data Identification"

Objects Supporting Annotations

The following figure illustrates in a class diagram what kind of SCADE Suite objects can be handled for annotation purposes.

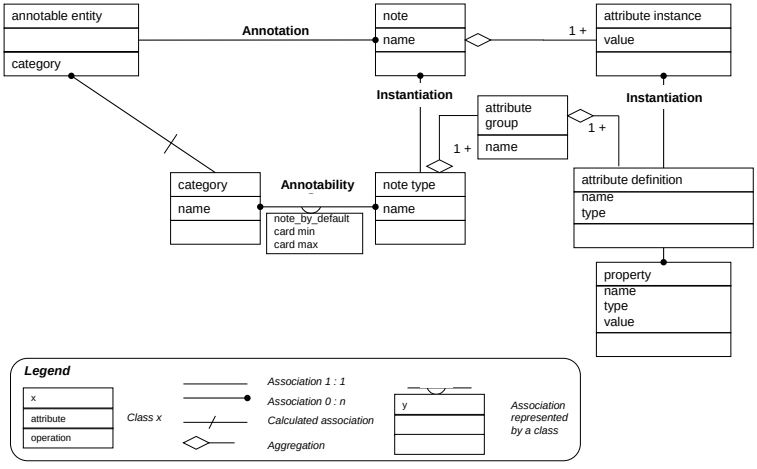


Figure 1.1: Class diagram representing objects in SCADE Suite

Annotable objects

An annotable object can contain any number of annotations. The associated class defines a *category* function which returns the object category. Inlayable annotations are only visible in graphical diagrams. Annotable objects in SCADE Suite environment are:

Operators	State Machines	If blocks and When blocks	Assertions
Local variables, signals and probes	States	If nodes and When branches	Enumeration value, structure element, and user types
Input, hidden input, and output	Transitions	Actions	Constants
Equations	Packages	Textual and graphical diagrams	Sensors

Objects and associations

The objects in SCADE Suite environment are divided into the following categories:

- Objects representing the structure of an annotation: “note type”, “attribute group”, “attribute definition”, “property”, and “category” types.
- Objects representing instances of annotations: “note” and “attribute instance”.

“Note type” Object

A note type defines the structure of annotations and consists of a list of attribute groups with a name.

“Attribute group” Object

This object is used to group the attributes together in a category. For example, displayed information related to inlayable annotations is stored in the attributes of an inlay group and the content to be displayed is stored in the attributes of a content group.

“Attribute definition” Object

An attribute is defined by a name and a type. The attribute types are: integer, real, Boolean, character, string, text, date, enum, file.

- The **`string`** and **`text`** type attributes contain character strings of any length. Their display characteristics are different; a `string` type attribute is represented by a field on a single line, whereas a `text` type attribute is represented by a field on several lines.
- A **`file`** type attribute contains the reference to a file. A **`text`** type attribute is a printable character string of any length. The content of the referenced file for a **`file`** type attribute is not stored in the note.

"Property" Object

Properties apply to an attribute definition. A property is defined by name, type and value. The property types are those of the attribute definitions. The categories of property are:

- Properties recognized by the annotation service and used, for example, to express restrictions on the attribute value or assign a default value to an attribute.
- Properties recognized by the annotation service UI and used, for example, to indicate whether an attribute can be edited or displayed.
- Properties recognized by the application such as inlaid annotations. This way, the application can define properties that are not processed by the kernel and the User Interface.

The distinction between such categories comes from the name of the property. There are also predefined properties recognized by the annotation service and the annotation service UI (see "Annotation Service Functions" on page 14). New properties can also be defined in SCADE Suite.

"Category" Object

This object represents a category of annotable objects. Categories available in SCADE Suite are listed below in Table 1.1. Moreover, each operator can have a category attached with its definition. This category is used for calls to this operator.

“Annotability” Association

This association defines the relationship between a category of annotable object and a note type. Several notes of different types can be attached to an object of a given category. The Boolean `default_note` attribute indicates that a note must be automatically created when an object of the given category is created. This note is created with the default values given by `NT_DEFAULT_VALUE` type properties (see “Note type definition module” on page 20).

The `card_min` and `card_max` attributes are used to indicate the minimum and maximum cardinality, namely the minimum and maximum number of notes of the type given by the association for an annotable object of that category. You can specify maximum cardinality by -1 to indicate that any number of notes is accepted. For example, for an annotable object of `operator` category and for a `type1` note type, if maximum cardinality is 1, it is only possible to create one `type1` type annotation for an object of `operator` category. When creating/deleting annotations, the command stops if cardinality limits are incorrect.

“Note” Object

An annotable object can have any number of notes. Each note is an instantiation of a note type: it contains a list of attribute instances and a name (unique to the annotated object).

“Attribute instance” Object

An attribute instance is an instantiation of an attribute and contains a value. The attribute instances of a note must correspond to the attribute of the related note type (same type and same property).

Table 1.1: Categories and their corresponding objects

Category	Object
`package`	Package
`constant`	Constant
`type_element`	Type element (for structure element and enumerated values)
`input`	Input, hidden input
`output`	Output
`probe`	Probe
`local`	Local
`signal`	Signal
`sensor`	Sensor
`named_type`	Named type
`operator`	Operator
`assertion`	Assertion (assume, guarantee)
`state`	State
`state_machine`	State machine
`transition`	Transition (transition and fork transition)
`activate_block`	If block and When block
`branch`	If node and When branch
`action`	Action
`predef_opr, ref_operator, ref_ident, literal, terminator`	Equation
`net_diagram`	Graphical diagram
`text_diagram`	Textual diagram
`equation_set`	Equation set

Inlaid Annotations

Inlaid annotations are annotations displayed in graphical views. The types of inlayable attributes supported depend on the environment. Such annotations have the following characteristics:

- The position of inlaid annotations can be fixed or modifiable:
  - If the position is fixed, annotations of a given type have the same position in any given view of SCADE Suite. It is impossible to move them.
  - If the position is modifiable, users can move the annotation by clicking its box. The new position is saved with the graphical view.
- The size of the box containing an inlayable annotation is either set (given size at note type creation) or calculated according to its content.

Storing display information

Display information is stored in a different attribute group as the one that defines the content of the annotation to be set. All attributes defined below must be defined for a view in an inlay attribute group:

Position of Annotations

The position of an annotation is defined by two attributes, `NT_POS_X` for the position along the x-axis, and `NT_POS_Y` for the position along the y-axis. This position indicates the coordinates of the upper left-hand corner of the rectangle containing the annotation. These are logical coordinates that are independent of the screen size.

Positioning Mode

The positioning mode is stored in an attribute called `NT_LOCATION`. It can be relative or absolute. Absolute positioning is used to position the annotations independently of the design elements they annotate.

Annotation Mobility

The Boolean value indicating whether the position is modifiable or not is stored in the `NT_MODIFY` attribute.

Font Attributes

See details in "Inlayable Annotations Predefined Attributes" on page 29.

Annotation Inlaying

An attribute used to indicate whether the annotation can be inlaid or not.

Permanent Inlaying

An attribute used to indicate if the annotation is still inlaid (consult "Enabling Display Options in Diagram Views" on page 141 in *SCADE Suite User Manual*).

Propagation in Generated Code

Annotations can be propagated in the generated code. If so, they appear as comments in the generated code. A propagated annotation type has the Boolean attribute **to_c**, belonging to the "information" group. This attribute defines whether a note is propagated in the generated code or not.

Data Identification

The name of a note is unique to the object it annotates. It is a character string given by the user containing any printable ASCII character. The name of a note type is unique within the annotation environment. The name of a note type is an identifier (case sensitive). An identifier is a sequence of alphanumeric and "_" characters.

Annotation Storage Principles

The annotation storage mode is explained below and illustrated in the following figure. Annotations are logically linked to the annotated model.

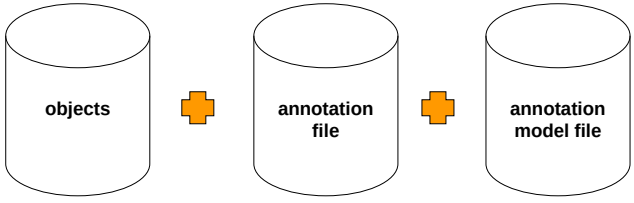


Figure 1.2: Modes of annotation storage

- All annotations of annotated objects are stored in a file called **annotation file** (`.ann`). Loading (respectively saving) is performed immediately after loading (respectively saving) these annotated objects. Storage format is ASN1 (Abstract Syntax Notation Number One). See "Annotation Files" on page 63 in *SCADE Suite User Manual*.
- The annotation types (note types and annotability relations) are stored in a file called **annotation type file** (`.aty`) which can be modified by users. See "Annotation Type Files" on page 63 in *SCADE Suite User Manual*.

A model is associated with one or several annotation type file(s). All annotations of a given object saved in a SCADE file named <XYZ>.xscade are saved into an annotation file named <XYZ>.ann and stored in the same directory.

Annotation Service Functions

The main annotation service operations are divided into the following groups:

- "Note Handling Functions"
- "Note Type Handling Functions"
- "Checking Functions"
- "Loading/Saving Functions"

It is possible to access such service functions via API as described in "Tcl Scripts from Metamodel Diagrams" on page 54 from *SCADE Java/Tcl API Guide*. About annotation metamodels, consult "SCADE Suite Metamodels for Tcl API" from *SCADE Metamodels Technical Reference Card*.

Note Handling Functions

The table below summarizes all functions available to manage notes in SCADE Suite.

Table 1.2: List of note handling functions

Function	Description
Creating/adding notes on annotable object	This function creates a note by instantiating a note type and creates an annotation relationship between the note and the annotable object if the maximum cardinality limit is respected.
Deleting annotations	This function deletes the note and the annotation relationship from an annotated object if the minimum cardinality limit is respected.
Editing annotations	This function is used to modify the attribute instance values of a note and the note name.

For creating notes, see "Adding and Editing Notes" on page 482 in *SCADE Suite User Manual*.

Note Type Handling Functions

A note type describes the structure of notes.

- There is no User Interface for functions handling the note types. Any modification must be entered in the file that contains annotation types.
- Several note type files can be associated with a project. The files are merged and the following rules must be fulfilled:
  - A note type cannot be declared more than once.
  - The association ability of an object for type of note can be specified only once.

The table below describes all functions available to manage note types in SCADE Suite.

Table 1.3: List of note type handling functions

Function	Description
Adding note types	You must add the description of the note type as well as the annotability relationships.
Deleting note types	If there are annotations of the deleted type, the user is warned when loading annotations of this type. It is not possible to display or edit the content of these notes. It is only possible to delete them. Such notes are kept when the next save is performed.

Table 1.3: List of note type handling functions (Continued)

Function	Description
Modifying note types	The user can add, delete, rename, or change the type of an attribute definition. • When an *attribute is deleted*, at next save, the notes are kept and the value corresponding to the attribute is deleted. The note is saved with the new schema. • When an *attribute is added*, at first load, a new value is created for the new attribute. This value is the default value property if it exists, or a default value. The note is saved with the new schema. • When an *attribute is renamed*, all remains as if the attribute was deleted and a new one was created. • When an *attribute type is modified*, you want the conversion to be applied if possible, otherwise it creates a new value. The note is saved. • When the *property of an attribute is modified*, the property is taken into account at next load except in case of default value. • If an *annotability relation is modified* by addition, deletion, default value, or cardinality, a warning displays when cardinality is exceeded. The note is still visible, it can be deleted but not edited; its content is not displayed.
Note type compatibility	This function checks that both types of note and notes are compatible. • Let TM be the model file note type or the note type. • Let TA be the type of note read in the annotation file. The TM type must contain at least all the attributes of TA. These attributes must have the same characteristics (same type and same property).

When loading annotation types, if there is an error, no type can be loaded. For associating annotation type files with projects, see "To edit the SCADE properties of a project" on page 424 in *SCADE Suite User Manual*.

Checking Functions

The table below summarizes all functions available to check notes in SCADE Suite.

Table 1.4: List of checking functions

Function	Description
Checking cardinality limits	This function verifies whether the cardinality limits are respected at any given moment. A warning is displayed when cardinality is exceeded. The note is still visible, it can be deleted but not edited; its content is not displayed.

Loading/Saving Functions

The table below summarizes all functions available to load/save notes.

Table 1.5: List of loading/saving functions

Function	Description
Saving annotations	This function is used to save in an annotation file all annotations sent by SCADE Suite. It is used by SCADE Suite when saving the model. It saves annotations in the file in the format indicated in "Storage File Formalism" on page 18.
Loading annotation types	This function loads all annotation types of the file(s) specified in the project parameter. The function loads the note types and the annotability relationships of the annotation model file(s). If an error is detected during loading, the command fails and no types is loaded.
Loading annotations	This function loads the annotations of the files. If an error is detected during loading, the command fails and no annotation is loaded. If cardinality limits are not respected, a warning message is displayed and the command fails.

Storage File Formalism

The annotations and note types are stored in the following types of file:

- **Annotation type files** (*.aty* format) contain note types and annotability relations. Such files can be edited by users (if access rights allow). See “Annotation Type Files (ATY)”.
- **Annotation files** (*.ann* format) contain annotations associated with a set of design objects defined by SCADE Suite. Such files cannot be edited by users. See “Annotation Files (ANN)”.

ANNOTATION FILES GENERAL SYNTAX

The language grammar is described below. Storage formalism is based on the CSN1 format (Concrete Syntax Notation One). It is both a format and a data exchange software service (based on ISO OSI standards, ASN1 (ISO 8824) in particular). The file syntax is specified by Extended Backus-Naur Forms (EBNF). Its specification is a set of production rules referred to as the “grammar”. Each rule consists of a non-terminal symbol and an EBNF expression separated by equal sign and terminated by period. The non-

terminal symbol is defined by the EBNF expression. The EBNF expression is formed by any number (even zero) of terminal signals, non-terminal symbols, and metasymbols identified below.

Table 1.6: Meaning of metasymbols used in annotation files

Metasymbols	Meaning
=	is defined to be
|	or
.	end of the production rule
[X]	0 or 1 occurrence of X
{X}	0 or more occurrences of X
(X|Y)	group: X or Y
“XYZ”	terminal symbol XYZ
identifier	identifier non-terminal symbol

Annotation Type Files (ATY)

The format of ATY files describes the Note types, the relationships between the category and the type of note, and the associated cardinalities. Learn how ATY files are structured and defined below.

STRUCTURE OF ANNOTATION TYPE FILES

A type file contains various modules:

- a module defining the note types
- a module defining the relationships between the category and the note type

The note type definition module must be in front of the annotability definition module.

```
annot_model_file      =      type_note_module
                             relation_note_module
```

NOTE TYPE DEFINITION MODULE

The module consists of a list of note type definitions:

- For each note type, the first two declarations are mandatory; the others correspond to attribute declarations. Saving an annotation must contain a reference to the annotated object and its name.
- The declaration of an attribute consists of its name, type, and the declaration of its associated properties.
- The declaration of a property contains its name and value.
- The reference to the annotated object must be unique to SCADE Suite, it is given by the application that uses the annotation service.

- The default initial values of note attributes are given in the NT_DEFAULT_VALUE type properties on a note type attribute definition.

```
                              “Notes-Types” [comment] “DEFINITIONS ::=”
type_note_module          =   “BEGIN”
                              {type_note_definition_list}
                              “END”.
comment                   =   “{string}”
type_note_definition_list =   type_note_definition
                            | type_note_definition_list  “,”  type_note_definition.
                              type_name “::=SEQUENCE OF{SEQUENCE {”
                              “annot_object OID,”
type_note_definition      =   “name STRING”
                              group_list
                              “}}”.
type_name                 =   ident.[1]
group_list                =   group
                            |  group_list “,” group.
group                     =    “group_name {“attribute_list”}”.
group_name                =   ident.
attribute_list            =   attribute
                            | attribute_list “,” attribute.
attribute                 =   att_name att_type [properties].
att_name                  =   ident.
att_type                  =     “STRING” | “string”
                            | “TEXT” | “text”
                            | “CHARACTER” | “character”
                            | “DATE” | “date”
                            | “INTEGER” | “integer”
                            | “REAL” | “real”
                            | “ENUM” | “enum”
                            | “FILE” | “file”
                            | “BOOLEAN” | “boolean”
properties                =   “{”
                              {property_list}
                              “}”
property_list             =   property
                            | property_list “,” properties.
property                  =   prop_name prop_value.
prop_name                 =   ident.
```

```
prop_value          =   value
                        | “{” enum_values “}”
value               =   “F”[2] | “T”[3] | integer[4] | real[5] | string[6] | “ " ” char[7] “ " ”.
enum_values         =   string
                        | enum_values “,” string.
```

1. ident is an identifier
2. F is a Boolean false value
3. T is a Boolean true value
4. integer is an integer value
5. real is a real value
6. string is a character string between quotes
7. char is a character

ANNOTABILITY RELATIONSHIP DEFINITION MODULE

For each category, indicate the note types that can be associated with it, the cardinalities, and a Boolean value indicating whether a default note must be: `default_note_boolean=booleancreated`.

```
                            “Annotation-Rules”  [ comment] “DEFINITIONS::=”
relation_module         =   “BEGIN”
                            {relation_type_annotable}
                            “END”.
                            category “::= {”
relation_type_annotable =   [relation_list]
                            “}”.
category                =   ident.
relation_list           =   relation
                            | relation_list “,” relation.
                            “{”    note_type_name
                            default_note_boolean
relation                =   min_card
                            max_card
                            “}”.
default_note_boolean    =   boolean
note_type_name          =   ident.
min_card                =   integer
max_card                =   integer
```

ident is an identifier - integer is an integer value - boolean is a Boolean value

The categories with which note types can be associated are listed below:

Associated with graphical objects	Associated with definition objects
assertion ::= { } ref_ident ::= { } literal ::= { } predef_opr ::= { } ref_operator ::= { } terminator ::= { } state_machine ::= { } state ::= { } transition ::= { } activate_block ::= { } branch ::= { } action ::= { } net_diagram ::= { } text_diagram ::= { } tree_diagram ::= { }	package ::= { } constant ::= { } sensor ::= { } named_type ::= { } type_element ::= { } output ::= { } input ::= { } hidden ::= { } local ::= { } probe ::= { } signal ::= { } operator ::= { }

Example: The following example defines three types of note: **`Civil_status`**, **`Marriage`**, **`Activity`**, **`Inlaid_ann`** (inlayable note). The Note type definition module can be expressed as follows:

```
Notes_Types Esterel} DEFINITIONS::=
    BEGIN
    Civil_status::= SEQUENCE OF { SEQUENCE {
        annot_object OID,
        name STRING,
        information {
            Name STRING, {NT_OBLIGATORY T, NT_UPPER T },
            First_name STRING {NT_CAPITAL T},
            Address TEXT,
            Date_of_birth DATE,
            Sex CHARACTER {NT_UPPER T},
            Situation ENUM {NT_ENUM_VALUES
              {“Unmarried”, “Married”,“Separated”, “Widow(er)”}}, dependent_persons INTEGER
{NT_MIN 0}, Photo FILE }}}
    Marriage::= SEQUENCE OF {SEQUENCE{
        annot_object OID
        name STRING,
        information {
            Married_name STRING, {NT_UPPER T },
            Name_jf STRING,
            Married_address TEXT,
            Date_of_marriage DATE,
            Number_of_children INTEGER {NT_MIN 0}}}}
    Activity::= SEQUENCE OF {SEQUENCE{
        annot_object OID
        name STRING,
        information {
            Leisure_activities STRING,
            Profession STRING }}}
    Inlaid_ann::= SEQUENCE OF {SEQUENCE{
        annot_object OID
        name STRING,
        note_graphical_information {
            NT_INCRUSTABLE BOOLEAN { {NT_DEFAULT_VALUE T},
            NT_PERMANENT BOOLEAN { {NT_DEFAULT_VALUE F},
            NT_POS_Y REAL {NT_DEFAULT_VALUE -1.0},
            NT_POS_Y REAL {NT_DEFAULT_VALUE -1.0},
            NT_LOCATION STRING {NT_DEFAULT_VALUE “RELATIVE”},
            NT_FONT_SIZE INTEGER {NT_DEFAULT_VALUE 10},
            NT_FONT_COLOR STRING {NT_DEFAULT_VALUE “(0,255,255)”}
            NT_MODIFY BOOLEAN },
        information {
            Description TEXT}}}
    END
```

Assuming there is a person category in the annotated model, the module describing the relationships between the category and type of note is as follows:

```
Annotation - Rules {Esterel} DEFINITIONS::=
    BEGIN
        person::= {
        {Civil_status T1 1},
        {Marriage F 1 10},
        {Activity F 0 -1},
        {Inlaid_ann F 1 1}}
    END
```

A `Civil_Status` type default note is created each time a person category object is created.

Annotation Files (ANN)

The ANN file format describes annotations and their associated note types. Annotations are grouped by type. Note types are stored in these files to allow checks wrt. the annotation model file when loading annotations. The note types definition module is the same as for the annotation model file. These files cannot be modified by users. Learn how ANN files are structured and defined below.

ANNOTATION FILE STRUCTURE

```
annotation_file       =       note_type_module
                              | note_value_module
```

ANNOTATION VALUES DEFINITION MODULE

```
                                “Notes-Values”  [ comment ] “DEFINITIONS”
                                “::=”
note_value_module          =    “BEGIN”
                                {note_by_type}
                                “END”.

                                note_type_name “::= {”
note_by_type               =    {note_list}
                                “}”.
note_type_name             =    ident.
note_list                  =    | note_list “,” note.
                                “{”    reference “,”
note                       =    note_name
                                {att_value_list}
                                “}”.
note_name                  =    string.
reference                  =    string.
att_value_list             =    “,” value
                                | att_value_list value.
```

N.B.: string is a character string between quotes.

Example: Assuming that references to annotated objects are integers: two notes (Note-1, Note-2) of Civil_status type and one note (activity_smith) of Activity type.

The annotation definition module is as follows:

```
Notes-Values {Esterel} DEFINITIONS::=
    BEGIN
        Civil_status::= {
           {“0”, “Note-1”, “JONES”, “David”, “Regent Street”, “1930/06/19”, ’M’, “Married”,
0,#},
          {“1”, “Note-2”, “SMITH”, “Fred”, “Oxford Street”, “1968/08/09”, ’M’, “Unmarried”,
0,#},
        Activity::={
            {“1”, “activity_smith”,“Horse_riding”,“School_teacher”}}
    END
```

List of Predefined Properties

Predefined properties are distinguished as follows:

- "Kernel Recognized Properties"
- "Properties Recognized in Annotation Editing Window"

Kernel Recognized Properties

The following table lists all kernel recognized properties at your disposal.

Property	Description
NT_DEFAULT_VALUE	Assigns the attribute default value when created. It is of the "att" type (attribute type).
NT_MIN	This property is defined for the "integer" and "real" type attributes. It gives the attribute minimum value. It is of the "att" type.
NT_MAX	This property is defined for the "integer" and "real" type attributes. It gives the attribute maximum value. It is of the "att" type.
NT_LOWER	This property is defined for "string", "text", "character" or "file" type attributes. It transforms the characters in the string into lower case if its value is "true". It is of the "Boolean" type. For "file" type attributes, only the value of the attribute is modified (filename).
NT_UPPER	This property is defined for "string", "text", "character" or "file" type attributes. It transforms the characters in the string into upper case if its value is "true". It is of the "Boolean" type. For "file" type attributes, only the value of the attribute is modified (filename).
NT_CAPITAL	This property is defined for "string", "text", "character" or "file" type attributes. It transforms the first letter in the string into upper case and the others into lower case if its value is "true". It is of the "Boolean" type. For "file" type attributes, only the value of the attribute is modified (filename).
NT_MANDATORY	The attribute must be valued. It is of the "boolean" type.
NT_ENUM_VALUES	This property is only defined for "enum" type attributes. Its value is a list of character strings ("enum" type).

Properties Recognized in Annotation Editing Window

The following table lists all properties recognized for annotation editing.

Property	Description
NT_DISPLAYABLE	This property is used to display or hide its attribute field in the annotation index attribute box. Its value is of the “boolean” type.
NT_EDITABLE	The attribute field cannot be modified from the **Edit Note** window or from **Diagram Views** if the value of this property is set to “False”. Its value is of the “Boolean” type.
NT_LEFT_LABEL_POS	Its value is an integer that gives the position of the attribute field label in the attribute box. This property is used to position the label above the field, or to the left of it. It is of the “integer” type (1 to position the label above the field, 2 to position it to the left).
NT_FIELD_WIDTH	Its value gives the width of the field in the attribute box. This property is not recognized for “Boolean” and “enum” type attributes. It is of the integer type.
NT_FIELD_HEIGHT	Its value gives the height of the field in the attribute box. This property is recognized for “text” type attributes. It is of the integer type.
NT_LEFT_LABEL	Its value is of the “string” type. It is used to modify the attribute field label. By default its name is the attribute.
NT_RIGHT_LABEL	Its value is of the “string” type. It gives the right-hand label (to display units for example).
NT_RADIO_BOX	This property is recognized for “Boolean” type attributes. If its value is set to “true”, the graphic field associated with the attribute is represented by a box with mutually exclusive selection buttons. By default, the fields of “Boolean” type attributes are boxes with two-state buttons.

Inlayable Annotations Predefined Attributes

An inlayable annotation type has attributes belonging to the *note_graphical_information* group. These attributes define the graphical display characteristics of inlayable annotations. The following table lists all available attributes for inlayable annotations.

Property	Description
`NT_INCRUSTABLE: (T, F)`	If equal to T, the annotation is inlayable, otherwise it is not.
`NT_PERMANENT: (T, F)`	If equal to T, annotation display in the diagram is permanent. If set to F, annotation display is not permanent, but can be made visible using *Display Options > Display Hidden Notes* context menu in diagrams.
`NT_MODIFY: (T, F)`	If equal to T, the annotation position, height and width can be modified independently from the object to which it is attached, otherwise the modification depends entirely on this object.
`NT_LOCATION: ("RELATIVE", "ABSOLUTE")`	If equal to *"RELATIVE"*, the position of the annotation is relative to the origin of the object to which it is attached; if equal to *"ABSOLUTE"*, it is absolute in the diagram reference point.
`NT_FONT_NAME: string`	This attribute indicates the font name.
`NT_FONT_SIZE: integer`	This attribute specifies the font size.
`NT_FONT_COLOR: string`	This attribute indicates the font color, by a string using the RGB format: for example "(0,0,255)" for blue color.
`NT_UNDERLINE, NT_BOLD, NT_ITALIC: Boolean`	This attribute can used to specify additional style for the font.
`NT_FONT: integer`	This attribute exists for compatibility reason and references a font index in the font file associated with the project.
`NT_POS_X and NT_POS_Y: real`	This attribute defines the relative or absolute position of the annotation along the X and Y axes according to the value of attribute `NT_LOCATION`. This attribute is compulsory when `NT_INCRUSTABLE` equals *T*.
`NT_FRAME: (T, F)`	If equal to T, a frame is displayed around the note content. Default value is F.
`INT_FRAME_HEIGHT: integer`	Attribute taken into account only if `NT_FRAME` is equal to *T*. It defines frame height in pixels; If `NT_FRAME_HEIGHT` is equal to 0, the height is adjusted to the content.

Property	Description
NT_FRAME_WIDTH: integer	Attribute taken into account only if NT_FRAME is equal to *T*. It defines frame width in pixels; If NT_FRAME_WIDTH is equal to 0, the width is adjusted to the content.
NT_CENTERING_RIGHT and NT_CENTERING_CENTER: (T, F)	• If NT_CENTERING_RIGHT equals to T, the annotation is right-aligned. • If NT_CENTERING_CENTER equals to T, the annotation is centered. • If NT_CENTERING_CENTER and NT_CENTERING_RIGHT have the same value, the annotation is left-aligned.
NT_UNDERLIGN: (T, F)	If equal to T, the text is underlined. Default value is false.

The table below summarizes possible layouts depending on values of NT_CENTERING_CENTER and NT_CENTERING_RIGHT.

Table 1.7: Possible layouts for inlayable annotations

	NT_CENTERING_RIGHT			
		T	F	absent
NT_CENTERING_CENTER	T	left	center	center
	F	right	left	left
	absent	right	left	left

The area allocated to annotation content display is fixed and depends on the content to be displayed (all of the content is displayed) and on the frame.

Printable Annotation Predefined Attributes

Printable annotations are not displayed on the screen and can only be printed. These annotations can be associated with blocks (Equation, Parameter, Type, Global Variable and Constant Blocks). The attributes of a printable annotation belong to the *note_graphical_information* group. These attributes define the graphical printing characteristics of the printable annotations. The following table lists all available attributes for printable annotations.

Property	Description
`NT_PRINTABLE: (T, F)`	If equal to T, the annotation is printable, otherwise it is not.
`NT_FONT_NAME: string`	This attribute indicates the font name.
`NT_FONT_SIZE: integer`	This attribute specifies the font size.
`NT_FONT_COLOR: string`	This attribute indicates the font color, by a string using the RGB format: for example "`(0, 0, 255)`" for blue color.
`NT_UNDERLINE, NT_BOLD, NT_ITALIC: Boolean`	This attribute can be used to specified additional style for the font.
`NT_FONT: integer`	This attribute exists for compatibility reason and references a font index in the font file associated with the project.
`NT_POS_X and NT_POS_Y: real`	This attribute defines the relative or absolute position of the annotation along the X and Y axes. The attribute is required when `NT_PRINTABLE` equals *T* to save the position of your annotation in its diagram.
`NT_FRAME: (T, F)`	If equal to T, a frame is displayed around the node content. Default value is F.
`NT_FRAME_HEIGHT: integer`	This parameter is taken into account only if `NT_FRAME` is equal to T. It defines the frame height in pixels. If `NT_FRAME_HEIGHT` is equal to 0, the height is adjusted to content.
`NT_FRAME_WIDTH: integer`	This parameter is only taken into account if `NT_FRAME` is equal to T. It defines the frame width in pixels. If `NT_FRAME_WIDTH` equals 0, the width is adjusted to content. The attribute is required when `NT_PRINTABLE` equals *T* to save the frame size of your annotation in its diagram.

Property	Description
NT_CENTERING_RIGHT and NT_CENTERING_CENTER: (T, F)	• If NT_CENTERING_RIGHT equals to T, the annotation is right-aligned. • If NT_CENTERING_CENTER equals to T, the annotation is centered. • If NT_CENTERING_CENTER and NT_CENTERING_RIGHT have the same value, the annotation is left-aligned.
NT_UNDERLIGN: (T, F)	If equal to T, the text is underlined. Default value is false (F).

The table below summarizes possible layouts depending on values of NT_CENTERING_CENTER and NT_CENTRING_RIGHT.

Table 1.8: Possible layouts for printable annotations

	NT_CENTERING_RIGHT			
		T	F	absent
NT_CENTERING_CENTER	T	left	center	center
	F	right	left	left
	absent	right	left	left

The area allocated to annotation content display is fixed and depends on the content (all of the content) and on the frame to be displayed.

An annotation cannot be both inlayable and printable. If NT_INCRUSTABLE equals *T* and NT_PRINTABLE equals *T*, the annotation is only inlayable: the inlayable property has priority over the printable property.

2 / Editor Save Format

This chapter describes the format used to save various types of information associated with SCADE Suite models and added in SCADE files (`.xscade`).

- "Save File Format"
- "Graphical Information Description"
- "Data Flow Graphical Representation"
- "Control Flow Graphical Representation"
- "XML Save Format of Graphical Elements"

Save File Format

For a given SCADE Suite project, all Editor-specific information is stored:

- in specific locations of `.xscade` files corresponding to *pragmas*
- in XML format following the UML diagram below

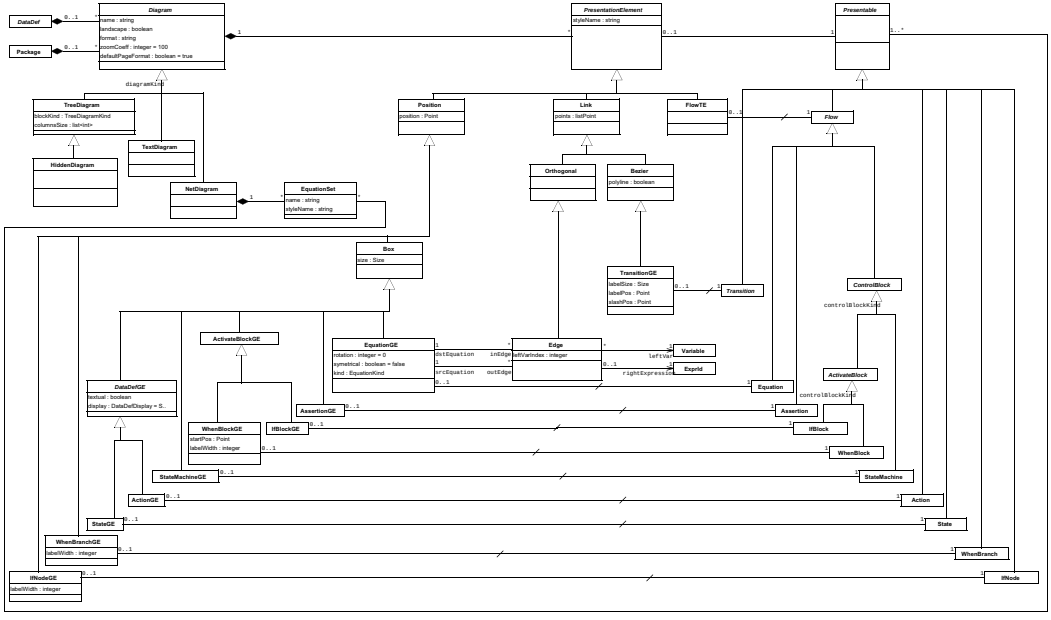


Figure 2.1: UML class module diagrams

- “Pragmas”
- “UML Diagram”

Pragmas

Pragmas provide a means to pass information to tools without having to use language semantics. The concept of pragma belongs to the language; the pragmas themselves do not.

Pragmas are introduced:

- in `.xscade` files by the `<pragmas>` tag as a child of the corresponding element
- in `.scade` files by the `#pragma#end` sequence

All editor pragmas are stored in `.xscade` files with the namespace prefix `ed` declared at the beginning of the file.

Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<File xmlns="http://www.esterel-technologies.com/ns/scade/1" xmlns:ed="http://
www.esterel-technologies.com/ns/scade/pragmas/editor/1" xmlns:kcg="http://
www.esterel-technologies.com/ns/scade/pragmas/codegen/1">
    <declarations>
    ...
    </declarations>
```

Each SCADE Suite element can be associated with a specific pragma, which name corresponds to the element name prefixed with `ed`.

Example: The following is an example of editor pragma for the *Constant* element:

```
<Constant name="kFailSoftRoll">
        <type>
            <NamedType>
                <type>
                    <TypeRef name="real"/>
                </type>
            </NamedType>
        </type>
        <value>
            <ConstValue value="20.0"/>
        </value>
        <pragmas>
            <ed:Constant oid="!ed/1A24/3B54/2D4/46CD8621/28EF"/>
        </pragmas>
</Constant>
```

UML Diagram

The XML format of Editor-specific information saved in `.xscade` files can be inferred from the UML diagram (Figure 2.1) with dedicated rules.

This allows to:

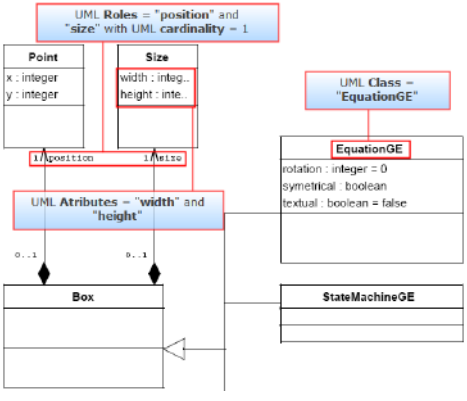
- get a precise description of Editor-specific information associated with each element
- represent in XML the UML notions of class, attribute, role, cardinality, and association

The UML to XML transformation rules are summed up below.

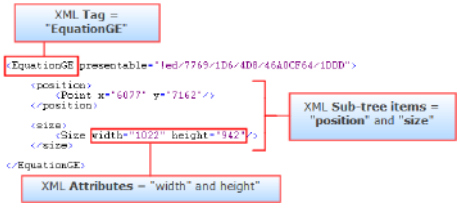
Table 2.1: Applicable transformation rules

A UML...	is represented in XML as...
Class	a sub-tree with the UML class as XML tag name
Attribute	an XML attribute having the same name
Association	an XML sub-tree with the tag of each item of the XML sub-tree corresponding to the UML association or role name
Cardinality	the possible number of an XML sub-tree items

Example: The following UML roles, classes, and attributes ...



... translate in XML as follows:



Graphical Information Description

SCADE Suite elements are arranged using diagrams associated with the Scade View, Textual Views, and Diagram Views. The figure below shows the attributes of the UML abstract class *Diagram*. The diagrams (NetDiagram, TextDiagram, HiddenDiagram, and TreeDiagram) inherit the attributes of *Diagram*:

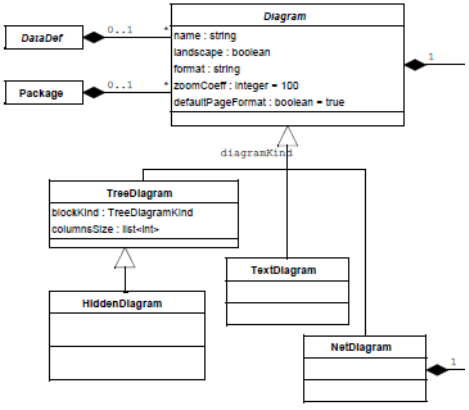


Figure 2.2: UML Diagram attributes

GRAPHICAL DIAGRAM

The graphical diagram (stored in `.xscade` files using `<NetDiagram>` XML tag) is used to present all elements contained in the graphical diagram (i.e., all graphical symbols linked to the semantic elements). The upper-left corner of diagrams has coordinates equal to (0,0).

Note

In Scade 6, the unit used for logical coordinates is 1/100 of centimeter. It ensures the pixel size is always greater than the unit used, even with high-resolution display.

The table below identifies the graphical objects used to represent elements in a graphical diagram depending on the type of flow (data or control flow) used to represent the model.

Table 2.2: Semantic and associated graphical objects used in model representations

Type of representation	Semantic Objects	Graphical Objects
Data flow	Equation	Box
	Assertion	Box
	Data Flow	Edge
Control flow	State Machine	Box
	State	Box
	Fork	Box
	Transition	Curve (polyline or Bezier line)
	If Block	Box
	When Block	Box

TEXTUAL DIAGRAM

The textual diagram (stored in `.xscade` files using the `<TextDiagram>` XML tag) is used to present the textual elements of a text diagram. It contains all textual symbols linked to the top-level semantic elements.

TREE DIAGRAM

The tree diagram (stored in `.xscade` files using the `<TreeDiagram>` XML tag) is used to present the package constants, types or sensors. The table below describes the attributes of the XML TreeDiagram tag.

Table 2.3: XML attributes of a tree diagram

Attribute	Description
blockKind	Refers to the kind of element represented by the diagram: constant, type, or sensor
columnSize	Refers to the size of the Grid Views adjusted by users: the column size information is saved as a list of values separated by a comma and following the column order

Example:

```
<pragmas>
   <ed:Package oid="!ed/401B/4AFD/A70/46A07F51/6F3B" xmlns="http://www.esterel-technologies.com/ns/scade/
pragmas/editor/5">
      <diagrams>
        <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/401C/4AFD/A70/46A07F51/E6F"
blockKind="constants"/>
        <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/401D/4AFD/A70/46A07F51/546E"
blockKind="types"/>
        <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/401E/4AFD/A70/46A07F51/6D8A"
blockKind="sensors"/>
      </diagrams>
    </ed:Package>
</pragmas>
```

HIDDEN DIAGRAM

The hidden diagram (stored in `.xscade` files using the `<HiddenDiagram>` XML tag) is used to present hidden input values of user operators in a dedicated table. The table below describes the attributes of the XML HiddenDiagram tag.

Table 2.4: XML attributes of a hidden diagram

Attribute	Description
blockKind	Refers to the kind of element represented by the diagram: constant, type, or sensor
columnSize	Refers to the size of the Grid Views adjusted by users: the column size information is saved as a list of values separated by a comma and following the column order

Data Flow Graphical Representation

In a graphical diagram, a data flow is the graphical representation of a set of equations involving the inputs, outputs, and local variables of an operator. Data flows are represented in SCADE Suite using two graphical objects (boxes and edges) associated with three semantic elements (see Table 2.2).

- "Equation (Box)"
- "Assertion (Box)"
- "Data Flow (Edge)"

Equation (Box)

Equations are graphically represented using boxes characterized by:

- output pins (right side of box) for local variables defined by the equation
- input pins (left side of box) representing the "parameters" of the equation
- properties of symmetry and rotation
- the nature of the equation (textual or not)

Note

Local variables (output pins) are also called "left variables" as they appear on the left of the "=" sign in the textual representation of the equation.

To define equations in graphical diagrams, three kinds of representations are used as illustrated below: flow sources, flow sinks, computation blocks.

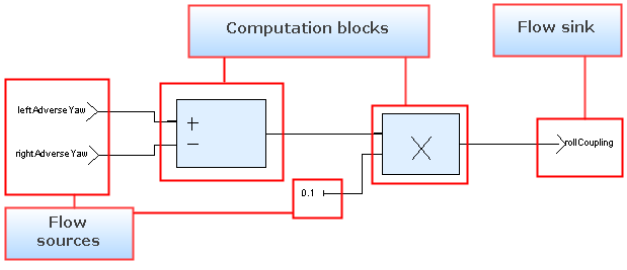
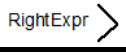


Figure 2.3: Elements representative of equations

FLOW SOURCE

The table below gives both representation of a simple *flow source* equation.

Table 2.5: Textual and graphical representation of a flow source

Textual Representation	Graphical Representation
LeftVar = RightExpr	

In a *flow source*, the two members of the equation are:

- `LeftVar` - identifier bound to local variable
- `RightExpr` - identifier bound to input (or local variable used as in-flow or sensor)

As in Table 2.5, *flow source* equations are graphically represented by a box labeled according to the `RightExpr` name with a single output pin representing `LeftVar`. For instance, flow sources represented in Figure 2.3 are _L2 = `rightAdverseYaw`, _L4 = `leftAdverseYaw`, and _L3 = `0.1`.

Depending on the kind of `RightExpr` (input, local variable, or sensor), the box has different graphical representations.

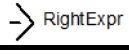
Note

Since SCADE Suite 6.0, SCADE Suite Editor offers a new kind of flow sources: textual expressions (consult "Creating Textual Expressions" on page 270 in *SCADE Suite User Manual*). In textual expressions, RightExpr is not limited to being a reference to an input or to a local. It can be almost any kind of single-valued expression.

FLOW SINK

The table below gives both representation of a simple *flow sink* equation:

Table 2.6: Textual and graphical representation of a flow sink

Textual Representation	Graphical Representation
LeftVar = RightExpr	

In a *flow sink*, the two members of the equation are:

- `LeftVar` - output (or local variable used as out-flow or probe)
- `RightExpr` - identifier bound to local variable

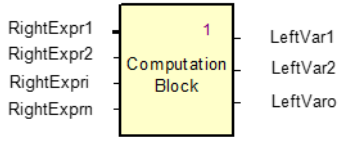
As in Table 2.6, *flow sink* equations are graphically represented by a box labeled according to the name of `LeftVar` with a single input pin representing `RightExpr`. For instance, flow sink represented in Figure 2.3 is `rollCoupling` = _L5.

Depending on the kind of `RightExpr` (output, local variable, or probe), the box has different graphical representations.

COMPUTATION BLOCK

The table below provides both representation of a *computation*:

Table 2.7: Textual and graphical representation of a computation

Textual Representation	Graphical Representation
$LeftVar_1,LeftVar_2,...,LeftVar_o$ = Expr ($RightExpr_1,RightExpr_2,...,RightExpr_i$)	

In a *computation*, the two members of the equation are:

- $LeftVar_{1,2,...,o}$ - "left variables" (variables defined by this equation)
- Expr($RightExpr_1,RightExpr_2,...,RightExpr_n$) - expression pattern involving a set of parameters bound to a set of subexpressions $RightExpr_n$

As shown in the table, *computation* equations are represented by a box with:

- output pins corresponding to each $LeftVar_o$
- input or hidden inputs corresponding to each $RightExpr_n$ (input pins for parameters that must be bound to local variables and hidden input pins for parameters that can be bound to a more general expression).

SCADE Suite Editor provides a number of different expression patterns to represent usual operations (calling a pre-defined or user-defined operator, applying a higher-order operator such as fold, map, etc. on a pre-defined or user-defined operator).

Assertion (Box)

Assertions are represented using an unconnected box labeled according to the name and expression of the assertion.

Data Flow (Edge)

Data flows are represented using edges linking the parameter of an equation ("target equation") to an actual local variable definition provided by another equation ("source equation"), through the identifier of that local variable. Each edge is therefore associated with a single local variable.

The table below describes the elements used to represent edges.

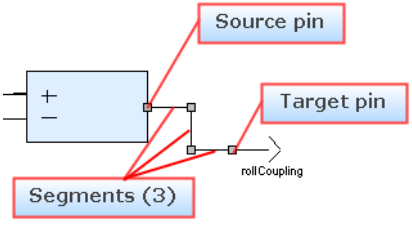
Table 2.8: Elements constitutive of an edge

Element	Description
Source pin	The source pin is given by: • the source equation - the source pin of the edge is an output pin of the source equation • the rank of the edge variable in the left part of the source equation (i.e the rank of the output pin)
Target pin	The target pin is given by: • the target equation - the target pin of the edge is an input pin of the target equation • the rank of the edge variable in the "rigt expression" list of the source equation (i.e the rank of the input pin)

Table 2.8: Elements constitutive of an edge

Element	Description
Set of segments	The set of segments is defined as a set of junction points with the following properties: • there must be an odd number of segments (i.e an even number of points) • the first point should correspond to the source pin • the last point should correspond to the target pin • intermediate points should be located on the horizontal or vertical axis of the previous point of the list

Example: The following shows the source pin, target pin and segments of an edge (in data flow representation).



Control Flow Graphical Representation

In a graphical diagram, a control flow representation is the graphical representation of a set of state machines, states, transitions, and fork symbols. Control flow representations are created in SCADE Suite using two graphical objects associated with four semantic elements (see Table 2.2).

- "State Machine (Box)"
- "State (Box)"
- "Fork (Box)"
- "Transition (Curve)"
- "If Block (Box)"
- "If Node (Box)"
- "If Action (Box)"
- "When Block (Box)"
- "When Branch (Box)"
- "When Action (Box)"

State Machine (Box)

State Machines are represented using an unconnected box outlined with a dotted line.

The State Machine box is labeled according to the name of the StateMachine name and contains all states and transitions involved in the automaton.

State (Box)

States are represented using a box labeled with the state name. The transitions involved in the state are connected to the box. The box has three representations, each corresponding to a type of state: simple, embedded or hidden displayed.

The table below describes the different representations associated with these states.

Table 2.9: Graphical representation of States

Nature of State	Representation
Simple	Simple plain box
Embedded	Box with a header containing: • the name of the state • a transparent container containing all graphical symbols associated with the state (data flows and other sub state machines)
Hidden displayed	Plain box with a header labeled with the name of the state. The content of the state is described in another diagram.

Fork (Box)

Fork symbols are represented using a simple box connected to transitions.

Transition (Curve)

Transitions and fork transitions are represented using a polyline or bezier curve connecting a source state to a destination state or linked to a fork symbol.

The priority of the transition (consult section 3.2 "State Transitions" in *Scade Language Primer*), is indicated by the circle located at the beginning of the line. Two labels are associated with the transition: the action and the

condition of the transition. Both labels have their own position and size (see "Graphical Diagram"). The table below gives information on the forms used to represent transitions:

Table 2.10: Graphical representation of Transitions

Transition representation	Description
Simple Bezier Line (no intermediate points)	A simple bezier line is represented by: • a first point representing the junction point between transition and source state (point at the source state border) • a last point representing the junction point between transition and target state (point at the target state border) • a third and a fourth representing the tangent points s to the first and last points
Bezier Line (with intermediate points)	A bezier line with one intermediate point is represented by: • a first and second point representing respectively: the junction point between transition and source state (point at the source state border) and the tangent point to the first point • a third, fourth and fifth point representing respectively: the tangent point to the intermediate point, the intermediate point and the second tangent point to the intermediate point • a sixth and seventh point representing respectively: the tangent point to the last point and junction point between transition and target state (point at the target state border)
Polyline	A polyline is seen as a bezier line where tangent points are not considered (even though they must exist) to enable the switching between bezier and polyline representations.

If Block (Box)

If Blocks are represented using an unconnected box labeled with the `IfBlock` name, and contains all If Branches and If Actions.

If Node (Box)

If Nodes are represented using a little diamond-shaped box with two lines that indicate the possible choices: then on the right and else below.

If Action (Box)

If Actions are represented using a box corresponding to the action name (then or else).

When Block (Box)

When Blocks are represented using an unconnected box labeled with the `WhenBlock` name, and contains all When Branches and When Actions.

When Branch (Box)

When Branches are represented using a little triangle-shaped box with a line that indicates the possible action.

When Action (Box)

When Actions are represented using a box with the action name (true).

XML Save Format of Graphical Elements

As seen in the previous sections, graphical diagrams of data flows and control flows are created using a set of graphical elements. In `.xscade` files, the information related to these graphical elements (size, position, form, rotation, etc.) is bordered by the `NetDiagram` XML tag and stored using specific XML tags and attributes.

Example: The following presents the empty XML structure of a graphical diagram.

```
(...)
<diagrams>
    <NetDiagram name="..." landscape="..." format="A4 (210 297)" oid="...">
    <presentationElements>
    <StateMachineGE presentable="...">
    ...
    </StateMachineGE>
    <StateGE stateDisplay="..." presentable="...">
    ...
    </StateGE>
    <TransitionGE polyline="..." presentable="...">
    ...
    </TransitionGE>
    <EquationGE rotation="..." symetrical="..." textual="..." presentable="...">
    ...
    </EquationGE>
    <Edge leftVarIndex="1" rightExprIndex="1" srcEquation="..." dstEquation="...">
    ...
    </Edge>
    <AssertionGE presentable="...">
    ...
    </AssertionGE>
    <presentationElements>
    </NetDiagram>
</diagrams>
(...)
```

The following sections list and detail all XML tags and attributes used to store data flow and control flow graphical elements:

- “Data Flow Graphical Elements”
- “Control Flow Graphical Elements”
- “Specific Information”

Data Flow Graphical Elements

The table below describes graphical elements contained by data flow graphical diagrams.

Table 2.11: XML save format (tags and attributes) for data flow representations

Semantic Object (graphical)	XML Tag	Attribute	Type	Description
Equation (Box)	`EquationGE`	presentable	string	Refers to the oid of associated semantic object
		rotation	positive integer	Refers to the rotation of the object Values: 0 (default), 90, 180, 270
		symmetrical	boolean	Refers to the symmetrical properties of an equation: True if the equation is symmetrical
		kind	string	Equals OBJ_LIT if the equation is a textual expression
		stylename	string	Refers to the style applied to the object, if not the default one.
Data flow (Edge)	`Edge`	leftVarIndex	integer	Refers to the rank of output pin
		rightExprIndex	integer	Refers to the rank of input pin
		srcEquation	string	Refers to the oid of source equation
		dstEquation	string	Refers to the oid of target equation
		stylename	string	Refers to the style applied to the object, if not the default one.
Assertion (Box)	`AssertionGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.

Example 1: In distributed RollControl model, the joystickCmd input is represented using a box



The *semantic* flow source equation (_L1=joystickCmd, see "Flow Source") corresponding to the joystickCmd input is stored in RollControl.xscade and given an object identifier (**oid**):

```
<!-- _L1 = joystickCmd; -->
<Equation>
    <lefts>
        <VariableRef name="_L1"/>
    </lefts>
    <right>
        <IdExpression>
            <path>
                <ConstVarRef name="joystickCmd"/>
            </path>
        </IdExpression>
    </right>
    <pragmas>
        <ed:Equation oid="!ed/22F/1469/E3C/46CC6DEA/7BE8"/>
    </pragmas>
</Equation>
```

Object identifier (oid)

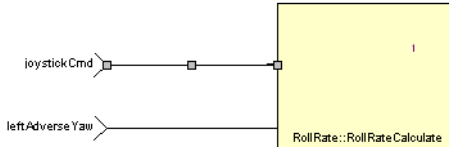
The *graphical* object (box) representative of the flow source equation is stored in RollControl.xscade using the **EquationGE** tag with the **presentable** attribute referring to the **oid** previously defined:

```
<EquationGE presentable="!ed/22F/1469/E3C/46CC6DEA/7BE8">
    <position>
        <Point x="4207" y="5186"/>
    </position>
    <size>
        <Size width="264" height="503"/>
    </size>
</EquationGE>
```

If, for some reason, we decide to rotate graphically the input by 90°, the attribute **rotation** appears in the XML save information of the equation:

```
<EquationGE rotation="90" presentable="!ed/22F/1469/E3C/46CC6DEA/7BE8">
    <position>
        <Point x="3625" y="4207"/>
    </position>
    <size>
        <Size width="264" height="503"/>
    </size>
</EquationGE>
```

Example 2: Take data flow edge from `joystickCmd` input to `RollRate::RollRateCalculate` operator



The *graphical* object (edge) representative of the data flow is stored in RollControl.xscade using the **Edge** tag with the **srcEquation** attribute referring to the **oid** of the input flow source equation (`_L1=joystickCmd`) and the **dstEquation** attribute referring to the **oid** of the `RollRate::RollRateCalculate` operator (computation block, see "Computation block"):

```
(...)
<Edge leftVarIndex="1" rightExprIndex="1" srcEquation="!ed/22F/1469/E3C/46CC6DEA/7BE8"
dstEquation="!ed/235/1469/E3C/46CC6DEA/FDB">

    <positions>
    <Point x="4471" y="5450"/>
    <Point x="6218" y="5450"/>
    <Point x="6218" y="5450"/>
    <Point x="7990" y="5450"/>
    </positions>

</Edge>
(...)
```

Control Flow Graphical Elements

The table below describes graphical elements contained by control flow graphical diagrams.

Table 2.12: XML save format for control flow representations

Semantic Object (graphical)	XML Tag	Attribute	Type	Description
State Machine (Box)	`StateMachineGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.
State (Box)	`StateGE`	presentable	string	Refers to the oid of associated semantic object
		display	string	Refers to the type of representation chosen for states: "embeddedGraphical" (default), "embeddedTextual", or "split"
		stylename	string	Refers to the style applied to the object, if not the default one.
Transition (Curve)	`TransitionGE`	presentable	string	Refers to the oid of associated semantic object
		polyline	boolean	Refers to the type of representation chosen for transitions: is true if the transition is represented by polyline
		stylename	string	Refers to the style applied to the object, if not the default one.
Fork	`N/A`	-	-	Fork symbols do not have any saved information: their location is inferred from forked transition positions

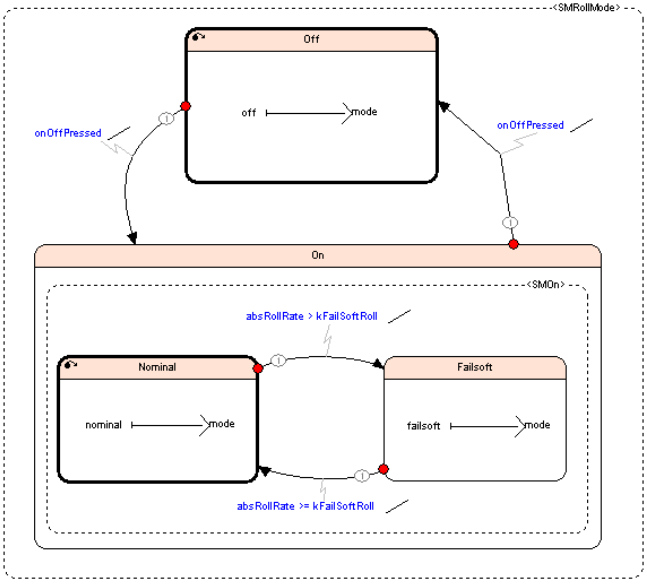
Table 2.12: XML save format for control flow representations (Continued)

Semantic Object (graphical)	XML Tag	Attribute	Type	Description
If Block (Box)	`IfBlockGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.
If Node (Box)	`IfNodeGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.
		labelWidth	integer	Refers to width used for displaying text. If width is shorter than text, several lines are used for text display.
Actions (If or When	`ActionGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.
		display	string	Refers to the type of representation chosen for states: "embeddedGraphical" (default), "embeddedTextual", or "split"
When Block (Box)	`WhenBlockGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.

Table 2.12: XML save format for control flow representations (Continued)

Semantic Object (graphical)	XML Tag	Attribute	Type	Description
When Branch (Box)	`WhenBranchGE`	presentable	string	Refers to the oid of associated semantic object
		stylename	string	Refers to the style applied to the object, if not the default one.
		labelWidth	integer	Refers to width used for displaying text. If width is shorter than text, several lines are used for text display.

Example: Consider TollMode operator with control flow representation in RollControl example



This control flow graphical diagram contains 16 graphical objects:

- two State Machines (SMRollMode and SMOn)
- four States (On, Off, Nominal, Failsoft)

- six equations (three flow sources and three flow sinks) associated with three data flows
- four transitions

The RollMode.xscade file contains the information associated with all graphical objects representative of 16 semantic objects: two StateMachineGE, four StateGE, six EquationGE, three EdgeGE and four TransitionGE XML tags together with their childs and attributes.

The following is the content of the XML `NetDiagram` tag of the RollMode.xscade file including all graphical information of the control flow from the model above:

```
<diagrams>
            <NetDiagram name="diagram_RollMode_1" landscape="true" format="A4 (210 297)"
              oid="!ed/19F9/3B54/2D4/46CD7AA5/1872">
               <presentationElements>
                 <StateMachineGE presentable="!ed/1A44/3B54/2D4/46CD8A7D/4133">
                   <position>
                     <Point x="1191" y="900"/>
                   </position>
                   <size>
                     <Size width="11879" height="9683"/>
                   </size>
                 </StateMachineGE>
                 <StateGE presentable="!ed/4E87/3B54/2D4/46CD8F85/153B">
                   <position>
                     <Point x="5027" y="1323"/>
                   </position>
                   <size>
                     <Size width="4181" height="2619"/>
                   </size>
                 </StateGE>
                 <StateGE presentable="!ed/4E9B/3B54/2D4/46CD9062/21BA">
                   <position>
                     <Point x="2223" y="4842"/>
                   </position>
                   <size>
                     <Size width="9710" height="5186"/>
                   </size>
                 </StateGE>

...
```

(continued)

```
<StateMachineGE presentable="!ed/5056/3B54/2D4/46CDA682/2409">
  <position>
    <Point x="2540" y="5794"/>
  </position>
  <size>
    <Size width="8996" height="3969"/>
  </size>
</StateMachineGE>
<StateGE presentable="!ed/5062/3B54/2D4/46CDA694/1CA5">
  <position>
    <Point x="2778" y="6932"/>
  </position>
  <size>
    <Size width="3625" height="2011"/>
  </size>
</StateGE>
<StateGE presentable="!ed/5093/3B54/2D4/46CDA69F/44A8">
  <position>
    <Point x="7964" y="6985"/>
  </position>
  <size>
    <Size width="3281" height="1931"/>
  </size>
</StateGE>
<TransitionGE presentable="!ed/50BD/3B54/2D4/46CDA84D/280C">
  <positions>
    <Point x="5027" y="2646"/>
    <Point x="3704" y="3731"/>
    <Point x="3995" y="4286"/>
    <Point x="3942" y="4842"/>
  </positions>
  <labelSize>
    <Size width="2566" height="318"/>
  </labelSize>
  <labelPos>
    <Point x="1535" y="3069"/>
  </labelPos>
  <slashPos>
    <Point x="3308" y="3069"/>
  </slashPos>
</TransitionGE>
<TransitionGE presentable="!ed/50FC/3B54/2D4/46CDAB8C/6910">
  <positions>
    <Point x="10425" y="4842"/>
    <Point x="9869" y="3334"/>
    <Point x="9684" y="2937"/>
    <Point x="9208" y="2540"/>
  </positions>
```

...

(continued)

```
      <labelSize>
        <Size width="2566" height="635"/>
      </labelSize>
      <labelPos>
        <Point x="9790" y="3122"/>
      </labelPos>
      <slashPos>
        <Point x="11563" y="3122"/>
      </slashPos>
    </TransitionGE>
    <TransitionGE presentable="!ed/5101/3B54/2D4/46CDAB93/6688">
      <positions>
        <Point x="6403" y="7144"/>
        <Point x="6959" y="6720"/>
        <Point x="7461" y="6826"/>
        <Point x="7964" y="7170"/>
      </positions>
      <labelSize>
        <Size width="4286" height="1588"/>
      </labelSize>
      <labelPos>
        <Point x="4498" y="6138"/>
      </labelPos>
      <slashPos>
        <Point x="7991" y="6138"/>
      </slashPos>
    </TransitionGE>
    <TransitionGE presentable="!ed/5104/3B54/2D4/46CDAB96/52FC">
    <positions>
       <Point x="7964" y="8731"/>
       <Point x="7303" y="9075"/>
       <Point x="6879" y="8969"/>
       <Point x="6403" y="8678"/>
    </positions>
      <labelSize>
        <Size width="4445" height="635"/>
      </labelSize>
      <labelPos>
        <Point x="5292" y="9340"/>
      </labelPos>
      <slashPos>
        <Point x="8943" y="9340"/>
      </slashPos>
    </TransitionGE>
```

...

(continued)

```
            <TransitionGE presentable="!ed/5104/3B54/2D4/46CDAB96/52FC">
            <positions>
               <Point x="7964" y="8731"/>
               <Point x="7303" y="9075"/>
               <Point x="6879" y="8969"/>
               <Point x="6403" y="8678"/>
            </positions>
              <labelSize>
                <Size width="4445" height="635"/>
              </labelSize>
              <labelPos>
                <Point x="5292" y="9340"/>
              </labelPos>
              <slashPos>
                <Point x="8943" y="9340"/>
              </slashPos>
            </TransitionGE>
            <EquationGE presentable="!ed/51AA/3B54/2D4/46CDB545/13A6">
              <position>
                <Point x="7514" y="2566"/>
              </position>
              <size>
                <Size width="1006" height="503"/>
              </size>
            </EquationGE>
          <EquationGE kind="OBJ_INPUT" presentable="!ed/51C5/3B54/2D4/46CDB559/6EED">
               <position>
                 <Point x="5027" y="7938"/>
               </position>
               <size>
                 <Size width="1006" height="502"/>
               </size>
            </EquationGE>
          <EquationGE kind="OBJ_INPUT" presentable="!ed/51CC/3B54/2D4/46CDB55B/702C">
               <position>
                 <Point x="9975" y="7885"/>
               </position>
               <size>
                 <Size width="1005" height="502"/>
               </size>
            </EquationGE>
```

...

(continued)

```
<EquationGE kind="OBJ_LIT" presentable="!ed/51D9/3B54/2D4/46CE85C0/6C16">
   <position>
     <Point x="5609" y="2672"/>
   </position>
   <size>
     <Size width="767" height="318"/>
   </size>
 </EquationGE>
 <Edge leftVarIndex="1" rightExprIndex="1"
  srcEquation="!ed/51D9/3B54/2D4/46CE85C0/6C16"
  dstEquation="!ed/51AA/3B54/2D4/46CDB545/13A6">
   <positions>
     <Point x="6376" y="2831"/>
     <Point x="7038" y="2831"/>
     <Point x="7038" y="2831"/>
     <Point x="7514" y="2831"/>
   </positions>
 </Edge>
<EquationGE kind="OBJ_LIT" presentable="!ed/51F2/3B54/2D4/46CE86D1/162C">
   <position>
     <Point x="2937" y="8043"/>
   </position>
   <size>
     <Size width="1270" height="318"/>
   </size>
 </EquationGE>
 <Edge leftVarIndex="1" rightExprIndex="1"
  srcEquation="!ed/51F2/3B54/2D4/46CE86D1/162C"
  dstEquation="!ed/51C5/3B54/2D4/46CDB559/6EED">
   <positions>
     <Point x="4207" y="8202"/>
     <Point x="4683" y="8202"/>
     <Point x="4683" y="8202"/>
     <Point x="5027" y="8202"/>
   </positions>
 </Edge>
```

...

(continued)

```
                  <EquationGE kind="OBJ_LIT" presentable="!ed/523C/3B54/2D4/46CE86DB/4681">
                     <position>
                       <Point x="8017" y="7990"/>
                     </position>
                     <size>
                       <Size width="1217" height="318"/>
                     </size>
                   </EquationGE>
                   <Edge leftVarIndex="1" rightExprIndex="1"
                    srcEquation="!ed/523C/3B54/2D4/46CE86DB/4681"
                    dstEquation="!ed/51CC/3B54/2D4/46CDB55B/702C">
                     <positions>
                       <Point x="9234" y="8149"/>
                       <Point x="9684" y="8149"/>
                       <Point x="9684" y="8149"/>
                       <Point x="9975" y="8149"/>
                     </positions>
                   </Edge>
                 </presentationElements>
               </NetDiagram>
             </diagrams>
           </ed:Operator>
         </pragmas>
       </Operator>
     </declarations>
     <pragmas>
       <ed:Package oid="!ed/19F4/3B54/2D4/46CD79CE/6462" xmlns="http://www.esterel-
technologies.com/
      ns/scade/pragmas/editor/1">
          <diagrams>
            <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/19F5/3B54/2D4/
46CD79CE/4C7B"
           blockKind="constants"/>
            <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/19F6/3B54/2D4/
46CD79CE/106E"
           blockKind="types"/>
            <TreeDiagram landscape="true" format="A4 (210 297)" oid="!ed/19F7/3B54/2D4/
46CD79CE/2B10"
           blockKind="sensors"/>
</diagrams>
```

Specific Information

The table below describes XML attributes associated with specific SCADE Suite objects:

Table 2.13: Specific XML attributes

Object	Attribute	Description
Type	symbolFiles	For a given type, this attribute is a list of strings giving all symbols associated with a variable of this type, depending on its kind
Variable	hidden	When an operator input is defined as hidden (*i.e.,* its hidden attribute is set to true), it appears graphically as a cross at the bottom of the graphical representation of the operator
Operator	symbolFile	This attribute is used to specify the path of the symbol file used to represent the operator
	category	This attribute is used to specify an annotation category for this operator calls
Any	oid	This attribute allows to specify a unique identifier to an object. It is used for example to associate graphical information to semantic information, but it is also used by the annotation service, SCADE LifeCycle ALM Gateway, or the Diff tool.
	comment	This attribute is used to store the comments entered in the properties of a given SCADE Suite element
	commenttoC	This Boolean attribute is used to propagate comments to the code (it is then set to true)

3 / Specific SCADE Suite Files

This chapter describes the general purpose and the content of specific files used in SCADE Suite projects.

- "Symbol Files"
- "Option Files"

Symbol Files

A symbol file (*.ssl* extension) contains the description of the graphical representation of any operator call, input reference, output reference or local variable.

The graphical representation pertaining to the reference of a variable is specific to its type and its definition as In or Out. Its representation is framed by a rectangle and positioned at point (0,0) in the upper left corner. Such a graphical representation is useful for the graphical description of an operator. The following sections detail the syntax used to describe graphical representation in SCADE Suite.

- "Lexical Aspects and Conventions"
- "Language Syntax and Graphical Representation"
- "Naming Rules"
- "Search Rules"
- "Creation Rules"

Lexical Aspects and Conventions

Lexical aspects and conventions are coming from Scade language syntax.

IDENTIFIERS

Identifiers consist of a sequence of character strings between double quote marks including no space, tabulation, carriage return nor double quote marks.

NUMBERS

There are two types of numbers: integers and reals.

- Integers as sequence of numbers, possibly prefixed by minus sign (-). Example: 7.
- Reals as sequence of numbers followed by a dot and a sequence of numbers, possibly prefixed by minus sign (-). Example: -52.77.

KEYWORDS

The identifiers listed in the table below are language keywords required in uppercase.

ARC	**CIRCLE**	**HIDDEN**	**IMAGE**
INPUT	**LINE**	**OUTPUT**	**BMP**
POLYGON	**RECT**	**TEXT**	**JPG**
FONT	**BEZIER**	**ROUNDRECT**	**EMF**

CONVENTIONS

The syntax used to describe Scade language constructions is specified in extended Backus-Naur form (EBNF). Its specification consists in a set of rules (or productions) forming a “grammar” indicating how the language is phrased. Each rule consist of a non-terminal symbol and an EBNF expression

separated by an equal sign and terminated with a period. The non-terminal symbol is defined with an EBNF expression. The EBNF expression is composed of any number, terminal symbols, non-terminal symbols and meta-symbols whose meaning is provided in the table below.

Notation	Meaning	Notation	Meaning
=	is defined as	{X}	0 occurrence of X or more
|	alternative	(X|Y)	grouping: either X, or Y
.	end of production	”XYZ”	terminal symbol XYZ
[X]	0 or 1 occurrence of X	<*xyz*>	non-terminal symbol *xyz*

The following conventions are adopted:

<ident> indicates an identifier:

- identifier “#N” represents the name of the called operator
- identifier “#NL” represents the name of the called operator with a left alignment
- identifier “#NR” represents the name of the called operator with a right alignment
- identifier “#NP” represents the size parameter instantiation of the operator within “<< >>”
- identifier “#O” represents the occurrence number of the called operator
- identifier “#H_i” represents the assigned value of the ith hidden input
- identifier “#S” represents the operator signature
- identifier “#R” is used to define “glass symbol” (the content of the corresponding called operator is visualized through the symbol)

<int_value> indicates the value of an integer

<real_value> indicates the value of a real

<full_transparent> indicates integer value 0 or 1 (full= 0, transparent = 1)

<direction> indicates one of the character string “N”, “E”, “W”, “S”

<order> indicates an integer value representing the front to back order

<font number> indicates an integer representing the font number used to display text

FONT FILE SYNTAX

The font corresponding to the font number is specified in a **font file**. The font file used is <*project path*>*.fonts* or, by default, the file located at *SCADE/tvg/ffonts* of the SCADE installation. The syntax is the following:

```
<font number> "<font name>" <font size> <"B"|"I"|"U">
```

Possible command options and parameters:

- **# <line>** can be used as comment
- **<font name>** indicates a font name
- **<font size>** indicates a font size (integer)
- **<font style>** is a character string formed with letters < "B" | "I" | "U" >. B, I, and U stand respectively for bold, italic and underlined, and can be combined.
- **<rotation>** indicates the rotation of the image in degree (0, 90, 180, 270)
- **<image_kind>** is the type of image of the element (BMP for bitmap, JPG for jpeg, or EMF for EnhancedMetaFile)
- **<image_path>** is the pathname of the file containing the image. If the pathname is relative, it is relative compared to the symbol file (ssl) containing this instruction

Language Syntax and Graphical Representation

The following illustrates the language syntax:

```
<repr_graph> = <object> { <object> } .

<object>     = <arc>
               | <circle>
               | <line>
               | <rect>
               | <roundrect>
               | <text>
               | <polygon>
               | <bezier>
               | <io_input>
               | <io_output>
               | <io_hidden>
               | <image> .
<arc>        =  "ARC" "(" <pos> "," <pos> "," <pos> "," <pos> ["," <full_transparent> ["," 
<order>]]")"] ";".
<circle>     =  "CIRCLE" "(" <pos> "," <pos> "," <pos> "," <pos> ["," <full_transparent>["," 
<order>]] ")" ";".
<line>       =  "LINE" "(" <pos> "," <pos> "," <pos> "," <pos> ")"  ";"  .
<rect>       =  "RECT" "(" <pos> "," <pos> "," <pos> "," <pos> ["," <full_transparent>] ["," 
<order>] ")" ";".
<roundrect>  =  "ROUNDRECT" "(" <pos> "," <pos> "," <pos> "," <pos> ","<width> ","<height>
                ["," <full_transparent>["," <order>]] ")" ";".
<polygon>    =  "POLYGON" "(" <poly_pos> { "," <poly_pos>  } "," <full_transparent> ["," 
<order>]")" ";" .
<bezier>     =  "BEZIER"  "(" <poly_pos> { "," <poly_pos>  } "," <full_transparent> ["," 
<order>]")"  ";"
<poly_pos>   =  "[" <pos>  "," <pos> "]"   .
<text>       =  "TEXT"  "(" <pos> "," <pos> "," <pos> "," <pos>","  <ident>  ")"  ";"  .
<text>      =  "TEXT"  "(" <pos> "," <pos> "," <pos> "," <pos>","  <ident>, <font number> ")"
";"  .
<text>       =  "TEXT"  "(" <pos> "," <pos> "," <pos> "," <pos>","  <ident>, "FONT" "("<font
ident>","
                <font size>","<font style> ")" ")"  ";"  .
<io_input>   =  "INPUT"  "(" <pos> "," <pos> ","  <direction>  ")"  ";"  .
<io_output>  =  "OUTPUT"  "(" <pos> "," <pos> ","  <direction>   ")"  ";"  .
<io_hidden>  =  "HIDDEN"  "(" <pos> "," <pos> ","  <direction> ")"  ";"  .
<pos>        =  <int_value>
                | <real_value>
<image>      =  "IMAGE" "(" <pos> "," <pos> "," <pos> "," <pos>"," <rotation>","<image_kind>","
                <image_path> ["," <order>]")" ";" .
```

<width> and **<height>** specify the width and height of the ellipse used to draw rounded corners (in logical coordinates).

Naming Rules

The name of a symbol file associated with a variable must comply with naming rules:.

If graphical representation is ...	then file is named as follows...
Reference to an input of “type” type	File name is based on the name of the type, followed by `_in_io` and ending with SSL extension
Reference to an output of “type” type	File name is based on the name of the type, followed with `_out_io` and ending by SSL extension
Input reference to a local variable	File name is based on the name of the type, followed with `_in_var` and ending by SSL extension
Output reference to a local variable	File name is based on the name of the type, followed with `_out_var` and ending by SSL extension
Input reference to a sensor	File name is based on the name of the type, followed with `_in_sensor` and ending by SSL extension

Search Rules

The search rules for symbols are the following:

- If the **Graphical symbol directories** option is set (**General** tab of the **Options** window from *Tools* > *Options*), symbols are searched in each of the specified directory in order of appearance (directories are separated by comas).
- If there is no search result or if the above option is empty, symbols are searched in the same directory as the `.xscade` file of the associated node (type for variables).
- If no symbol defined, the default symbol distributed in product installation at *SCADE/SCADE/tvg/symbols* is used.

Creation Rules

If no user symbol is associated with a node, applying changes from **Symbol Editor** creates a new file containing the new description of the symbol. By default, this file is created in the same directory as the `.xscade` file of the associated node (or type for variables). If **Create in specific symbol directory** option is checked (see "Selecting General Options" on page 75 in *SCADE Suite User Manual*), the file is created in the first directory specified by **Graphical symbol directories**. To specify symbols for the variable of predefined types, it is necessary to enable **Create in specific symbol directory**. It is not possible to specify or edit the symbol of any type variable contained in a library.

Option Files

The styles and options of SCADE Suite projects are stored in option files (`.tot` extension) that can be inserted in projects through the **File View**. The content of option files represent user preferences which can be inherited with redefinition or not.

A powerful means to create, delete, or modify styles consists in creating, editing, and removing TOT files. Styles saved in a TOT file are reusable in projects that include the file. Removing a file from a project deletes the styles it contains.

Note

If several projects are open, all styles of any given project are available to all others.

When a project contains several option files, their order indicates their respective priorities but specific options can be forced. Even if a project includes option files with higher priority, forced options are always taken into account. For user instructions, consult "Selecting Advanced Options" on page 82 in *SCADE Suite User Manual*.

Option files are useful for sorting preferences in decreasing order of priority, as well as for blocking or hiding values and files.

An option file is a sequence of keys and values. Operations are organized like a tree. Each node in the tree is a key, and for each node you can specify a new value for any option attached to this key. Here is an example of option file content:

```
[Scade/Preferences/General/FilesManagement]
"UseCreationDialogBox"="true"
"PropagateNameChanges"="true"
"AssocCreationFile"="false"

[Scade/Preferences/View/NetView]
"DisplayInternalTypes"="true"

[Studio/Settings]
"ShowAdvancedOptionPage"="true"

[Studio/Settings/Scc]
"UseScc"="true"
```

The path to the key is in square brackets and the option name is in quotes. Only the value that differs from the default value of an option is saved in the file.

4 / Scripting Capabilities in SCADE Environments

This chapter introduces the various scripting capabilities available within SCADE Suite, SCADE Architect, or SCADE Test environments.

- "SCADE Scripting Capabilities"
- "Registering Customization Scripts"
- "Custom Script Example"

SCADE Scripting Capabilities

SCADE Suite, SCADE Architect, and SCADE Test support various scripting capabilities. These SCADE environments can be used to open Python/Tcl files and edit scripts with the commands they support.

SCRIPTS FOR ACCESSING SCADE MODEL DATA

The access to SCADE model data via SCADE APIs can be performed using Python or Tcl scripts. Such scripts can be executed in batch or directly in SCADE Suite, SCADE Architect, or SCADE Test environments.

The commands supported for accessing model data via SCADE APIs are available from:

- "API-Related Python Commands" on page 94 in *SCADE Python API Guide*
- "API-Related Tcl Commands" on page 67 in *SCADE Java/Tcl API Guide*

SCRIPTS FOR DISPLAYING CONTENT IN SCADE USER INTERFACE

The display of information from API model data into SCADE Suite, SCADE Architect, or SCADE Test user interfaces can be performed using Python or Tcl scripts. Such scripts can be executed directly in SCADE environments.

The commands supported for displaying content in SCADE Suite, SCADE Architect, or SCADE Test are available from:

- "GUI Display-Related Python Commands" on page 97 in *SCADE Python API Guide*
- "GUI Display-Related Tcl Commands" on page 76 in *SCADE Java/Tcl API Guide*

SCRIPTS FOR USER INTERFACE CUSTOMIZATION

The customization of SCADE Suite, SCADE Architect, and SCADE Test with additional user interface components can be performed via Python or Tcl scripts. Such customization scripts must be registered as external components in SCADE environment for proper evaluation at software launch.

The commands supported for customizing SCADE Suite, SCADE Architect, and SCADE Test environments are available from:

- "Customization-Related Python Commands" on page 108 in *SCADE Python API Guide*
- "Customization-Related Tcl Procedures" on page 124 in *SCADE Java/Tcl API Guide*

SCRIPTS DEDICATED TO SCADE SUITE CHECKER

The execution of check sessions on SCADE Suite models can be performed via Tcl scripts. Such scripts can be executed in batch or directly in SCADE Suite.

The commands supported for controlling SCADE Suite Checker are available from: "Checker-Related Tcl Commands" in *SCADE Java/Tcl API Guide*.

SCRIPTS DEDICATED TO SCADE LIFECYCLE REPORTER

The customization of SCADE LifeCycle Reporter for SCADE Suite models can be performed via Tcl scripts. These scripts must be registered via **Report script** option in the Reporter tool settings (see "Setting Report Generation" on page 29in *SCADE LifeCycle User Manual*).

The commands supported for customizing project documentation are available from: "Reporter-Related Tcl Commands" in *SCADE Java/Tcl API Guide*.

SCRIPT EXECUTION

Except customization-related Tcl procedures and Python commands, you can execute all other Tcl and Python commands directly in SCADE Suite, SCADE Architect, or SCADE Test environments with the help of the **Script Wizard** available from the **Script** toolbar.

- To execute scripts with Tcl commands, consult "Executing Tcl Scripts" on page 172 in *SCADE Java/Tcl API Guide*.
- To execute scripts with Python commands, consult "Executing Python Scripts" on page 29 in *SCADE Python API Guide*.

Customization-related Python commands or Tcl procedures allow for creating graphical user interface extensions. You cannot execute them directly with the Script Wizard. The final script file must be registered and evaluated as an extension of SCADE environment as detailed in "Registering Customization Scripts" on page 77. The registration enables to activate automatically the custom user interface component created by the script at SCADE Suite, SCADE Architect, or SCADE Test launch.

SCRIPT CREATION BASED ON SCRIPT WIZARD

SCADE Suite Script Wizard enables you to navigate in SCADE Suite, SCADE Architect, or SCADE Test UML models and generate scripts. You can reuse such scripts generated from SCADE project model structure into your own scripts.

For details about script creation, consult either:

- Chapter 5 about "Specific Commands for Python Scripting" in *SCADE Python API Guide*
- Chapter 4 about "Specific Commands for Tcl Scripting" in *SCADE Java/Tcl API Guide*

Registering Customization Scripts

The evaluation of registered scripts (Tcl or Python) dedicated to adding user interface (UI) components is performed when launching SCADE Suite, SCADE Architect or SCADE Test, or whenever a script file is modified. You must register customization scripts using one of the following methods:

- "Registering Scripts from User Interface"
- "Registering Scripts in Batch Command"
- "Registering Customization Scripts from File"
- "Removing Customization Scripts"

Customization scripts can also be registered in SCADE environment by means of Python packages using SCADE Extension Manager. For details, consult Chapter 6 about "Managing SCADE Extensions" in *SCADE Python API Guide*.

Registering Scripts from User Interface

You can register customization scripts directly from SCADE Suite, SCADE Architect, or SCADE Test to enable custom UI components at product launch.

To register customization scripts

1 Select *Insert* > *Components* > *Custom* to open **Custom Registration**.

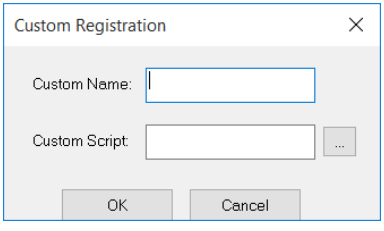


Figure 4.1: Custom Registration dialog

2 Type a name in **Custom Name**.

3 Click browsing button and set the script file path in **Custom Script**.

4 Click **OK**. Close SCADE and launch it again. The script is automatically registered and evaluated at application start or upon script file update.

Registering Scripts in Batch Command

It is also possible to register customization scripts with the following batch command:

```
TCLLNC -batch "<install path>/SCADE/scripts/CustomRegisterBatch.tcl" "<Custom
Name" "<Custom Script Path>"
```

Registering Customization Scripts from File

Customization scripts can be registered using a file (`.srg` extension). Once registered, scripts are called at SCADE Suite launch.

To save the script for a specific SCADE version, save the file in this system file location:

```
%APPDATA%/Scade/Customize/<Product Version Number>/
```

To save the script for any SCADE version, save the file in this system file location:

```
%APPDATA%/Scade/Customize/
```

The SRG file must contain the following to register a custom script:

```
[Studio/Work Interfaces/<Custom Name>]
"Pathname"="ETCUST.DLL"
[Custom/Studio/Extensions/<Custom Name>]
"Pathname"="<script full pathname>"
```

Removing Customization Scripts

Delete the SRG file in either of the specified locations to uninstall custom scripts.

Custom Script Example

A custom Python script is delivered as an example in SCADE installation. This script demonstrates various UI capabilities.

The P**`ythonGuiExample.py`** script file is located in `<install>/SCADE/APIs/Python/examples/custom`. Additional files are located in the same folder.

Note

A `ReadMe.txt` file provides some explanations about the script and how to register and use it. An `install.bat` file is provided for automatic installation and a `PythonExample.srg` file is provided for manual installation.

The script is fully commented to ease understanding and usage. It implements the following capabilities:

- A *Custom script demo* menu added to the main *Tools* menu of SCADE.
- A **Custom script demo** toolbar with the same commands (not visible by default, user must display it explicitly the first time).
- Commands called from both menu and toolbar.
- A *Browse directory* contextual menu available in **Scade view** only.
- A dialog box, opened from a menu/toolbar button, showing a wide variety of widgets usage.
- A **Custom script demo** property page available when a SCADE Suite model or a SCADE Suite package is selected. The property page uses widgets that are dealing with SCADE objects.
- Usage of common **Open**, **Save**, and **Choose directory** dialogs.
- Toggle button/menu item.

Ansys

Technical Manual

Part 2

Simulator

- Chapter 5: "Dedicated Simulation Functions"
- Chapter 6: "SCADE Suite Simulation Batch"
- Chapter 7: "Data Format for SCADE Suite Simulation"
- Chapter 8: "Simulation Command Line Errors"

5 / Dedicated Simulation Functions

This chapter describes the syntax of available Simulator commands and details the usage and structure of the conversion/comparison functions necessary for simulation. It also describes and illustrates the public API functions provided for running simulation remotely from external applications, as well as the Test Services API functions provided for external testing environments.

- "Simulator Commands"
- "Conversion and Comparison Functions for Types"
- "Conversion Functions for Imported Operators"
- "Public API Functions for Co-Simulation Mode"

Simulator Commands

Various commands and variables are supported by SCADE Suite Simulation for running simulation sessions. They can also be extended with Tcl scripting commands. The syntax of these commands and variables is described below.

- "Simulation-Specific Commands"
- "Simulation-Specific Variables"
- "Simulator Tcl Extension"

Simulation-Specific Commands

Specific commands can monitor simulation executables in SCADE Suite. Such commands can be used in `.sss` scenario files, in command line, or in breakpoint expressions. Command arguments in brackets are optional.

For details about simulation path format necessary for some of the specific commands, see "Simulation Path Format" on page 159. The table below lists all available commands and specifies their context of usage.

Table 5.1: List of simulation specific commands and usage context

Command name	SSS scenario file	Interactive command line	Breakpoint expr.
"SSM::alias"	X	X	
"SSM::alias_value"	X	X	
"SSM::back"		X	
"SSM::check"	X	X	
"SSM::cycle"	X	X	
"SSM::default_inputs"	X	X	X
"SSM::default_value"	X	X	X
"SSM::default_value_matching"	X	X	X
"SSM::get"	X	X	X
"SSM::get_active_branches"	X	X	X
"SSM::get_active_matches"	X	X	X
"SSM::get_active_states"	X	X	X
"SSM::get_cycle"	X	X	X
"SSM::get_fired_strong_transitions"	X	X	X
"SSM::get_fired_weak_transitions"	X	X	X

Table 5.1: List of simulation specific commands and usage context (Continued)

Command name	SSS scenario file	Interactive command line	Breakpoint expr.
"SSM::get_obs_list"	X	X	X
"SSM::get_one_obs_list"	X	X	X
"SSM::get_outputs"	X	X	X
"SSM::go"		X	
"SSM:print"		X	
"SSM::print_all_obs"		X	
"SSM::print_obs"		X	
"SSM::print_watches"		X	
"SSM::print_watch_values"		X	
"SSM::random_bool"	X	X	X
"SSM::random_char"	X	X	X
"SSM::random_cycle"	X	X	
"SSM::random_inputs"	X	X	X
"SSM::random_int"	X	X	X
"SSM::random_real"	X	X	X
"SSM::random_sign"	X	X	X
"SSM::random_value"	X	X	X
"SSM::random_value_matching"	X	X	X
"SSM::re_alias"	X	X	
"SSM::reset"		X	
"SSM::set_breakpoint"		X	

Table 5.1: List of simulation specific commands and usage context (Continued)

Command name	SSS scenario file	Interactive command line	Breakpoint expr.
"SSM::set"		X	
"SSM::set_inputs"	X	X	
"SSM::set_inputs_file"		X	
"SSM::set_tolerance"	X	X	
"SSM::type"	X	X	X
"SSM::uncheck"	X	X	
"SSM::unset_breakpoint"		X	
"SSM::unset_inputs_file"		X	
"SSM::unwatch"		X	
"SSM::unwatch_node"		X	
"SSM::watch"		X	
"SSM::watch_node"		X	
"SSM::what"		X	

SSM::ALIAS

Command and description:

```
SSM::alias <alias> <path>
```

Creates an alias to the specified variable to access to it more easily.

- `<alias>` is the alias name using the following regular expression: [a-zA-Z0-9_]+
- `<path>` is a path to a settable/observable data: (part of) input, output, sensor, probe of the operator under test. See syntax details in "Simulation Path Format" on page 159.

SSM::ALIAS_VALUE

Command and description:

```
SSM::alias_value <alias> <val>
```

Creates an alias to the specified value to access to it more easily from `SSM::check` or `SSM::set` commands.

- `<alias>` is the alias name using the following regular expression: [a-zA-Z0-9_]+
- `<val>` is either a literal (predefined type, enumeration, array, structure), a constant, or an expression (lambda/range)

SSM::BACK

Command and description:

```
SSM::back n
```

Returns n cycles backwards.

Example:

```
[22]>SSM::back 3
[19]>
```

SSM::CHECK

Command and description:

```
SSM::check <path> <val> {sustain=(forever|<integer>) | real=<tolerance>}
```

Registers a test of observable data (outputs, active probes, or part of their inner structure) against a specified value and according to a tolerance applied to any real part of the data.

- <path> is a path to an observable data (see syntax in "Simulation Path Format" on page 159)
- <val> is the expected value
- **sustain**= (optional)
  - <integer> must be strictly greater than 0.
  - If forever keyword used instead of <integer>, the test is performed until the end of the record.
  - If no **sustain** argument specified, the test is registered only for the next cycle.
- **real**= (optional)
  - <tolerance> defined as <integer>|<real>|<extended real> [r] to set to any real part of <var> declared after the instruction unless specified otherwise in the test declaration, and until a new tolerance is set, or until the end of the current record registered test is performed by the next calls to SSM::cycle instruction and after every execution of steps during the next set of cycles
  - If no **real** argument specified, the test of real type data is performed according to the tolerance currently set to the tested data by **SSM::set_tolerance** instruction.
  - Any notation involving a % trailing character for tolerance value in SSS file format is deprecated but supported for backward compatibility.
- values declared by this command are verified by test criterion evaluation.

Available command parameters can be defined as follows:

<**val**> = <value> | **{**<value>**}**

- curly brackets around <value> are mandatory when the value contains at least one <leaf value> matching <string> or <char>

<**value**> = <**leaf value**> | **(**<**value list**>**)**
<**leaf value**> = <bool>|<integer>|<real>|<extended real>|<string>|<char>|<enum>|?|<empty>

- <enum> is the pathname of a value in a SCADE Suite enumerated type
- ? deactivates any active check on the corresponding data part (same capability as ssm::uncheck) (e.g., check Y (1,?,2))
- <empty> declares no value changes but keeps value from previous ssm::set or ssm::check (e.g., set X (1, ,2))

<**value list**> = <value> | <value>, <value list>

If a test is defined for an element for which a test is already registered, the new test replaces or partially replaces the first one when applying to a parent data. For instance, when `SSM::check` is performed on a structured variable, then again on one element of the same structure, the second check replaces the first one on the structure element whereas other elements of the structure variable are still checked by the first `SSM::check`. Any notation involving a % trailing character for tolerance value in SSS file format is deprecated but supported for backward compatibility.

The verification of actual values with respect to expected values declared in SSS files (via `ssm::check`) is performed by the evaluation of test criterion as follows:

If Scade type of tested data is of real predefined type:

- absolute tolerance defined: `|actual_value - expected_value| <= tolerance`
- relative tolerance defined: `|actual_value - expected_value| <= tolerance*expected_value`

If Scade type of tested data is of any other predefined type:

- exact comparison

Note

In case no tolerance is specified for testing real values, a default tolerance is used to perform comparison. The default absolute tolerance is 2^{-1022} when the `real` predefined type is defined as double or 2^{-126} when the `real` predefined type is defined as `float`.

SSM::CYCLE

Command and description:

```
SSM::cycle [n]
```

Executes n cycles if n is present; n is a positive integer; n default value is 1. After each cycle, the declared stop conditions are tested. After n cycles, the values of the traced variables are displayed.

Example 1: executes 5 cycles.

```
[19]>SSM::cycle 5
[24]>
```

Example 2: executes a number of cycles equivalent to twice the value of the variable `nb_cycles` (*i.e.*, 14 in that example).

```
[24]>set nb_cycles 7
[24]>SSM::cycle [expr $nb_cycles*2]
[38]>
```

SSM::DEFAULT_INPUTS

Command and description:

```
SSM::default_inputs
```

Returns a character string that represents a default input vector of the simulated node. Default values of basic types are listed below:

- Int: 0
- Bool: false
- Char: 0
- Real: 0.0

Example:

```
[0]>SSM::set_inputs [ SSM::default_inputs ]
```

SSM::DEFAULT_VALUE

Command and description:

```
SSM::default_value type_signature
```

Returns a character string that represents the default value matching `type_signature`. Default values are the same as those generated by the `SSM::default_inputs` command. The argument should conform to the type signatures generated by the `SSM::type` command.

Example:

```
[0]> puts [ SSM::default_value (I,R,(B,R,I)) ]
(0, 0.0, ( f, 0.0, 0 ) )
```

SSM::DEFAULT_VALUE_MATCHING

Command and description:

```
SSM::default_value_matching obs_var
```

Returns a character string that represents the default value matching the type of the observable variable `obs_var`. Default values are the same as those generated by the `SSM::default_inputs` command.

SSM::GET

Command and description:

```
SSM::get var_name
```

Returns the value of the SCADE variable `var_name` in the form of a character string. This command can apply to all observable variables.

Example 1: This one assigns the value of the SCADE v1 variable to the v1 variable. Naming spaces and Simulator are distinct.

```
[0]> set v1 [ SSM::get v1 ]
```

Example 2: This one displays the value of the SCADE v2 variable..

```
[0]> puts [ SSM::get v2]
3
```

SSM::GET_ACTIVE_BRANCHES

Command and description:

```
SSM::get_active_branches
```

Returns a list with the path of all `if` conditional blocks active branches.

SSM::GET_ACTIVE_MATCHES

Command and description:

```
SSM::get_active_matches
```

Returns a list with the path of all `when` conditional blocks active matches.

SSM::GET_ACTIVE_STATES

Command and description:

```
SSM::get_active_states
```

Returns a list with the path of all active states.

SSM::GET_CYCLE

Command and description:

```
SSM::get_cycle
```

Returns the current cycle number.

SSM::GET_FIRED_STRONG_TRANSITIONS

Command and description:

```
SSM::get_fired_strong_transitions
```

Returns a list with the path of all fired strong transitions.

SSM::GET_FIRED_WEAK_TRANSITIONS

Command and description:

```
SSM::get_fired_weak_transitions
```

Returns a list with the path of all fired weak and synchro transitions.

SSM::GET_OBS_LIST

Command and description:

```
SSM::get_obs_list
```

Returns a list with the name of all observable SCADE variables in the current simulation session.

Example:

```
[38]>puts [SSM::get_obs_list]
Package1::Operator1/L Package1::Operator1/H Package1::Operator1/x Package1::Oper
ator1/y Package1::Operator1/del_x Package1::Operator1/del_y
[38]>
```

SSM::GET_ONE_OBS_LIST

Command and description:

```
SSM::get_one_obs_list <kind>
```

Returns a list with all the observable SCADE variables of the kind argument type in the current simulation session.

<kind> argument can have the following values:

- root_input: inputs of the simulated operator
- root_output: outputs of the simulated operator
- input: observable inputs of nodes different from the simulated operator
- output: observable outputs of nodes different from the simulated operator
- local: observable local variables of the model
- probe: observable probe variables of the model
- signal: observable signal variables of the model
- sensor: observable sensor variables of the model

Note

An error is raised if the <kind> argument does not belong to this set of values.

SSM::GET_OUTPUTS

Command and description:

```
SSM::get_outputs
```

Returns a character string composed of values of the simulated node outputs.

Example: Suppose the input signature is similar to the outputs one (same types in same order). This one assigns the output vector of the simulated node to the simulated node inputs.

```
[0]> SSM::set_inputs [SSM::get_ouputs]
```

SSM::GO

Command and description:

```
SSM::go n
```

Requests SCADE Simulator to go to cycle n. If n is greater than the current cycle, SCADE Simulator executes the number of cycles necessary for going to the n cycle. If not, it executes again the n first cycles which led to the current state of SCADE Simulator.

SSM:PRINT

Command and description:

```
SSM::print var_list
```

Displays the value of the SCADE variables mentioned in the `var_list` list. This list can be reduced to a single element.

Example:

```
[0]> SSM::print [ list a b ]
a = 1
b = ( 1, t)
[0]> SSM::print c
c = f
```

SSM::PRINT_ALL_OBS

Command and description:

```
SSM::print_all_obs
```

Displays the list of all variables observable in the current simulation session.

It is equivalent to the following command:

```
[0]> SSM::print_obs [list ri ro i o l m g]
```

SSM::PRINT_OBS

Command and description:

```
SSM::print_obs kind_list
```

For each variable type included in the `kind_list` list, displays the list of observable variable names (one per line). The values of the elements in the `kind_list` list are those supported by the `SSM::get_one_obs_list` command.

SSM::PRINT_WATCHES

Command and description:

```
SSM::print_watches
```

Displays the watched variables.

SSM::PRINT_WATCH_VALUES

Command and description:

```
SSM::print_watch_values
```

Displays the values of the watched variables. It is equivalent to the call to the `SSM:print` command on each viewed variable.

SSM::RANDOM_BOOL

Command and description:

```
SSM::random_bool
```

Returns a character string that represents a random boolean.

SSM::RANDOM_CHAR

Command and description:

```
SSM::random_char
```

Returns a character string that represents a random character.

SSM::RANDOM_CYCLE

Command and description:

```
SSM::random_cycle [n]
```

Performs n simulation cycles from n input vectors generated randomly (using SSM::random_inputs). If n is not specified, only 1 cycle is performed.

SSM::RANDOM_INPUTS

Command and description:

```
SSM::random_inputs
```

Returns a character string that represents a random input vector of the simulated operator.

Intervals of generated random values are:

- `Int`: [`-INT_MAX`, `INT_MAX`]
- `Bool`: [`true`, `false`]
- `Char`: [`-CHAR_MAX`, `CHAR_MAX`]
- `Real`: [`-1000.0`, `+1000.0`]

`INT_MAX` and `CHAR_MAX` are maximum values of an int and of a char respectively, as defined by the C language for a given architecture. The random Tcl generator can be initialized with a specific value through the command 'expr': `expr srand(value)`.

SSM::RANDOM_INT

Command and description:

```
SSM::random_int
```

Returns a character string that represents a random integer.

SSM::RANDOM_REAL

Command and description:

```
SSM::random_real
```

Returns a character string that represents a random real.

SSM::RANDOM_SIGN

Command and description:

```
SSM::random_sign
```

Returns a character string that represents a random sign (+1 or -1).

SSM::RANDOM_VALUE

Command and description:

```
SSM::random_value type_signature
```

Returns a character string that represents a random value matching `type_signature`. Random values are the same as those generated by `SSM::random_inputs`. The argument should conform to the type signatures generated by `SSM::type`.

Example:

```
[0]> puts [ SSM::random_value (I,R,(B,R,I)) ]
(387, -125.67, ( t, 876.98, -87568 ) )
```

SSM::RANDOM_VALUE_MATCHING

Command and description:

```
SSM::random_value_matching obs_var
```

Returns a character string that represents a random value matching the type of the `obs_var` observable variable. Random values are the same as those generated by `SSM::random_inputs`.

SSM::RE_ALIAS

Command and description:

```
SSM::re_alias <alias name>
```

Removes the alias created by `SSM::alias`. `<alias name>` specifies the alias to remove.

SSM::RESET

Command and description:

```
SSM::reset
```

Restarts Simulator with a call to the <simulated_node>_init function of the generated C code. Memory is not reset. It resets Boolean entries to false and int entries to 0. See also SSM::default_inputs.

SSM::SET_BREAKPOINT

Command and description:

```
SSM::set_breakpoint <kind> <expr>
```

Sets control, data, or Tcl breakpoints.

If <kind> = ...	<expr> = ...
control	"<control_path>" where <control_path> is either a state, a transition, an activate if branch, or an activate when match path.
data	"<instance_data_path>" "<comparator>" "<value>" where <instance_data_path> points to a variable
tcl	"<name>" "<tcl_expr>"

For details about path syntax, see "Simulation Path Format" on page 159.

Example 1: This one sets a breakpoint on the del_x output with the on-value condition "del_x = 8".

```
[0]>SSM::set_breakpoint data "Package1::Operator1/del_x" "=" "8"
```

Example 2: This one sets a breakpoint on StateMachine1 when State1 is true.

```
[0]>SSM::set_breakpoint control "Package1::Operator1/StateMachine1:State1"
```

SSM::SET

Command and description:

```
SSM::set <path> <val>
```

Assigns the value expression to the Scade <var> variable. It can only apply to observable and alterable Scade variables (*i.e.*, inputs or sensors).

- <path> is a path to a settable/observable entity: input, output, sensor, or probe of the operator under test. See syntax details in "Simulation Path Format" on page 159.

Available command parameters can be defined as follows:

```
<val> = <value> | {<value>}
```

- curly brackets around <value> are mandatory when the value contains at least one <leaf value> matching <string> or <char>

```
<value> = <leaf value> | (<value list>)
<leaf value> = <bool>|<integer>|<real>|<extended real>|<string>|<char>|<enum>|<?>|<empty>
```

- <enum> is the pathname of a value in a SCADE Suite enumerated type
- <empty> declares no value changes but keeps value from previous set or check command (e.g., ssm::set X (1, ,2))

```
<value list> = <value> | <value>, <value list>
```

Example 1: simply assigns the integer value 3 to the Scade Input1 variable.

```
[0]>SSM::set Operator1/Input1 3
```

Example 2: assigns an entire array or structure to the SCADE Input1 variable.

```
SSM::set Operator1/Input1 (true, false, true, false)
SSM::set Operator1/Input1 ((3.14, false), 42)
```

Example 3: assigns part of an array or structure to the SCADE Input1 variable based respectively on specific indexes or labels.

```
SSM::set Operator1/Input1 "{0:true, 2:false}"
SSM::set Operator1/Input1 "{label1:{l1abel2:true}, label2:42}"
```

Example 4: assigns a specific array index or structure label to the SCADE Input1 variable.

```
SSM::set {Operator1/Input1[1]} true
SSM::set {Operator1/Input1.label2} 43
```

SSM::SET_INPUTS

Command and description:

```
SSM::set_inputs value_list
```

Assigns the values of the `value_list` vector to all simulated node inputs. The format of this vector is that of the data-driven mode, but can be written as a list. If the number of list elements does not match the number of inputs in the simulated node, you get an error message. Similarly, if a type error occurs, an error message specifying both the expected and given types is displayed. Controls for assignments and types are performed consecutively: thus a partial assignment may be found before a type error is detected.

Example: assigns the three inputs 1, (2,3) and (3,4.0,t), of int, tuple[int, int] and tuple[int, real,bool] types respectively to the three inputs of the simulated node.

```
[0]> SSM::set_inputs 1 (2, 3) (3, 4.0, t)
[0]> SSM::set_inputs [list 1 (2, 3) (3, 4.0, t)]
```

SSM::SET_INPUTS_FILE

Command and description:

```
SSM::set_inputs_file file_name
```

Loads the `file_name` file as the input vectors source for future simulation cycles. All the calls to `SSM::cycle` or `SSM::go` read the inputs in `file_name`. If an input file has already been loaded, it is closed and replaced with the new one. When the file inputs are exhausted, the file is automatically closed (implicit call to `SSM::unset_inputs_file`).

SSM::SET_TOLERANCE

Command and description:

```
SSM::set_tolerance {[path=<path>] real=<tolerance>}
```

Assigns a test tolerance to the real part of observable data (outputs or probes).

- <path> is the pathname of an observable output or probe, or the pathname of an alias. See syntax details in "Simulation Path Format" on page 159.
- <tolerance> is the tolerance value to set to any real part of <var> declared after the instruction unless specified otherwise in test declaration, and until a new tolerance is set, or until the end of the current record (tolerance does not apply to parts of a variable)
- if no path argument set, tolerance is associated with the tests of all observable entities

SSM::TYPE

Command and description:

```
SSM::type var_name
```

Returns the type of var_name observable variable in a string. Possible values: I (integer), B (Boolean), R (real), C (character), or a bracketed combination of these atomic types.

Example: Say var_1 is tuple of 3 integers and var_2 is structured type (3 reals):

```
[0]> puts [ SSM::type var_1 ]
(I,I,I)
[0]> puts [ SSM::type var_2 ]
(R,R,R)
```

SSM::UNCHECK

Command and description:

```
SSM::uncheck <path>
```

Unregisters active test for observable data (output, probe, part of inner structure). When applied to data with test declared higher in parent data, only the test of the "unchecked" part is unregistered.

- <path> is a path to observable data. See syntax in "Simulation Path Format" on page 159.

SSM::UNSET_BREAKPOINT

Command and description:

```
SSM::unset_breakpoint expr
```

Unset control, data or Tcl breakpoints previously set.

If the breakpoint to unset is...	<expr> = ...
a *control* breakpoint	"<control_item_path>"
a *data* breakpoint	"<data_item_path>" ["<comparator>" "<value>"]
a *tcl* breakpoint	"<name>"

For details about path syntax, see "Simulation Path Format" on page 159.

Example: Unsets the breakpoints previously set using the `SSM::set_breakpoint` command.

```
[0]>SSM::unset_breakpoint "Package1::Operator1/del_x" "=" "8"
[0]>SSM::unset_breakpoint "Package1::Operator1/StateMachine1:State1"
```

SSM::UNSET_INPUTS_FILE

Command and description:

```
SSM::unset_inputs_file
```

Closes the current input file if there is one and disables the automatic reading of input vectors from this file. If new input values are not given, the next calls to `SSM::cycle` or `SSM::go` keeps on using the last simulation cycle input vector.

SSM::UNWATCH

Command and description:

```
SSM::unwatch watch_list
```

Removes variables included in the `watch_list` list from traced variable set (`watch_list` can be a single variable). An error is generated when a variable is not previously traced.

SSM::UNWATCH_NODE

Command and description:

```
SSM::unwatch_node node [deep]
```

Removes the `node` variables from the traced variable list. If `deep` is mentioned and is equal to 1, then the watched variables of all the nodes called by `node` are unwatched. See `SSM::watch_node` for a usage example based on the complementary command.

SSM::WATCH

Command and description:

```
SSM::watch var_list
```

Adds SCADE `var_list` to the traced variable list. The value of a traced variable is displayed after each execution of `SSM::cycle`. The `var_list` list can be reduced to a unique SCADE variable name. The existence of the mentioned SCADE variables is not tested. An error message is however raised at each display attempt. An error is generated if `var_list` contains a SCADE Suite variable which is already traced.

Example: adds the SCADE variables a, b, c and d to the traced variable list.

```
[0]>SSM::watch [ list a b c d ]
```

SSM::WATCH_NODE

Command and description:

```
SSM::watch_node node [deep]
```

Adds the variables (inputs, outputs or locals) of the `node` SCADE node to the traced variable list. It is equivalent to a calling `SSM::watch` on each of these variables. Setting the `deep` parameter equal to 1 adds the variables of all the nodes called by `node`. Its default value is 0.

Example: Say A is a node with input `in_A`, output `out_A`, and calls node B. Say B is a node with input `in_B` and output `out_B`. Both the following command sequences are not executed during the same simulation session. The variables of A and B are viewed.

```
[0]> SSM::watch_node A
[0]> SSM::print_watches
in_A
out_A
Only the variables of node A are viewed.
[0]> SSM::watch_node A 1
[0]> SSM::print_watches
in_A
out_A
in_B
out_B
```

SSM::WHAT

Command and description:

```
SSM::what var_name
```

Returns the kind of the observable variable `var_name` as a string. Possible values are: `Root Input`, `Root Output`, `Input`, `Output`, `Probe`, `Sensor`, `Signal` or `Local`.

Root in the context of the present manual stands for “simulation root operator”, namely the node selected before starting simulation.

Simulation-Specific Variables

Variables dedicated to simulation are available for specific context of usage:

Table 5.2: Simulation-specific variables and usage context

Variable name	SSS scenarios	Interactive command line	Breakpoint expression
SSM::default_int	x	x	x
SSM::default_bool	x	x	x

Table 5.2: Simulation-specific variables and usage context (Continued)

Variable name	SSS scenarios	Interactive command line	Breakpoint expression
SSM::default_real	x	x	x
SSM::default_char	x	x	x
SSM::watches		x	

Commands and description:

`SSM::default_int`

Default value of integers. It is used when default vectors are generated.

`SSM::default_bool`

Default value of Boolean variables. It is used when default vectors are generated.

`SSM::default_real`

Default value of real variables. It is used when default vectors are generated.

`SSM::default_char`

Default value of characters. It is used when default vectors are generated.

`SSM::watches`

List of SCADE traced variables. The values of the variables mentioned in that list are displayed after each `SSM::cycle` call. This command can be set by `SSM::setSimVar` and retrieved by `SSM::getSimVar`.

Simulator Tcl Extension

SCADE Suite Simulation supports a Tcl command interpreter which allows to mix simulation commands and Tcl commands[1].

1. For details about Tcl commands, consult online Tcl documentation.

Conversion and Comparison Functions for Types

SCADE Suite allows for customizing conversion/comparison functions used at simulation run-time.

- *Conversion functions* are necessary for displaying observable variables (outputs, probes, locals) in Simulation (Instances view, diagrams, Watch window, and Graph window), for producing simulation output files (`.out`, `.in`, `.csv`, `.sns`, and `.sss`), and for parsing input files (`.in`, `.sns`, and `.sss`).
- *Comparison functions* are necessary for evaluating breakpoints and for checking expected values against simulation values.

Conversion and comparison functions must be referenced in a **VTable** structure.

- "Basic Use Cases"
- "Structure of VTable for C Code"
- "Structure of VTable for Ada Code"
- "Definition of SimConvType (C only)"
- "Defining VTables in C Source File"
- "Defining VTables in Ada Source File"
- "Examples of Function Definitions"

Basic Use Cases

Simulation functions are needed in the following cases:

FORMAT OF TYPE DISPLAY OR DUMPS

SCADE Suite Simulation interface displays values as strings. The default format used by SCADE Suite to display values in the user interface or dump values in output files can be changed to user-defined format with the help of conversion/comparison functions. By default, all real numbers are displayed with the "`%.5g`" format in simulation:

`0.123456789` displayed as `0.12346`
`0.00000123456789` displayed as `1.2346e-006`
`123456789` displayed as `1.2346e+008`

The default display (not hard-coded) is set in the Simulation Properties of the project under **Real numbers display** options. If unsatisfied with the default, users can provide functions to customize values displayed in user interface and dumped to output files. For example:

- "`%.17g`" to increase significant digits
- "`%.5f`" to force decimal display
- "`%.5e`" to force scientific display
- "%X" to display integers in hexadecimal

USER DEFINITION OF KCG TYPES AND MACROS

Conversion and comparison functions are mandatory for processing types whose redefinition is not natively supported in SCADE Suite. For example: `kcg_real` redefined into another C type than float and double (fixed point, long double). For guidelines, see "Simulating Models with User C-Macros" on page 678 in *SCADE Suite User Manual*. For details about macro and type redefinition see "Implementing C Macro User Definitions" on page 670 in *SCADE Suite User Manual*.

IMPORTED TYPES

Conversion and comparison functions are mandatory for simulating models with imported types in SCADE Suite. For guidelines, see "Simulating Models with Imported C Types" on page 669 when using C types or "Simulating Models with External Ada Code" on page 679 when using Ada types in *SCADE Suite User Manual*. In the context of snapshot files, see Appendix E about "Test Services API" in *SCADE Test User Manual*.

Structure of VTable for C Code

A VTable structure for C includes:

- One versioning attribute (allows managing previous VTable versions)
- One function defining which conversions are implemented by the VTable
- One function dumping a value to a specified type
- One function compiling a value from a specified type
- One function comparing two values
- One function comparing two values with tolerance (only for real types like kcg_real, kcg_float32, and kcg_float64)

The C declaration of the VTable structure is defined in `SmuVTable.h` as follows:

```
struct SimTypeVTable {
    /*
    * version of the table
    */
    SimConvVersion m_version;
    /*
    * non zero if both nTo and nFrom are provided
    * (SptNone is considered as beeing always provided)
    */
    int (*m_pfnGetConvInfo)(SimConvType nTo, SimConvType nFrom);
    /*
    * serializes a value
    * returns the address of the conversion result (non NULL if successful)
    */
    const void* (*m_pfnToType)(SimConvType, const void* pT);
    /*
    * compiles a value (non zero if successful)
    */
    int (*m_pfnFromType)(SimConvType, const void* pFrom, void* pT);
    /*
    * compares two values (non zero if successful)
    */
    int (*m_pfnCompare)(int* result, const void* pValue1, const void* pValue2);
    /*
    * compares two values with tolerance (non zero if successful)
    */
    int (*m_pfnCompareWithTol)(int* result, const void* pValue1, const void* pValue2, const
void* pTol);
    /*
    * release a value (non zero if successful)
    */
    int (*m_pfnRelease)(void* pValue);
};
```

Structure of VTable for Ada Code

A VTable structure for Ada includes:

- One versioning attribute
- One function defining which conversions are implemented by the VTable
- One function dumping a value to a specified type
- One function compiling a value from a specified type
- One function comparing two values with or without tolerance (only for kcg_float32 and kcg_float64)

The Ada declaration of the VTable structure is defined in `smu_vtable.ads` as follows:

```
    type Sim_Conv_Version is (Scv_R17, Scv_R18);

    type Sim_Conv_Type is (Spt_None, Spt_String, Spt_Long, Spt_Double);

    type PFN_VTABLE_GET_CONV_INFO is access function (To_Type: Sim_Conv_Type; From_Type:
Sim_Conv_Type) return C.int;
    type PFN_VTABLE_TO_TYPE is access function (To_Type: Sim_Conv_Type; From_Addr:
System.Address; To_Addr: System.Address) return C.int;
    type PFN_VTABLE_FROM_TYPE is access function (From_Type: Sim_Conv_Type; From_Addr:
System.Address; To_Addr: System.Address) return C.int;
    type PFN_VTABLE_COMPARE is access function (Value1_Addr1: System.Address; Value2_Addr:
System.Address; Tol: access Sim_Tolerance) return C.unsigned;

    type Sim_Type_VTable is
        record
            -- version of the table
            Version: Sim_Conv_Version;
            -- non zero if both To_Type and From_Type are provided
            Get_Conv_Info_Func: PFN_VTABLE_GET_CONV_INFO;
         -- serializes a value, updates the object pointed by To_Addr (non NULL if successful)
            To_Type_Func: PFN_VTABLE_TO_TYPE;
            -- compiles a value (non zero if successful)
            From_Type_Func: PFN_VTABLE_FROM_TYPE;
            -- compares two values (non zero if successful)
            Compare_Func: PFN_VTABLE_COMPARE;
        end record;
```

Definition of SimConvType (C only)

Conversion functions convert variables from KCG context from/to one or more predefined types defined by **SimConvType** in `SmuVTable.h`. The definition of **SimConvType** is as follows:

```
typedef enum {
    SptNone,
    SptString,
    SptChar,
    SptShort,
    SptLong,
    SptFloat,
    SptDouble,
    SptBool,
    _SptLast/* bounds checking */
} SimConvType;

typedef enum {
    SbFalse,
    SbTrue
} SimBool;
```

Note

Values from basic or imported types can be displayed in the Simulator Graph window only if convertible to numeric types (*i.e.,* SptShort, SptLong, SptFloat, SptDouble).

All values are passed/returned by address as defined in the following table:

SptNone	
SptString	char**
SptChar	char*
SptShort	short*
SptLong	long*
SptFloat	float*
SptDouble	double*
SptBool	
SbFalse	n/a (only for compatibility with VTable versions before Scv65)
SbTrue	

Defining VTables in C Source File

For C code generation, the definition of VTables with conversion/comparison functions must be contained in a C source file.

Note

To declare C source files in projects, see "Simulating Models with Imported C Types" on page 669 and "Simulating Models with User C-Macros" on page 678.

The definition of VTables requires to take care of the following steps:

- Include of `SmuVTable.h` in C source file
- Custom definition of needed conversion/comparison functions
- Declaration of VTable content
- Link between type and VTable

The following sections provide some guidelines for writing VTables.

DEFINING CONVERSION CAPABILITY OF VTABLE

The type conversions made possible by the VTable are defined as follows:

```
int <T>GetConvInfo(SimConvType nTo, SimConvType nFrom)
```

This function returns 1 when the VTable implements conversion from `nFrom` to `nTo`, if not it returns 0.

Usage of function:

Define how the type can be converted. This function is called before each conversion (<T>FromType or <T>ToType) to ensure VTable capability to convert from or to a type.

where <T> is the shortname of the type

DEFINING CONVERSION FUNCTIONS OF VTABLE

The conversion functions available for VTables of type <T> are defined as follows:

```
const void* <T>ToType(SimConvType nTo, const void* pT)
```

This function implements the conversion of the object pointed by `pT` of type <T> into a new object whose type depends on `nTo`. For instance, if `nTo` is of type `SptString` or `SptDouble`, the returned type is respectively `char**` or `double*`. The function returns the address of the new object. In case of invalid conversion function, it returns 0.

Usage of function:

In Simulation:

- Display of observable variables in **Instances View**, in **Network Simulation Views**, and in **Graph** and **Watch** windows
- Production of simulation values in supported file formats (.csv, .sss, .sns, or .out)

In Model Coverage:

- Production of execution results in execution result files (.out)

In Testing Environment:

- Production of test results in raw result files (_RAW.txt)

```
int <T>FromType(SimConvType nFrom, const void* pFrom, void* pT)
```

This function implements the conversion of the object pointed by `pFrom` of type identified by `nFrom`. For instance, if `nFrom` is of type `SptString`, the actual type of `pFrom` is `char**`. The resulting value is given to the object pointed by `pT`. If the function is supported it returns 1, if not it returns 0.

Usage of function:

For Simulation:

- Setting input simulation values in **Instances View** and in **Watch** window
- Parsing simulation values from simulation files (.in, .sns, or .sss)

For Model Coverage and Test Environment:

- Parsing test case values from simulation files (.sss, .sns, or .in in Model Coverage; .sss or.csv in Test Environment)

DEFINING COMPARISON FUNCTIONS OF VTABLE

The comparison functions available for VTables of type <T> are defined as follows:

```
int <T>Compare(int* result, const void* pValue1, const void* pValue2)
```

This function compares two values of type <T> pointed to by `pValue1` and `pValue2`. It also updates the content of the result variable as a bitwise combination of the following flags:

- SIM_CMP_RES_EQ /*a==b */
- SIM_CMP_RES_NEQ /*a!=b */
- SIM_CMP_RES_LT /*a<b*/
- SIM_CMP_RES_GT /*a>b*/
- SIM_CMP_RES_ERR /*misc error*/

The function returns 0 in case of equality, 1 if `pValue2>pValue1`, else it returns -1. It implements the evaluation of the exact comparison criteria specified by Test Environment for Host.

Usage of function:

For Simulation:

- Breakpoint evaluation
- Check of expected values versus simulation values (use of `SSM::check` in .sss simulation input files)

For Model Coverage and Test Environment:

- Check of expected values versus test case values (use of `SSM::check` in .sss simulation input files or conformance test in .csv), for data of type different from real

```
int <T>CompareWithTol(int* result, const void* pValue1, const void* pValue2, const void* pTol)
```

This function compares the value of type <T> pointed to by `pValue1` and `pValue2` with respect to the absolute/relative tolerance specified by `pTol`. The object pointed by `pTol` is of type `SimTolerance` defined in `SmuVTable.h`. It also updates the content of the `result` parameter as a bitwise combination of the following flags:

- SIM_CMP_RES_EQ /*a==b or |a-b| <= tol (if tol defined)*/
- SIM_CMP_RES_NEQ /*a!=b or |a-b| > tol (if tol defined)*/
- SIM_CMP_RES_LT /*a<b*/
- SIM_CMP_RES_GT /*a>b*/
- SIM_CMP_RES_ERR /*misc error*/
- SIM_CMP_RES_BOUNDARY /* if near boundary warning is raised*/

It implements the evaluation of the comparison criteria with tolerance.

Usage of function:

For Simulation, Model Coverage, and Test Environment:

- Check of values of type `real` (use of `SSM::set_tolerance` in .sss simulation input files or use of `SSM::check` with tolerance parameter)

Note

The comparison functions implement the evaluation of test criterion.

DEFINING MEMORY MANAGEMENT FUNCTION OF VTABLE

The memory management made possible by the VTable is defined as follows:

```
int <T>Release (void* pValue)
```

This function is dedicated to free the memory allocated by the initialization function of an imported type. This allows supporting imported types as pointers. It returns 1 if successful, otherwise it returns 0.

Usage of function:

The function is typically called for `SSM::check` expected values when the check duration expires.

where <T> is the shortname of the type

DECLARING VTABLE CONTENT

The declaration of the VTable and functions for type <T> has the following format:

```
struct SimTypeVTable sim<T>VTable = {
    ScvR18,
    <T>GetConvInfo,
    <T>ToType,
    <T>FromType,
    <T>Compare,
    <T>CompareWithTol,
    <T>Release,
};
```

where **Scv65** identifies the VTable version

SETTING LINK BETWEEN VTABLE AND TYPE

The link between the VTable and type is a pointer to the VTable used for simulation. The link must respect the following format:

```
struct SimTypeVTable* pSim<Label>VTable = &sim<T>VTable;
```

Depending on the kind of type <T>, **<Label>** can take the following values:

If <T> is:	<Label> is:
kcg_real	Double
kcg_int	Long
kcg_bool	Bool
kcg_char	Char
kcg_intN	IntN
kcg_uintN	UintN
kcg_floatN	FloatN

For imported types and structured types (array, structure, enumeration), **<Label>** must be the name of the C type generated by KCG for <T>.

Defining VTables in Ada Source File

For Ada code generation, the definition of VTables with conversion/ comparison functions must be contained in an Ada source file.

Note

To declare Ada source files in projects, see "Simulating Models with External Ada Code" on page 679.

The definition of VTables requires to take care of the following steps:

- Include of "`smu_vtable.ads`" in Ada source file
- Custom definition of needed conversion/comparison functions
- Declaration of VTable content
- Link between type and VTable

The following sections provide some guidelines for writing VTables.

DEFINING CONVERSION CAPABILITY OF VTABLE

The type conversions made possible by the VTable are defined as follows:

```
function <T>_Get_Conv_Info(To_Type: Sim_Conv_Type; From_Type: Sim_Conv_Type) return C.int
```

This function returns 1 when the VTable implements conversion from `From_Type` to `To_Type`, if not it returns 0.

Usage of function:

Define how the type can be converted. This function is called before each conversion (`Type_To_<T>` or `<T>_To_Type`) to ensure VTable capability to convert from or to a type.

where <T> is the shortname of the type

DEFINING CONVERSION FUNCTIONS OF VTABLE

The conversion functions available for VTables of type <T> are defined as follows:

```
function <T>_To_Type (To_Type: Sim_Conv_Type; From_Addr: System.Address; To_Addr: System.Address) return
C.int
```

This function implements the conversion of the object pointed by `From_Addr` of type <T> into a new object whose type depends on `To_Type`. The resulting value is given to the object pointed by `To_Addr`. In case of invalid conversion function, it returns 0.

Usage of function:

In Simulation:

- Display of observable variables in **Instances View**, in **Network Simulation Views**, and in **Graph** and **Watch** windows
- Production of simulation values in supported file formats (.csv, .sss, .sns, or .out)

In Model Coverage:

- Production of execution results in execution result files (.out)

In Test Environment:

- Production of test results in raw result files (_RAW.txt)

```
function Type_To_<T>(From_Type: Sim_Conv_Type; From_Addr: System.Address; To_Addr: System.Address) return
C.int
```

This function implements the conversion of the object pointed by `From_Addr` of type identified by `To_Type`. The resulting value is given to the object pointed by `To_Addr`. In case of invalid conversion function, it returns 0.

Usage of function:

For Simulation:

- Setting input simulation values in **Instances View** and in **Watch** window
- Parsing simulation values from simulation input files (.in, .sns, or .sss)

For Model Coverage and Test Environment:

- Parsing test case values from simulation input files (.sss, .sns, or .in in Model Coverage; .sss or.csv in Test Environment)

DEFINING COMPARISON FUNCTIONS OF VTABLE

The comparison functions available for VTables of type <T> are defined as follows:

```
function <T>_Compare(Value1_Addr: System.Address; Value2_Addr: System.Address; Tol: access
Smu_Types.Sim_Tolerance) return C.unsigned
```

This function compares two values of type <T> pointed to by `Value1_Addr` and `Value2_Addr` with respect to the absolute/relative tolerance specified by `Tol`. The object pointed by `Tol` is of type `Sim_Tolerance` defined in `smu_vtable.ads`. It also returns the result variable as a bitwise combination of the following flags:

- SIM_CMP_RES_EQ /*a==b */
- SIM_CMP_RES_NEQ /*a!=b */
- SIM_CMP_RES_LT /*a<b*/
- SIM_CMP_RES_GT /*a>b*/
- SIM_CMP_RES_ERR /*misc error*/

The function implements the evaluation of the exact comparison criteria specified by TEE.

Usage of function:

For Simulation:

- Breakpoint evaluation
- Check of expected values versus simulation values (use of `SSM::check` in .sss simulation input files)

For Model Coverage and Test Environment:

- Check of expected values versus test case values (use of `SSM::check` in .sss simulation input files or conformance test in .csv), for data of type different from real

---

Note

The comparison functions implement the evaluation of test criterion.

DECLARING VTABLE CONTENT

The declaration of the VTable and functions for type <T> has the following format:

```
<T>_Vtable: aliased Sim_Type_VTable := (
Scv_R17,
<T>_Get_Conv_Info'Access,
<T>_To_String'Access,
String_To_<T>'Access,
<T>_Compare'Access
);
```

where **Scv_R17** identifies the VTable version

SETTING LINK BETWEEN VTABLE AND TYPE

The link between the VTable and type is a pointer to the VTable used for Ada simulation. The link must respect the following format:

```
<T>_VTable: access Smu_VTable.Sim_Type_VTable := <T>_VTable'Access; for <T>_VTable'Address
use Smuw_Type.Smuw_<TypeQualifiedPath>.VTable'Address;
```

<T> is a shortname for the type and <TypeQualifiedPath> is the Ada qualified path to the type

Examples of Function Definitions

When implementing new functions for types, you can look at the following examples and consult for reference the `libimpl` library implementation from its `libimpl.c` file located in <SCADE installation folder>\SCADE\libraries\libimpl.

Example 1: The kcg_real type is defined as follows in an H-file

```
/*****************************************************************//**
* @file user_macros.h
* @brief Header file with type definitions and macros.
*
*********************************************************************/

#ifndef USER_MACROS_H
#define USER_MACROS_H

#define kcg_real kcg_real
typedef float kcg_real;
```

For displaying observable variables and producing simulation output values using this type, the appropriate definition of the `ToType` conversion function would be required. Since the type definition above is a supported custom definition (see Table 14.1), the conversion function is not necessary. Simulation display and output of reals can be managed by setting significant digits as explained in "Setting Project for Simulation Sessions" on page 581.

Example 2: The Pair type is an imported structure type defined as follows in an H-file

```
#ifndef PAIR_H
#define PAIR_H
typedef struct{
  int C1;
  int C2;
} Pair_Package ;
#define Pair_init_Package(pTo) {\
  (pTo)->C1=0; \
  (pTo)->C2=0; \
}
#define kcg_copy_Pair_Package(pTo,pFrom) {\
  (pTo)->C1 = (pFrom)->C1; \
  (pTo)->C2 = (pFrom)->C2; \
}
#define kcg_comp_Pair_Package(x, y)   ( ( (x)->C1==(y)->C1 &&  (x)->C2==(y)-
>C2)?kcg_true:kcg_false)
#endif
```

The conversion/comparison functions are defined as follows in a C-file:

`#include <stdio.h>`
`#include "SmuVTable.h"`
`#include "kcg_imported_types.h"`
`...`
	Header

`/* Pair_Package */`
 	
`static int PairGetConvInfo(SimConvType nTo, SimConvType nFrom){`
`  int bTo = nTo == SptString || nTo == SptNone;`
`  int bFrom = nFrom == SptString || nFrom == SptNone;`
`  return bTo && bFrom;`
`}`
	Definition of VTable conversion capability

`static const void* PairToType(SimConvType nTo, const void* pT)`
`{`
`  switch (nTo) {`
`  case SptString:`
`  {`
`  static char szValue[32];`
`  static char* pszResult = szValue;`
 	
`  Pair_Package *pair=(Pair_Package*) pT;`
`  sprintf(szValue,"(%d,%d)", pair->C1, pair->C2);`
`  return &pszResult;`
`  }`
`  break;`
`  default:`
`  return NULL;`
`  }`
`  return NULL;`
`}`
	Definition of conversion function from Pair type to another type

static int PairFromType(SimConvType nFrom, const void* pType,
void* pT){
    switch (nFrom) {
    case SptString:
    {
        char* p = (*(char**) pType);
 	
        Pair_Package *pair=(Pair_Package*) pT;
        int tmp1;
        int tmp2;
        sscanf(p, "(%d,%d)", &tmp1, &tmp2);
        pair->C1 = tmp1;
        pair->C2 = tmp2;
    }
    break;
    default:
    return 0;
    }
/* value successfully compiled */
    return 1;
}
...
	Definition of conversion function from another type to Pair type

`static int PairCompare(int* result, const void* pValue1, const void* pValue2)`
`{`
`  int nRet=0;`
`  static Pair_Package nValue1, nValue2;`
`  nValue1 = *(const Pair_Package*) pValue1;`
`  nValue2 = *(const Pair_Package*) pValue2;`
`  if (nValue1.C1 < nValue2.C1) {`
`  *result |= SIM_CMP_RES_NEQ;`
`  *result |= SIM_CMP_RES_LT;`
`  nRet = -1;`
`  } else if (nValue1.C1 > nValue2.C1) {`
`  *result |= SIM_CMP_RES_NEQ;`
`  *result |= SIM_CMP_RES_GT;`
`  nRet = 1;`
`  } else { /* (nValue1.C1 == nValue2.C1) */`
`  if (nValue1.C2 < nValue2.C2) {`
`  *result |= SIM_CMP_RES_NEQ;`
`  *result |= SIM_CMP_RES_LT;`
`  nRet = -1;`
`  } else if (nValue1.C2 > nValue2.C2) {`
`  *result |= SIM_CMP_RES_NEQ;`
`  *result |= SIM_CMP_RES_GT;`
`  nRet = 1;`
`  } else { /* (nValue1.C2 == nValue2.C2) */`
`  *result |= SIM_CMP_RES_EQ;`
`  nRet = 0;`
`  }`
`  }`
`  return 1;`
`}`
	Definition of comparison function for variables of Pair type

`static int PairRelease(void* pValue)`
`{`
`  Pair_Package *pair=(Pair_Package *)pValue;`
`  /* insert here code for freeing memory */`
`  return 1;`
`}`
	Definition of release function for Pair type

`static struct SimTypeVTable simPairVTable = {`
`  ScvR18,`
`  PairGetConvInfo,`
`  PairToType,`
`  PairFromType,`
`  PairCompare,`
`  NULL,/*no comparison with tolerance*/`
`  PairRelease`
`};`
	Declaration of VTable content for Pair type

`struct SimTypeVTable* pSimPair_PackageVTable = &simPairVTable;`	Declaration pointer of VTable content for Pair type (used for Simulation)

Example 3: VTable specification (.ads)

```
with System;
with Interfaces, Interfaces.C, Interfaces.C.Strings; use Interfaces;
with Smu_VTable;
with Smu_Types;

package User_VTable is

    function tImported_Get_Conv_Info(To_Type: Smu_VTable.Sim_Conv_Type; From_Type:
Smu_VTable.Sim_Conv_Type) return C.int;
    pragma Convention (C, tImported_Get_Conv_Info);

    function tImported_To_String (To_Type: Smu_VTable.Sim_Conv_Type; From_Addr:
System.Address; To_Addr: System.Address)
    return C.int;
    pragma Convention (C, tImported_To_String);

    function String_To_tImported(From_Type: Smu_VTable.Sim_Conv_Type; From_Addr:
System.Address; To_Addr: System.Address)
    return C.int;
    pragma Convention (C, String_To_tImported);

   function tImported_Compare(Value1_Addr: System.Address; Value2_Addr: System.Address; Tol:
access
    Smu_Types.Sim_Tolerance) return C.unsigned;
    pragma Convention (C, tImported_Compare);

end User_VTable;
```

Conversion/Comparison functions in VTable for Ada:

`with System;`
`with Interfaces, Interfaces.C, Interfaces.C.Strings; use Interfaces;`
`with Ada.Strings.Fixed;`
 	
`with P1, P1.Kcg_Imported, P2.Kcg_Imported;`
 	
`with Smu_Types, Smu_VTable;`
`with Smuw_Type;`
 	
`use type Interfaces.C.int;`
`use type Interfaces.C.unsigned;`
`use type P1.Kcg_Imported.tImported;`
 	
`package body User_VTable is`
	Header

`function tImported_Get_Conv_Info(To_Type: Smu_VTable.Sim_Conv_Type;`
`From_Type: Smu_VTable.Sim_Conv_Type) return C.int is`
`  use type Smu_VTable.Sim_Conv_Type;`
`  T: Boolean := To_Type = Smu_VTable.Spt_String`
`  or To_Type = Smu_VTable.Spt_None;`
`  F: Boolean := From_Type = Smu_VTable.Spt_String`
`  or From_Type = Smu_VTable.Spt_None;`
`begin`
`  if T and F then`
`  return 1;`
`  end if;`
`  return 0;`
`end User_tImported_Get_Conv_Info;`
	Definition of VTable conversion capability

`function tImported_To_String (To_Type: Smu_VTable.Sim_Conv_Type;`
`From_Addr: System.Address; To_Addr: System.Address) return C.int is`
`  Value: P1.Kcg_Imported.tImported; for Value'Address use From_Addr;`
`begin`
`  case To_Type is`
`  when Smu_VTable.Spt_String =>`
`  Tostr: declare`
`  Res: C.Strings.chars_ptr; for Res'Address use To_Addr;`
`  begin`
`  Res := C.Strings.New_String(Ada.Strings.Fixed.Trim`
`  (P1.Kcg_Imported.tImported'Image(Value),`
`Ada.Strings.Left));`
`  return 1;`
`  end Tostr;`
`  when others =>`
`  return 0;`
`  end case;`
`end tImported_To_String;`
	Definition of conversion function from tImported type to another type

`function String_To_tImported(From_Type: Smu_VTable.Sim_Conv_Type;`
`From_Addr: System.Address; To_Addr: System.Address) return C.int is`
`  Value: P1.Kcg_Imported.tImported; for Value'Address use To_Addr;`
`begin`
`  case From_Type is`
`  when Smu_VTable.Spt_String =>`
`  Strto: declare`
`  Str: C.Strings.chars_ptr := Smu_Types.to_chars_ptr(From_Addr);`
`  Buf: String := C.Strings.Value(Str);`
 	
`  begin`
`  Value := P1.Kcg_Imported.tImported'Value(Buf);`
`  return 1;`
`  end Strto;`
`  when others =>`
`  return 0;`
`  end case;`
`end String_To_tImported;`
	Definition of conversion function from another type to tImported type

function tImported_Compare(Value1_Addr: System.Address; Value2_Addr: System.Address; Tol: access Smu_Types.Sim_Tolerance) return C.unsigned is
    Value1: P1.Kcg_Imported.tImported; for Value1'Address use Value1_Addr;
    Value2: P1.Kcg_Imported.tImported; for Value2'Address use Value2_Addr;
    Res: aliased C.unsigned := 0;
begin
    if (Value1 = Value2) then
            Res := Smu_Types.SIM_CMP_RES_EQ;
    elsif (Value1 < Value2) then
          Res := Smu_Types.SIM_CMP_RES_NEQ or Smu_Types.SIM_CMP_RES_LT;
    else
        Res := Smu_Types.SIM_CMP_RES_NEQ or Smu_Types.SIM_CMP_RES_GT;
    end if;
    return Res;
end tImported_Compare;
	Definition of comparison function for variables of tImported type

tImported_VTable: aliased Smu_VTable.Sim_Type_VTable := (
    Smu_VTable.Scv_R17,
    tImported_Get_Conv_Info'Access,
    tImported_To_String'Access,
    String_To_tImported'Access,
    tImported_Compare'Access
);
	Declaration of VTable content for tImported type

tImported_VTable_A: access Smu_VTable.Sim_Type_VTable := tImported_VTable'Access; for tImported_VTable_A'Address use Smuw_Type.Smuw_P1.Kcg_Imported.tImported.VTable'Address; end User_VTable;	Declaration pointer of VTable content for tImported type (used for Simulation)

Conversion Functions for Imported Operators

Conversion functions are useful for saving and loading imported operator context structure as text into snapshot files (SNS format). They apply to SCADE Suite Simulation, SCADE Test Model Coverage, and SCADE Test Environment.

Such functions are not mandatory. By default, SCADE Suite Simulation saves all data in hexadecimal format. In case, the snapshot file must be exchanged between heterogeneous platforms, it is required to define the conversion functions and ensure portability.

t is possible to provide additional files implementing functions for conversion from a context variable to a string and from a string to a context variable.

- "Structure of VTable for C Code"
- "Structure of VTable for Ada Code"
- "Defining VTables in C Source File"
- "Defining VTables in Ada Source File"
- "Examples of Function Definitions"

Structure of VTable for C Code

A VTable structure for C includes:

- One versioning attribute (allows managing previous VTable versions)
- One function dumping an imported operator context to a string
- One function compiling an imported operator context from a string

The C declaration of the VTable structure is defined in `SmuVTable.h` as follows:

```
typedef int (*PFN_IOCVTABLE_TO_STRING)(const void* pOutCtx, PFN_STR_APPEND pfnStrAppend,
void* pStrObj);
typedef int (*PFN_IOCVTABLE_FROM_STRING)(const char* szString, void* pOutCtx, char** endptr);

struct SimImportedOpCtxVTable {
    /*
    * version of the table
    */
    SimConvVersion m_version;

    /*
    * serializes the output context of an imported operator to a string
    * returns non zero if successful
    */
    PFN_IOCVTABLE_TO_STRING m_pfnToString;

    /*
    * compiles the output context of an imported operator from a string
    * returns non zero if successful
    */
    PFN_IOCVTABLE_FROM_STRING m_pfnFromString;
};
```

Structure of VTable for Ada Code

A VTable structure for Ada includes:

- One versioning attribute
- One function dumping an imported operator context to a string
- One function compiling an imported operator context from a string

The Ada declaration of the VTable structure is defined in `smu_vtable.ads` as follows:

```
type PFN_IOCVTABLE_TO_STRING is access function (Out_Ctx_Addr: System.Address;
Str_Append_Func: PFN_STR_APPEND; Str_Addr: System.Address) return C.int;
type PFN_IOCVTABLE_FROM_STRING is access function (C_Str: C.Strings.chars_ptr; Out_Ctx_Addr:
System.Address; End_Ptr: access System.Address) return C.int;

type Sim_ImportedOpCtx_VTable is
    record
        -- version of the table
        Version: Sim_Conv_Version;
       -- serializes the output context of an imported operator to a string, returns non zero
if successful
        To_String_Func: PFN_IOCVTABLE_TO_STRING;
      -- compiles the output context of an imported operator from a string, returns non zero
if successful
        From_String_Func: PFN_IOCVTABLE_FROM_STRING;
    end record;
```

Defining VTables in C Source File

For C code generation, the definition of VTables with conversion functions must be contained in a C source file.

Note

To declare C source files in projects, see "Simulating Models with External C Code" on page 667.

The definition of VTables requires to take care of the following steps:

- Include of `SmuVTable.h` in C source file

- Custom definition of conversion functions
- Declaration of VTable content
- Link between imported operator context and VTable

The following sections provide some guidelines for writing VTables.

DEFINING CONVERSION FUNCTIONS OF VTABLE

The conversion functions available for VTables of imported operator <IOp> are defined as follows:

```
int <IOp>ToString(const void* pOutCtx, PFN_STR_APPEND pfnStrAppend, void* pStrObj)
```

This function implements the conversion of the context structure pointed by pOutCtx of type <ImpOpCtx> into a string pointed by pStrObj using the provided function pfnStrAppend. If the function succeeds it returns a non null value; in case of error it returns 0. <ImpOpCtx> is the full C path to the imported operator context structure

```
int <IOp>FromString(const char* szString, void* pOutCtx, char** endptr)
```

This function implements the conversion of the string pointed by szString into a context structure pointed by pOutCtx of type <ImpOpCtx>. The function must set the endptr output parameter to the next character after the parsed value. If the function succeeds it returns a non null value; in case of error it returns 0.

DECLARING VTABLE CONTENT

The declaration of the VTable and functions for imported operator <IOp> has the following format:

```
struct Sim<IOp>VTable sim<IOp>VTable = {
    ScvR17,
    <IOp>ToString,
    <IOp>FromString
};
```

ScvR17 identifies the VTable version

SETTING LINK BETWEEN VTABLE AND IMPORTED OPERATOR CONTEXT

The link between the VTable and imported operator context is a pointer to the VTable used for simulation. The link must respect the following format:

```
struct SimImportedOpCtxVTable* pSim<ImpOpCtx>IOCVTable = &sim<IOp>VTable;
```

`<ImpOpCtx>` is the full C path to the imported operator context structure

Defining VTables in Ada Source File

For Ada code generation, the definition of VTables with conversion functions must be contained in an Ada source file.

Note

To declare Ada source files in projects, see "Simulating Models with External Ada Code" on page 679.

The definition of VTables requires to take care of the following steps:

- Include of `smu_vtable.ads` in Ada source file
- Custom definition of conversion functions
- Declaration of VTable content
- Link between type and VTable

The following sections provide some guidelines for writing VTables.

DEFINING CONVERSION FUNCTIONS OF VTABLE

The conversion functions available for VTables of imported operator <IOp> are defined as follows:

```
function <IOp>_To_String (Out_Ctx_Addr: System.Address; Str_Append_Func: Smu_Types.PFN_STR_APPEND;
Str_Addr: System.Address) return C.int
```

This function implements the conversion of the context record pointed by Out_Ctx_Addr of type <ImpOpCtx> into a string pointed by Str_Addr using the provided function Str_Append_Func. If the function succeeds it returns a non null value; in case of error it returns 0.

where <ImpOpCtx> is the Ada qualified path to the imported operator context record

```
function <IOp>_From_String (C_Str: C.Strings.chars_ptr; Out_Ctx_Addr: System.Address; End_Ptr: access
System.Address) return C.int
```

This function implements the conversion of the string pointed by C_Str into a context record pointed by Out_Ctx_Addr of type <ImpOpCtx>. The function must set the End_Ptr output parameter to the address of the next character after the parsed value. If the function succeeds it returns a non null value; in case of error it returns 0.

DECLARING VTABLE CONTENT

The declaration of the VTable and functions for imported operator <IOp> has the following format:

```
 <IOp>_VTable: aliased Smu_VTable.Sim_ImportedOpCtx_VTable := (
                                          Smu_VTable.Scv_R17,
                                          <IOp>_To_String'Access,
                                          <IOp>_From_String'Access);
```

where **Scv_R17** identifies the VTable version

SETTING LINK BETWEEN VTABLE AND TYPE

The link between the VTable and imported operator is a pointer to the VTable used for Ada simulation. The link must respect the following format:

```
<IOp>_VTable_Access: access Smu_VTable.Sim_ImportedOpCtx_VTable := <IOp>_VTable'Access;
for <IOp>_VTable_Access'Address use Smuw_Snapshot.Smuw_<ImpOpCtx>.VTable'Address;
```

where <IOp> is the shortname of the imported operator and <ImpOpCtx> is the Ada qualified path to the imported operator context record

Examples of Function Definitions

When implementing new functions for imported operators, you can look at the following examples.

Example 1: Conversion functions in VTable for C

`#include "ImportedNode_P2.h"`
`extern int kcg_float64_to_string(const void *pValue, PFN_STR_APPEND pfnStrAppend, void *pStrObj);`
`extern int string_to_kcg_float64(const char *str, void *pValue, char **endptr);`
	Header

`static int outC_ImportedNode_P2ToString(const void* pOutCtx, PFN_STR_APPEND pfnStrAppend, void* pStrObj)`
`{`
`  int bOK=1;`
`  outC_ImportedNode_P2* pC = (outC_ImportedNode_P2*)pOutCtx;`
`  bOK &= pfnStrAppend("(", pStrObj);\`
`  bOK &= kcg_float64_to_string(&pC->O_f64, pfnStrAppend, pStrObj);`
`  pfnStrAppend(", ",pStrObj);`
`  bOK &= kcg_float64_to_string(&pC->the_mem, pfnStrAppend, pStrObj);`
`  bOK &= pfnStrAppend(")", pStrObj);\`
`  return bOK;`
`}`
	Definition of conversion function of ImportedNode_P2 context structure into string

static int outC_ImportedNode_P2FromString(const char* szString,
void* pOutCtx, char** endptr)
{
  int bOK=1;
  outC_ImportedNode_P2* pC = (outC_ImportedNode_P2*)pOutCtx;
  endptr = (char)szString;\
  bOK &= *(*endptr) == '('; (*endptr)++;
  bOK &= string_to_kcg_float64(*endptr, &pC->O_f64, endptr);
  bOK &= (*(*endptr)==',');(*endptr)++;
  bOK &= (*(*endptr)==' ');(*endptr)++;
  bOK &= string_to_kcg_float64(*endptr, &pC->the_mem, endptr);
  bOK &= *(*endptr) == ')'; (*endptr)++;
  return bOK;
}
	Definition of conversion function of string into ImportedNode_P2 context structure

static struct SimImportedOpCtxVTable
simoutC_ImportedNode_P2IOCVTable = {
  ScvR17,
  outC_ImportedNode_P2ToString,
  outC_ImportedNode_P2FromString
};
	Declaration of VTable content for ImportedNode_P2 context structure

struct SimImportedOpCtxVTable* pSimoutC_ImportedNode_P2IOCVTable = &simoutC_ImportedNode_P2IOCVTable;	Declaration pointer of VTable content for ImportedNode_P2 context structure

Example 2: Conversion functions in VTable for Ada

with System; with Interfaces, Interfaces.C, Interfaces.C.Strings; use Interfaces; with ada.Strings.Unbounded, Ada.Strings; with Kcg_Config; use Kcg_Config; with P1, P1.Kcg_Imported, P2.Kcg_Imported; with Kcg_Top.Kcg_Sub; with Smu_Types, Smu_VTable, Smu_Predef_Conv; with Smuw_Type, Smuw_Snapshot, Smuw_Interface; use type Interfaces.C.int; package body User_VTable is	Header

Context_ImportedNode_Object: P2.Kcg_Imported.Context_ImportedNode; Context_ImportedNode_Fields: aliased Smu_Types.Array_Sim_Field_Utils(0..1):= ((Interfaces.C.Strings.New_String("O_f64"), Context_ImportedNode_Object.O_f64'Position, Smuw_Type.Smuw_Kcg_Config.Kcg_Float64.Type_Utils'Access), (Interfaces.C.Strings.New_String("the_mem"), Context_ImportedNode_Object.the_mem'Position, Smuw_Type.Smuw_Kcg_Config.Kcg_Float64.Type_Utils'Access)); function User_Context_ImportedNode_To_String (Out_Ctx_Addr: System.Address; Str_Append_Func: Smu_Types.PFN_STR_APPEND; Str_Addr: System.address) return C.int is begin return Smu_Types.Converter.Structure_To_String_Func(Out_Ctx_Addr, Context_ImportedNode_Fields(Context_ImportedNode_Fields'First)'Acce ss, Context_ImportedNode_Fields'Length, Str_Append_Func, Str_Addr); end User_Context_ImportedNode_To_String;	Definition of conversion function of context structure into string
function User_Context_ImportedNode_From_String (C_Str: C.Strings.chars_ptr; Out_Ctx_Addr: System.Address; End_Ptr: access System.Address) return C.int is begin return Smu_Types.Converter.String_To_Structure_Func(C_Str, Out_Ctx_Addr, Context_ImportedNode_Fields(Context_ImportedNode_Fields'First)'Acce ss, Context_ImportedNode_Fields'Length, End_Ptr); end User_Context_ImportedNode_From_String;	Definition of conversion function of string into context structure
Context_ImportedNode_VTable: aliased Smu_VTable.Sim_ImportedOpCtx_VTable := (Smu_VTable.Scv_R17, User_Context_ImportedNode_To_String'Access, User_Context_ImportedNode_From_String'Access);	Declaration of VTable content for context structure
User_Context_ImportedNode_VTable: access Smu_VTable.Sim_ImportedOpCtx_VTable := Context_ImportedNode_VTable'Access; for User_Context_ImportedNode_VTable'Address use Smuw_Snapshot.Smuw_P2.Kcg_Imported.Context_ImportedNode.VTable'Addr ess;	Declaration pointer of VTable content for context structure

Public API Functions for Co-Simulation Mode

The Simulation API is distributed with SCADE to enable the creation of an external application written in C and interacting with SCADE Simulator. This co-simulation mode can run most of the simulation activities you start from SCADE Simulator GUI. For user guidelines, consult "Using Simulation API for Co-Simulation" on page 681 in *SCADE Suite User Manual*.

Note

The API functions used for the remote control of SCADE Simulator are declared in the `SsmSlaveLib.h` file located in the `include` folder of SCADE installation.

- "Description of API Functions"
- "API Function Errors"
- "Example of Enriched Main Function"

Description of API Functions

The SCADE Simulation API contains functions that can call SCADE-generated code from the main function of an external application.

SsmOpenScadeSimulator
SsmCloseScadeSimulator
SsmGuiActivate
SsmGuiRefresh
SsmSetHostName
SsmSetConnectRetry
SsmSetScadeInstallPath
SsmSetInput
SsmSetInputVector
SsmStep
SsmDbgStep
SsmGetOutput
SsmGetOutputVector
SsmGetLastError

For details about the formats to be used in the definition of the function values and paths, see "Simulation Value Format" on page 156 and "Simulation Path Format" on page 159.

SsmOpenScadeSimulator API function

Function description and signature:

```
extern int SsmOpenScadeSimulator (const char* pszProjectPathName, const char*
pszConfName, const char* pszRootNode, int nSocketPort);
```

Simulation activity: Opening SCADE Simulator GUI given an absolute project/workspace path, a root operator, a configuration name, and a port number

- `pszRootNode` should be filled only if no root operator is set in the project settings, otherwise the default root operator defined in settings is taken.

SsmCloseScadeSimulator API function

Function description and signature:

```
extern int SsmCloseScadeSimulator();
```

Simulation activity: Closing SCADE Simulator GUI

SsmGuiActivate API function

Function description and signature:

```
extern int SsmGuiActivate();
```

Simulation activity: Viewing SCADE Simulator GUI as foreground window.

SsmGuiRefresh API function

Function description and signature:

```
extern int SsmGuiRefresh();
```

Simulation activity: Refreshing simulation values in SCADE GUI

SsmSetHostName API function

Function description and signature:

```
extern int SsmSetHostName (const char* pszHostName);
```

Simulation activity: Setting the hostname to connect with when calling SsmOpenScadeSimulator (default is 127.0.0.1)

SsmSetConnectRetry API function

Function description and signature:

```
extern int SsmSetConnectRetry (int nRetry);
```

Simulation activity: Setting the number of connection retry when calling SsmOpenScadeSimulator (default is 100).

SsmSetScadeInstallPath API function

Function description and signature:

```
extern int SsmSetScadeInstallPath(const char* pszPath);
```

Simulation activity: Setting SCADE binary path to override the PATH environment variable.

SsmSetInput API function

Function description and signature:

```
extern int SsmSetInput(char* pszPath, char* pszValue);
```

Simulation activity: Setting a specific value (pszValue) to input named pszPath.

SsmSetInputVector API function

Function description and signature:

```
extern int SsmSetInputVector(const char* pszInputVector);
```

Simulation activity: Sending input to SCADE Simulator as formatted vector.

SsmStep API function

Function description and signature:

```
extern int SsmStep(int nCycle, int bRefresh);
```

Simulation activity: Executing simulation step (nCycle) and returning output values ready to be retrieved at cycle completion.

- `nCycle`: number of steps to perform
- `bRefresh`: Set 1 to refresh values in SCADE GUI on each step (default mode), or set 0 to ignore value refresh on each step (the caller has the responsibility to call SsmGuiRefresh)

SsmDbgStep API function

Function description and signature:

```
extern int SsmDbgStep(int nCycle);
```

Simulation activity: Executing a simulation step and waiting until you press SCADE Simulator **Go** button.

SsmGetOutput API function

Function description and signature:

```
extern int SsmGetOutput(char* pszPath, const char** ppszValue);
```

Simulation activity: Retrieving the value of a specific output related to `pszPath`.

SsmGetOutputVector API function

Function description and signature:

```
extern int SsmGetOutputVector(const char** ppszOutputVector);
```

Simulation activity: Retrieving the value of the output vector computed by SCADE Simulator.

SsmGetLastError API function

Function description and signature:

```
extern int SsmGetLastError(const char** ppszError);
```

Simulation activity: Getting explicit error messages.

API Function Errors

All API functions return an integer value corresponding to an error code: 0 means the function call is successful. The main error codes are listed below.

Table 5.3: SCADE Simulator API error codes

Error code	Meaning
1 - 1000 Errors related to API gateway	
1	Cannot access project
2	Cannot connect to SCADE on port 6311
3	More than one instance of SCADE is running
4	Cannot connect to SCADE on requested port
5	Not connected to SCADE
6	Incorrect path when using the SsmGetOutput function
1001-2000 Errors related to proxy	
1001	Error during evaluation of a command
1002	Cannot Access simulator interface
1003	Simulator is not launched
1004	Cannot access simulator command
1005	Cannot access simulator data
1006	Cannot find variable
1007	Undefined error
1008	Cannot start simulator due to configuration or root operator
1009	SCADE co-simulation is already started

Example of Enriched Main Function

This section illustrates a modified main function necessary to run SCADE co-simulation using the Simulation API. This example of main function is a simple controller model controlling the inner light power of a car (see Figure 5.1). It is enriched with the following examples:

- In `TestCosimulationAPI.c`, the code calls the API functions provided by SCADE to support socket-based communication with SCADE. Users can use this example to write co-simulation that can run simulation step by step within SCADE simulation environment.
- In `TestCosimulation.c`, the code calls directly SCADE Suite KCG code (reset, cycle, etc). Users can use this example to write co-simulation that can run as standalone without using SCADE.

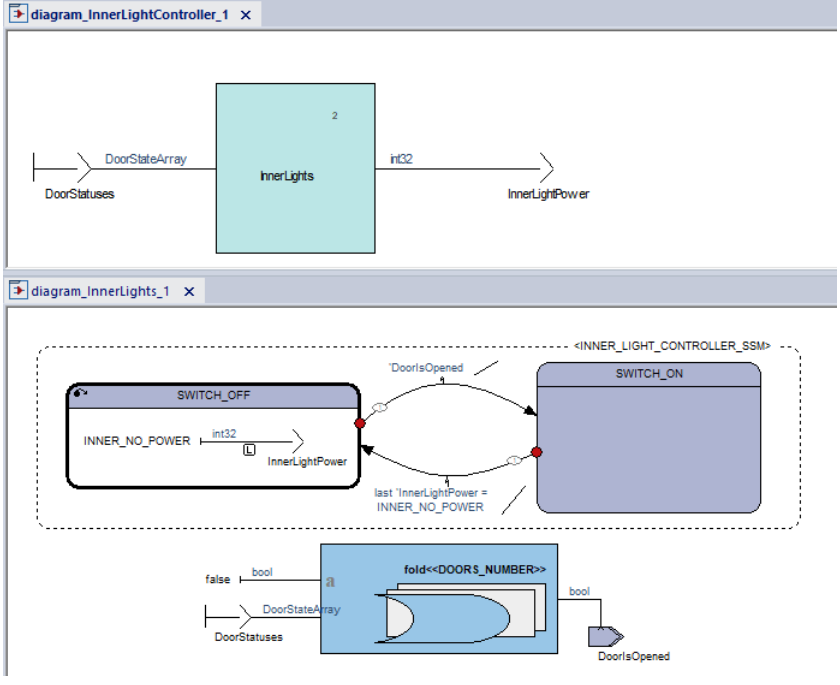


Figure 5.1: Model used for co-simulation from an external application

Example 1: TestCoSimulationAPI.c file with calls to API functions

```
// This file contains the 'main' function. Program execution begins and ends there.//

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include "BCM_MODEL/Simulation/kcg_consts.h"
#include "SsmSlaveLib.h"
#define INSTALL "C:/Program Files/ANSYS Inc/v231/SCADE/SCADE/bin"
#define PRJ "D:/ANSYSDev/ScadeStudies/TestCoSimulationAPI/BCM_MODEL/BCM_MODEL.etp"
#define CONF "Simulation"
#define NODE "InnerLightPkg::InnerLightController"
#define PORT 64064
#ifndef true
#define true 1
#define false 0
#endif

int main()
{
    int i;
    int nResult;
    const char* pszReturnValue = NULL;

    /* Point out SCADE instalation */
    SsmSetScadeInstallPath(INSTALL);
    /* Open SCADE Simulator, connect on port 64064 and start simulation of
    InnerLightController in "Simulation" configuration */
    nResult = SsmOpenScadeSimulator(PRJ, CONF, NODE, PORT);
    if (nResult) {
    SsmGetLastError(&pszReturnValue);
    printf("%s\n", pszReturnValue);
    return -1;
    }
    /* Make SCADE active window */
    SsmGuiActivate();
    /* Wait for user interaction */
    SsmDbgStep(1);
    /* Run the simulation for 31 steps */
    for (i = 0; i < 31; i++) {
    int j;
    char DoorStatusesInputBfr[100];
    //char string[1][100];
    const char* pszVector;

(...)
```

```
(continued)

/* Set inputs */
    DoorStatusesInputBfr[0] = '(';
    for (j = 0; j < DOORS_NUMBER_InnerLightPkg; j++) {
        kcg_bool doorsStatus = ((i < 5) && (j == 0)) ? kcg_true : kcg_false;
        DoorStatusesInputBfr[2 * j + 1] = (doorsStatus == kcg_true) ? 't' : 'f';
        DoorStatusesInputBfr[2 * j + 2] = ',';
    }
    DoorStatusesInputBfr[2 * DOORS_NUMBER_InnerLightPkg] = ')';
    DoorStatusesInputBfr[2 * DOORS_NUMBER_InnerLightPkg + 1] = '\0';
    printf("At step %d, set inputs %s\n", i, DoorStatusesInputBfr);
    nResult = SsmSetInputVector(DoorStatusesInputBfr);
    if (nResult) {
        SsmGetLastError(&pszReturnValue);
        printf("%s\n", pszReturnValue);
        return -1;
    }

    /* Play one step and refresh values */
    nResult = SsmStep(1, true);
    if (nResult) {
        SsmGetLastError(&pszReturnValue);
        printf("%s\n", pszReturnValue);
        return -1;
    }
    /* Read outputs */
    nResult = SsmGetOutputVector(&pszVector);
    if (nResult) {
        SsmGetLastError(&pszReturnValue);
        printf("%s\n", pszReturnValue);
        return -1;
    }
    else {
        printf("Output Vector = %s\n", pszVector);
    }
    }
    /* Wait for user interaction */
    SsmDbgStep(1);
    /* Close SCADE Simulator and exit SCADE if not started before */
    if (SsmCloseScadeSimulator()) {
    printf("error close\n");
    return -1;
    }
    return 0;
}
```

Example 2: TestCoSimulation.c file with direct calls to SCADE Suite KCG

```
// This file contains the 'main' function. Program execution begins and ends there.//

#include <stdio.h>
#include <stdlib.h>
#include "InnerLightController_InnerLightPkg.h"
#include "kcg_consts.h"

#ifndef true
#define true 1
#define false 0
#endif

int main()
{
    //DoorStateArray_InnerLightPkg DoorStatuses;
    inC_InnerLightController_InnerLightPkg inC;
    outC_InnerLightController_InnerLightPkg outC;
    int i;
    int nResult;
    const char* pszReturnValue = NULL;
    /* Call the InnerLightController SCADE node initialization function */
    InnerLightController_reset_InnerLightPkg(&outC);

    /* Run the simulation for 31 steps */
    for (i = 0; i < 31; i++) {
    int j;
    char DoorStatusesInputBfr[100];
    //char string[1][100];
    const char* pszVector;
    /* Set inputs */
    DoorStatusesInputBfr[0] = '(';
    for (j = 0; j < DOORS_NUMBER_InnerLightPkg; j++) {
        inC.DoorStatuses[j] = ((i < 5) && (j == 0)) ? kcg_true : kcg_false;
        DoorStatusesInputBfr[2 * j + 1] = (inC.DoorStatuses[j] == kcg_true) ? 't' : 'f';
        DoorStatusesInputBfr[2 * j + 2] = ',';
    }
    DoorStatusesInputBfr[2 * DOORS_NUMBER_InnerLightPkg] = ')';
    DoorStatusesInputBfr[2 * DOORS_NUMBER_InnerLightPkg + 1] = '\0';
    printf("At step %d, set inputs %s\n", i, DoorStatusesInputBfr);

    /* Call the InnerLightController SCADE node cycle reaction function */
    InnerLightController_InnerLightPkg(&inC, &outC);

    /* Read outputs */
    printf("Output: Inner light power = %d\n", outC.InnerLightPower);
    }
    return 0;
}
```

6 / SCADE Suite Simulation Batch

This chapter explains how to use SCADE Suite Simulation in command line to run simulation executables generated from SCADE Suite models.

- "Simulation Modes"
- "Running Simulation Executable"

While running simulation in command line, SCADE Suite Simulation can raise messages detailed in Chapter 8 about "Simulation Command Line Errors".

Simulation Modes

The following table identifies which simulation mode starts depending on command line options and settings. Each simulation mode is described after.

If setting command line with...	SCADE Suite Simulation...
only a state scenario (.sts) or snapshot scenario (.sns)	is set in **interactive command line mode** with the state and cycle number of the scenario
a state scenario (.sts) or snapshot scenario (.sns) with **-out** option	enters in **full batch mode**. The content of the output file corresponds to the values of the last cycle
a simulation scenario (.sss)	runs the scenario and turns into **interactive command line mode**
a simulation scenario (.sss) with **-out** option	enters in **full batch mode**
only an input vector scenario (.in)	runs the scenario and turns into **interactive command line mode**
input vector scenario (.in) with **-out** option	enters in **full batch mode**
no scenarios	enters in **interactive command line mode**

At the end of full batch mode, SCADE Suite Simulation returns the number of played steps. If the list of scenarios contains a state dump load (.sts) or a snapshot (.sns), the steps are counted from the last loaded .sts or .sns file.

INTERACTIVE COMMAND LINE MODE

This simulation mode allows to enable SCADE Suite Simulation interactively via command lines described in "Simulator Commands" on page 83.

It is initialized by default when no options are specified in the command line. The mode allows for interacting directly with the simulation executable: enter commands on the prompt and the executable displays computed results.

This mode can be particularly useful when debugging huge models.

FULL BATCH MODE

This mode runs simulation sessions driven by SCADE Suite Simulation input scenarios (input scenarios, Simulator scenarios, or states) without any human intervention. The outputs for successive simulation cycles are produced in a file. Simulation cycles are computed from the values of the simulation input file to be specified by –scenario option.

In full batch mode, you improve variable visibility using the `-obs` option to list additional observable variables. For a given generated code, the visible and alterable variables are defined by visibility criteria. Consult "Observable Variables and Traceability" on page 567 in *SCADE Suite User Manual*.

Note

The set of observable variables handled by SCADE Suite Simulation varies with code generation options like expansion or non-expansion of blocks.

This mode can be useful to perform activities such as non-regression testing, building test bases, and, generally, processing large amounts of data.

Running Simulation Executable

Before running simulation, it is necessary to generate the simulation executable and a dynamic library named respectively after the simulated node as `<root_node>.exe` and `<root_node>.dll`.

To generate simulation executable and dynamic library

Use the following command line syntax and options:

```
<SCADE_installation_dir>\bin\scade -code <project_name>
                                   -sim
                                   -conf <configuration_name>
```

Possible command options and parameters:

- `<project_name>` is the SCADE Suite project file
- `<configuration_name>` is the selected configuration (*e.g.*, "Simulation" or "KCG")

Once executable and library are generated, run simulation in batch mode by:

- Starting the SCSSMLNC executable, or
- Setting a PATH environment variable and starting the generated simulation executable

To run simulation batch mode using SCSSMLNC executable

Use the following command line syntax and options:

```
SCSSMLNC <root_node>.dll
        [-scenario <filename.sss>* | <filename.in>* | <filename.sts> | <filename.sns>]
        [-out <filename.out> | <filename.csv>]
        [-obs <filename.obs>]
```

Possible command options and parameters:

- **`-scenario`**: Loads all kinds of scenarios as discriminated by file extension (`.sts`, `.sss`, `.sns`, or `.in`). You can specify a list of paths to heterogeneous simulation input files separated by a space. The specified scenarios are played consecutively in list order. **Warning**: Because state scenarios reset simulation when loaded, it is preferable to specify a state scenario first if you do not want to lose data from previous scenarios.
- **`-out`**: Specifies the output file where simulation results are saved and its format (**.out** or `.csv`). Once scenarios are played, simulation exits. This option enables the "Full Batch Mode". If this option is not in the command line, simulation starts in "Interactive Command Line Mode". This option saves only the inputs and outputs of the simulated root operator in the output file. The option below may include additional observable variables in `.out` files.
- **`-obs`**: Specifies a list of additional observable variables in a specified file. These variables are added to the variables generally displayed or saved in the output file at each simulation cycle. The syntax of OBS files is specified in "Format of OBS Files" on page 168.
- From SCADE Suite 5.1 and higher versions, the **-scn** and **-in** options are obsolete.

Example 1: Starts SCADE Suite Simulation in batch mode for the RollControl project

```
<SCADE_installation_dir>\bin\SCSSMLNC RollControl__RollControl.dll
```

To run simulation batch mode using generated executable

1 Set PATH environment variable to following directory:

```
<SCADE_installation_dir>\bin
```

2 Start the generated executable with the following command line syntax and options:

```
<root_node>.exe [-scenario <filename.sss>* | <filename.in>* | <filename.sts> |
<filename.sns>]
                [-out <filename.out> | <filename.csv>]
                [-obs <filename.obs>]
```

Example 2: Starts SCADE Suite Simulation in batch mode for the RollControl project

```
RollControl__RollControl
```

7 / Data Format for SCADE Suite Simulation

This chapter details the files and data formats used while running simulation sessions with SCADE Suite.

- "Simulation Value Format"
- "Simulation Path Format"
- "Simulation Input Files"
- "Simulation Output Files"
- "Format of OBS Files"
- "Example with Simulation Scenario Files"

Simulation Value Format

The simulation value formats SCADE Suite Simulation supports are listed below.

Table 7.1: Possible formats of simulation values

Value	Format
Boolean	`t` or `f`, `T` or `F`, `true` or `false`, TRUE or FALSE
Integers	`[-]?[0-9]+` Signed and unsigned types (int8, int16, int32, int64, uint8, uint16, uint32, uint64) must not be redefined by users. SCADE Suite Simulation checks value range against C type. By default Scade int32 is mapped to C "signed long"; range is [LONG_MIN, LONG_MAX]. Scade int16 is mapped to C "signed short"; range is [SHRT_MIN, SHRT_MAX]...

Float	`[-]?[0-9]*.[0-9]* (e.g., 10.20; .89; or 12.)`
	`[-]?[0-9]+[eE][+-]?[0-9]+ (e.g., 1e6; 2E-5; or 723E+12)`
	`[-]?[0-9]*.[0-9]+[eE][+-]?[0-9]+ (e.g., .78e-5; 78.98E7; or .9E+45)`
	`+Inf | -Inf | NaN`
	Floating-point types (float32, float64) must not be redefined by users. SCADE Suite Simulation checks value range against C type. By default Scade float64 is mapped to C "double"; range is [DBL_MIN, DBL_MAX]. Scade float32 is mapped to C "float"; range is [FLT_MIN, FLT_MAX].

Characters	Printable characters: `[a-zA-Z0-9] | space | [!"$#%&'()*+,-./:;<=>?@[]\^`|{}~]` Non-printable characters: `'\xhh'` Special cases: `'\0', '\t', '\n', '\r' (equiv. to '\x00', '\x09', '\x0a', '\x0d')` Characters must be written between quotes

Table 7.1: Possible formats of simulation values (Continued)

Value	Format

Structured types	List of values can be written in different formats:
	`(value1, value2, value3, ..., valueN)`
	`{labelA:value1, labelB:value2, ... , labelX:valueN}`
	In case of nested structures, the scheme is repeated recursively down to scalar elements. The formats can be written for instance as:
	`((value1, value2), (value3, value4), ... , (valueX, valueY))`
	`{labelA:{labelB:value1}, labelC:{labelD:value2}}`
	You can also mix formats as in:
	`{labelA:{labelC:value10}, labelF:(value1, value2, value3)}`

Arrays	List of values can be written in different formats:
	`(value1, value2, value3, ..., valueN)`
	`{index1:valueA, index2:valueB, ... , indexN:valueX}`
	In case of nested arrays, the scheme is repeated recursively down to scalar elements. The formats can be written for instance as:
	`((value1, value2), (value3, value4), ... , (valueX, valueY))`
	`{index1:{index2:valueA}, index3:{index4:valueB}}`

Strings (char arrays)	Characters between double quotes
Imported types	User-provided format from VTables. Double quotes can be used as value delimiters when needed. Values must be written between quotes.

Options for strings are taken into account only in simulation sessions (but neither by qualified Testing Environment nor by qualified Model Test Coverage):

- **Display char arrays as string**: This option has no impact on the values specified for inputs as scenario or manual entries. It impacts only the values displayed in user interface or written in output files (`.out`).

- **Fill end of strings with padding character**: This option allows entering strings with a length smaller than char array size. When combined with **Display char arrays as string**, the end of strings are not displayed.

Warning

For character, structure/array, and string values, it is recommended to specify their values between curly braces {} for any SCADE Suite Simulation usage.

Check below some examples of simulation value definition.

Example 1: Defining directly the label/index value of a structure/array input for the following definition:

```
Type1 = {label1 : int32, label2 : int32, label3 : int32}
function Op1(i1 : Type1) returns (out : int)
let
   out = i1.label3;
tel
```

Value format for i1 can be for instance:

```
SSM::set Op1/i1 {{1,2,3}}
SSM::set Op1/i1 {{label1:1,label2:2,label3:3}}
SSM::set Op1/i1 {{label3:3}}
```

Example 2: Values specified for inputs (SSM::set, SSM::check) as arrays of char

```
{('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd',
'e')}
{('"', 'P', 'u', 's', 'h', ' ', ''', 'G', 'o', ''', '"', '\t', '\r', '\n',
'\0')}
```

Example 3: Values specified for inputs (SSM::set, SSM::check) as characters between double quotes

```
{"0123456789\x61\x62\x63\x64\x65"}
{""Push 'Go'"\x09\x0d\x0a\x00"}
```

Simulation Path Format

A simulation path is a string of characters composed of the instance hierarchical path and the instance name separated by a separator character:

```
<path>= <HierarchicalPath><sep><LeafItem>[<sep>] | {<path>}
```

The instance hierarchical path represents the hierarchy from the root operator instance to the scope item referencing the leaf item. It has the following pattern:

```
<HierarchicalPath>=
<RootOperatorInstance><sep><ScopeItems><sep>…<sep><ScopeItems><sep><LeafItem>[<sep>]
```

- `<RootOperatorInstance>` is the name of the root operator
- `<ScopeItems>` are the names of items describing a hierarchy
- `<LeafItem>` is the last item of the hierarchy

The following table lists the possible elements representing scope or leaf items:

<ScopeItems>	<sep>	<LeafItem>
Operator instance path item	/	**State** path item
Iterated operator instance path item	/	**Transition** path item
State Machine path item	:	**Variable instance** path item (with or without composite element path)
State path item	:	**Conditional Block** path item
Conditional Block path item	:	
Transition path item	:	

A path separator `<sep>` is suffixed to sub-path elements (suffix optional for leaf items).

Warning

Paths must be specified between curly braces {} for SCADE Suite Simulation and SCADE Test Environment usage whenever paths contain spaces or square brackets [].

All formats that SCADE Suite Simulation supports in simulation paths for each possible design element are listed below.

Table 7.2: Possible formats of simulation paths by elements

Path Item	Path Format	Comment
Operator instance	`<[Package::]*Operator> [occ]`	`occ` is a string value for instance name
Variable instance	`<VariableName>[composite element path]`	composite element path with this pattern: `<.label|[index]>*`
Iterated operator instance	`<[Package::]*Operator> [<occ>] <iteration>*`	`<iteration>` is `[<n>]` and n the iteration index number
State machine	`<StateMachineName>`	
State	`<StateName>`	
Transition	`<<TransitionPriority>>`	
Forked transition	`<<TransitionPriority>.<TransitionPriority>…>`	
If Conditional block	`<CondBlockName>:<then* | [else]*>>`	
When Conditional block	`<CondBlockName>:<MatchName>`	`<MatchName>` is the name of the match element

Expressions in SCADE Suite models require specific path formats as listed below.

Table 7.3: Specific simulation path formats for expressions

Expression...	Path Format	Comment
called in condition of a conditional block	`<CondBlock Name>:<expression>`	`<expression>` can be an operator, a state machine, or a conditional block
called in condition of a transition	`<Transition>:<expression>`	
called in action of a transition		

In all commands using simulation paths, it is possible to use the <path> pathname as explained above or using <alias> defined by SSM::alias simulation command or part of the alias using the syntax is: <alias>[<.label|[index]>*].

Example 1: Samples of valid path formats

```
#Operator instances:
{Package1::Operator2/Input1}
{Package1::Operator3 [1]/Input2}

#Variable instances:
{P1::Operator2/Output2.R1}
{P1::Operator2/Output4[1].ArrInt[0]}

#Iterated operator instances:

  #Probe in iterator map: iteration 1
  {Operator1/Operator2 1[1]/Probe1}

#State machines and states:
{SSM1/SM1:State1:SM2:State3:SM3:State4:Probe1}

   #Path with transition
   {Operator1/SM1:State1:<1>:Probe1}

   #Path with Forked transition
   {Operator1/SM1:State1:<1.1>:Probe1}

#Conditional blocks
{Operator1/IfBlock1:then:Operator2 1/IfBlock1:then:Probe1}
{Operator1/IfBlock1:then:Operator2 1/IfBlock1:then:Operator3 2/
IfBlock1:else:WhenBlock1:value1:Probe1}
{Operator4/WhenBlock1:value1:Probe1}
```

Example 2: Samples of valid path formats with expressions in scenarios

```
#Path to Array of float:
SSM::set {P1::Operator2/Input1} (1.0,2.0,3.0)

#Path to Structure:
SSM::set {P1::Operator2/Input3} {(8,12.2,'Z',value1,(1,2,3))}
SSM::set {P1::Operator2/Input1} {{a:1.0, b:2.0, c:3.0}}

#Path to Array of structure
SSM::set {P1::Operator2/Input4} {((8,12.2,'Z',value1,(1,2,3)),(8,12.2,'Z',value2,(4,5,6)))}
SSM::set {P1::Operator2/Input4} {{0:(8,12.2,'Z',value1,(1,2,3)),
1:(8,12.2,'Z',value2,(4,5,6))}}

#Expression called in the condition of a conditional block
SSM::check {Operator2/IfBlock1:Operator3 1/MyExpression} true
SSM::cycle

#Expression called in the condition of a transition
SSM::set Operator1/Input1 false
SSM::check {Operator1/SM1:State1:<1>:Operator2 2/MyExpression} true
SSM::cycle

#Expression called in the action of a transition
SSM::check {Operator1/SM1:State1:<1>:Probe1} 0
SSM::cycle
```

Simulation Input Files

Several formats of files can be used by SCADE Suite Simulation for simulation inputs.

- "Format of SNS Files"
- "Format of STS Files"
- "Format of SSS Files"
- "Format of IN Files"

Note

These formats can be used to save input files as described in "Recording Simulation Scenarios/Application States" on page 610 from *SCADE Suite User Manual*.

Format of SNS Files

SNS files store at any moment a snapshot of the application state. Snapshot files contain the root input and output values, the memories representing the context structures of root and sub-operators, the sensors, and the current cycle number. Unlike STS files, such files are readable textual files which allow for reviewing scenarios and porting them across SCADE Suite environment and other execution/testing platforms.

In case, users need to execute SCADE Suite KCG-generated code in their own testing environment, it is possible to produce an API able to save and load these snapshot states. For details, see "Producing Code for Testing on External Platforms" on page 82 in *SCADE Test User Manual*.

Format of STS Files

STS files store the state of SCADE Suite Simulation at any given step of a simulation session. Such input files are binary files which cannot be edited by SCADE Suite users.

Format of SSS Files

SSS files are scenario files that describe simulation cycles in terms of input values and expected values (outputs and active probes) using a format editable with simulation commands as described in "Simulator Commands" on page 83.

For examples, see "Example with Simulation Scenario Files" on page 170.

Format of IN Files

IN files are scenario files that describe simulation values in an input vector format that is easily editable. Such input files are divided into the following sections:

```
<Preamble>
<Header>
<Input vector>
```

- **<Preamble>** is a string of characters: #$Version=2.
- **<Header>** is a list separated by spaces or tabs containing the paths of inputs and sensor variables. The information in the list is taken into account to sort out the information described in <Input vector>. The list must contain at least one variable declaration.
- **<Input vector>** describes the sequence of values for inputs and sensor variables separated by spaces or tabs. The size of the input vector is equal to the number of inputs of the simulated operator. The order of values must match the order of the variable declaration in the list section.
- All lines starting with # are considered as comments.

Warning

It is not possible to insert header information or comments on top of this file before the variable declaration or preamble information.

Example:

```
#$VERSION=2
Package1::SensorStruct  Package1::Sensor1  Package1::Op1/Input1  Package1::Op1/Input2
#This is a comment
(1, false, 0.0)         true               false                 false
#Another comment
#Last comment
(2, false, 0.0)         false              true                  true
```

FORMER IN FILE FORMAT

For ascending compatibility, the format of IN files prior to SCADE Suite 6.0 is still supported. This former format of IN files consists of several lines. Each line corresponds to an input vector the size of which is equal to the number

of inputs of the simulated operator. This input vector is a sequence of values, separated by spaces or tabs. Their order must correspond to the order of the simulated operator formal parameters.

Example: Considering the operator with signature (a:int32, b: bool, c:tuple(int32, bool), d: float32), an input file can look like this:

```
1 t (1, f) 2.02
2 f (2, f) –1e-4
```

Simulation Output Files

Several formats of files can be used by SCADE Suite Simulation for simulation outputs.

- "Format of OUT Files"
- "Format of CSV Files"

Format of OUT Files

This format is made up of a sequence of blocks, each one corresponding to a cycle. Each block is a sequence compliant with the following syntax:

```
STEP <current cycle number>
  [SENSOR <sensor instance path> = <value>]*
  [INPUT <input variable instance path> = <value>]*
  [OUTPUT <output variable instance path> = <value>]*
  [ROOT INPUT <root input variable instance path> = <value>]*
  [ROOT OUTPUT <root output variable instance path> = <value>]*
  [LOCAL <local instance path> = <value>]*
  [PROBE <probe instance path> = <value>]*
  [STATE <state instance path> = [<Active>|<Not Active>]]*
  [MATCH <match instance path> = [<Active>|<Not Active>]]*
  [BRANCH <branch instance path> = [<Active>|<Not Active>]]*
  [WEAK TRANSITION <weak transition instance path> = [<Fired>|<Not Fired>]]*
  [STRONG TRANSITION <strong transition instance path> = [<Fired>|<Not Fired>]]*
  [SIGNAL <signal instance path> = [<Emitted>|<Not Emitted>]]*
  [ASSUME <assume instance path> = [<Valid>|<Not Valid>]]*
  [GUARANTEE <guarantee instance path> = [<Valid>|<Not Valid>]]*
```

Example:

```
STEP 1
INPUT i = 67
INPUT j = t
INPUT k = (t, (f, 987))
OUTPUT o3 = 3

STEP 2
INPUT i = -12
INPUT j = f
INPUT k = (f, (t, 101))
OUTPUT o3 = -11
```

The elements present in OUT files are the sensors of the model, followed by the inputs and outputs of the tested operator, and optionally followed by the observed variables and controls as specified in an OBS file. It is possible to specify this OBS file in simulation command line or through the **Obs** keyword in the text file format defining test records.

Format of CSV Files

This format is a line-oriented ASCII file made up of the following components:

- **Comment lines**: lines starting with a '#' character are considered as comment lines. They may appear at any place in the file.
- **Format version**: If the first comment line contains the string `$Version=<number>`, `<number>` is interpreted as the version identifier of the file format. This information is used by tools for handling file format changes transparently to users. `<Version>` is any sequence of decimal digits possibly separated by a decimal point character (`.`).
- **Header**: The first non-comment line is a header line containing the identification of the data contained in the other lines. Each element identifies an input, an output, a probe, or more generally any observable data in the model. The identification contains the path from the root operator, including package identifiers (followed by `::`). Example: `Package1::Oper1/Inp2`. Elements are separated with semicolons (`;`).
- **Data lines**: Non-comment lines following the header line are interpreted as data. There is one line per cycle. Each line contains a list of values, where each value corresponds to the data of the header with the same rank. Elements of that list are separated with semicolons (`;`). The format of individual values is described in "Simulation Value Format" on page 156.

The order of elements in the header and in the data lines is not significant. In particular, it does not need to be the same as the order of inputs/outputs of the root operator. It is only required that the rank of data elements match the rank of their identifiers in the header line.

Example: Sample of CSV file

```
#$Version=1.0
STEP;ABC_N::ABC_N/Lock;ABC_N::ABC_N/Buttons;ABC_N::ABC_N/B;ABC_N::ABC_N/F;ABC_N::ABC_N/
LockLight;
1;false;(false, false);(ABC_N::b_none, ABC_N::b_none);(ABC_N::f_none, ABC_N::f_none);false;
2;false;(false, false);(ABC_N::b_none, ABC_N::b_none);(ABC_N::f_none, ABC_N::f_none);false;
```

Format of OBS Files

An OBS file (extension `.obs`) provides the list of variables to be observed in addition to the inputs/outputs of the tested operator.

Each line must provide a variable path. If this path contains any space, it has to be surrounded by quotes (see example below).

All characters included between the first character following the last quote or the first space, and the end of the line are considered as a comment. In full batch mode, such comment is written after the Scade variable name in the output file.

Note

Only observable variables are accepted (see `SSM::print_all_obs` to see how to display all observable variables in interactive mode).

Example: The following is based on the ABC example. To check that a variable is observable, start the `SSM::print_all_obs` command in interactive mode:

```
C:\Documents and Settings\user\Desktop\ModelExample\ABC_N\Simulation>scssmlnc
ABC_N__ABC_N.dll
SCADE (TM) Simulator Batch Version 6.0d (build i5)
Entering interactive command line mode (ABC_N__ABC_N.dll)
[0]>SSM::print_all_obs

ABC_N::ABC_N/Lock
ABC_N::ABC_N/Buttons
ABC_N::ABC_N/B
ABC_N::ABC_N/F
ABC_N::ABC_N/LockLight
ABC_N::ABC_N/ABC_N::Button 4[0]/background
ABC_N::ABC_N/ABC_N::Button 4[0]/foreground
ABC_N::ABC_N/ABC_N::Button 4[1]/background
ABC_N::ABC_N/ABC_N::Button 4[1]/foreground
[0]>
```

Select the variables to be observed and create your `.obs` file:

```
ABC_N::ABC_N/Lock this is a comment
ABC_N::ABC_N/lockFall
"ABC_N::ABC_N/ABC_N::Button 4[0]/foreground"
```

Start the `scssmlnc` simulation executable with `-scenario`, `-out`, and `-obs` options set (see "Running Simulation Executable" on page 151):

```
scssmlnc ABC_N__ABC_N.dll -scenario MyScenarioFile.sss -out MyOutputFile.out -obs
MyOBSFile.obs
```

The `.obs` file translates as follows in `.out`:

```
In Output file:
STEP 1
…
INPUT ABC_N::ABC_N/Lock = false this is a comment
OUTPUT ABC_N::ABC_N/ABC_N::Button 4[0]/foreground
LOCAL ABC_N::ABC_N/lockFall = false
```

The `.obs` file translates as follows in `.csv` files:

```
In CSV file:
STEP;…; ABC_N::ABC_N/Lock this is a comment;ABC_N::ABC_N/ABC_N::Button 4[0]/foreground
1;…;false;false
```

If a model contains probes, you can observe them and display their outputs in OUT or CSV files.

Notes

Probes are indicated as PROBE in your output files.

The format of OBS files path complies with traceability mechanisms between source and target code as explained in "Traceability from Specification Models" on page 229.

Example with Simulation Scenario Files

The syntax of simulation scenario files is illustrated in the example below.

Example: This example is based on the ABC_N example of the standard SCADE Suite product installation. The following shell command line generates a simulation executable:

```
<SCADE_install_path>\bin\scade -code ABC_N.etp -sim -conf Simulation
```

- `ABC_N.etp` is the SCADE Suite project file
- `Simulation` is the name of the selected configuration

The following shell command line initiates a simulation session (see "Running Simulation Executable"):

```
<Simulation_target_dir>\scssmlnc ABC_N__ABC_N.dll
```

SCADE Suite Simulation commands in interactive mode and their results:

```
SCADE Suite (TM) Simulator Batch Version 6.0 (build i28)
Entering interactive command line mode (ABC_N__ABC_N.dll)
[0]>SSM::print_all_obs
ABC_N::ABC_N/Lock
ABC_N::ABC_N/Buttons
ABC_N::ABC_N/B
ABC_N::ABC_N/F
ABC_N::ABC_N/LockLight
ABC_N::ABC_N/ABC_N::Button 4[0]/background
ABC_N::ABC_N/ABC_N::Button 4[0]/foreground
ABC_N::ABC_N/ABC_N::Button 4[1]/background
ABC_N::ABC_N/ABC_N::Button 4[1]/foreground
[0]>SSM::watch ABC_N::ABC_N/Lock
[0]>SSM::watch ABC_N::ABC_N/Buttons
[0]>SSM::watch ABC_N::ABC_N/LockLight
```

The command below shows the pattern-matching with the previous `watch` command:

```
[0]>SSM::print_watches
ABC_N::ABC_N/Lock
ABC_N::ABC_N/Buttons
ABC_N::ABC_N/LockLight
```

The call to `SSM:default_inputs` below provides a model of *input vector* (see "Format of IN Files" on page 164). Use it to build the real input vector.

```
[0]>SSM::print_watch_values
ABC_N::ABC_N/Lock = false
ABC_N::ABC_N/Buttons = (false,false)
ABC_N::ABC_N/LockLight = false
[0]>puts [SSM::default_inputs]
false (false, false)
```

Three variables are printed since observed variables display after each cycle:

```
[0]>SSM::set_inputs true (false, true)
[0]>SSM::cycle
ABC_N::ABC_N/Lock = true
ABC_N::ABC_N/Buttons = (false,true)
ABC_N::ABC_N/LockLight = false
[1]>SSM::cycle
ABC_N::ABC_N/Lock = true
ABC_N::ABC_N/Buttons = (false,true)
ABC_N::ABC_N/LockLight = true
[2]>SSM::reset
[0]>
[0]>exit
```

8 / Simulation Command Line Errors

This chapter sums up the error codes and messages that can be returned while running simulation sessions in command line.

- "Error Codes"
- "Error Messages"

Note

SCADE Simulator returns messages from code produced by SCADE Suite KCG. In case the simulation code contains imported code, you can get exception error messages due to this code. For information about such messages, please consult the documentation applicable to your imported code.

Error Codes

The following sections provide the list of error codes likely to be returned by a simulation executable or by the SCADE Suite Simulation used in batch mode.

SIMULATION EXECUTABLE

A value different from 0 indicates the presence of an error.

Table 8.1: Error codes possibly returned by simulation executable

Number	Description
0	No error
21	Error during interpretation of a simulation scenario command

Table 8.1: Error codes possibly returned by simulation executable (Continued)

Number	Description
22	Error during analysis of a SCADE value vector
23	TCL_LIBRARY environment variable not set
24	Attempt to display or assign a SCADE variable which is unknown or not observable
25	Invalid number of inputs during a `SSM::set_inputs` command
26	Assertion violated during execution of a simulation cycle
27	Error when opening a file
28	Unknown option or invalid option combination
29	Type error during a SCADE variable assignment attempt
30	Numeric error
31	Other error
32	Error when exception raised

Note

In full batch simulation mode, SCADE Simulator stops as soon as any of these errors is detected.

In interactive mode, SCADE Simulator stops only when errors 23 and 28 occur at simulation executable start.

SCADE.EXE -CODE -SIM

When the simulation executable is called using `scade.exe -code -sim` command line, the errors 23 and 28 return the code provided by the binary itself. For other uses, the following error codes can be returned:

Table 8.2: Error codes possibly returned by SCADE /SM

Number	Description
0	No error
2	Argument missing or incorrect option
3	Inexistent file (or directory) or access denied
4	Abnormal configuration
5	Incompatible options

Error Messages

Errors occur in different situations. In any case, the return code of SCADE Simulator is set.

ERRORS FROM SCADE VARIABLE ASSIGNMENT

When assigning an expression to a SCADE variable, SCADE Simulator checks the types. Assignment is possible only if the expression type complies with the SCADE variable type.

Example: The SCADE variable i is an input of the root operator (thus observable) and of integer type.

```
> SSM::set i 3
```

The above example is executed correctly. Consider the following example:

```
> SSM::set i (1, 2.3, t)
```

It generates the following error message:

```
Type check error. Type of i is: int
```

SYNTAX ERROR IN INPUT VECTORS (DATA-DRIVEN SIMULATION)

Suppose the root operator of the simulated model has one input of integer type and the content of the _"test.in"_ input file is: `1 2 3`.

The command execution `test –in test.in` raises the following message:

```
Bad input list (3 arguments instead of 1) line 1
```

Ansys

Technical Manual

Part 3

Code Generator

- Chapter 9: "Code Production Using SCADE Suite Code Generators"
- Chapter 10: "Multicore Target Integration of Task-Based Code"
- Chapter 11: "Code Generators Command Line Options"
- Chapter 12: "Code Generation Messages"

9 / Code Production Using SCADE Suite Code Generators

This chapter introduces the translation mechanism that handles code production using SCADE Suite KCG or Multicore Code Generator (MCG). The chapter starts with a general overview of the code generators, including their inputs/outputs, principles, and scope. Input and output files are then described in detail with examples of generated code. The access to traceability and pragma information in code is also described in details.

Note

The content of this chapter is intended for an advanced use of code generation. This information is extracted and compiled from *SCADE Suite KCG Tool Operation Requirements (TOR)* to ease access directly from online user documentation. In the framework of certification activities, *SCADE Suite KCG TOR* is the only reference to be used.

- "General Overview"
- "Input and Output Files"
- "Description of Generated Information Files"
- "Description of Generated Files for C Code"
- "Description of External Files for C Code"
- "Description of Generated Files for Ada Code"
- "Description of External Files for Ada Code"
- "Traceability from Specification Models"
- "Pragma Syntax and Usage"

General Overview

SCADE Suite Code Generators take SCADE files (`.xscade/.scade` format, see "SCADE Suite Files" on page 60 from *SCADE Suite User Manual*) as input to produce C/Ada code (KCG) or C code (MCG) that translates the behavior of the input model described in SCADE files. It also produces the SCADE description in `.scade` file format from the input model in `.xscade` file format. The figure below shows the basic types of inputs and outputs around KCG/MCG.

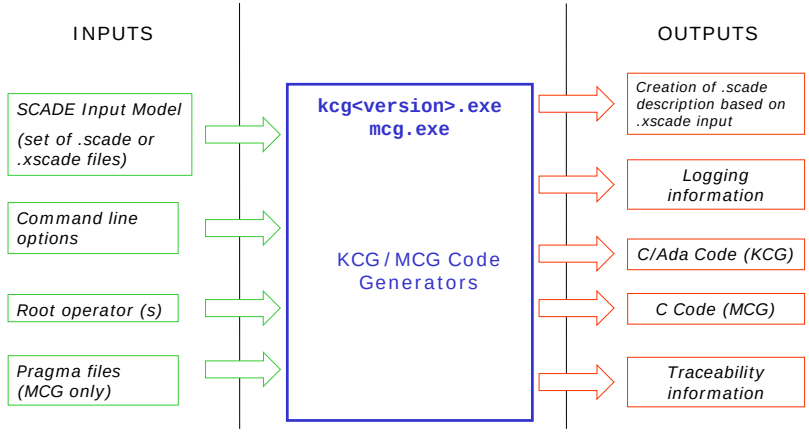


Figure 9.1: Inputs and outputs of KCG / MCG Code Generators

For launching SCADE Suite KCG executable in batch, see "Launching KCG Executable in Command Line" on page 857 in *SCADE Suite User Manual*. For launching Multicore Code Generator executable in batch, see "Launching MCG Executable in Command Line" on page 895 in *SCADE Suite User Manual*.

Note

Check Figure 9.2 or Figure 9.3 respectively about C or Ada code generation for more information on the name and format of input and output files.

SCADE SUITE CODE GENERATOR PRINCIPLES

SCADE Suite Code Generators are based on principles which determine the way code is generated. The table below summarizes these principles:

Table 9.1: SCADE Suite KCG principles

Principle	Meaning
The input model has a *formal definition*	Its content is completely accurate and is formally defined by Scade language semantics.
Scade language is *deterministic*	A given input sequence always produces the same output sequence. The tool implements the language in total respect of the principle of determinism.
KCG or MCG itself is *deterministic*	A given input model always produces the same output C/Ada Code provided that code generation options are fixed.
The generated code is *safe*	There is no dynamic allocation and dynamic accesses are checked.
The generated code is *traceable*	The C or Ada code can be traced back to the input model.

With respect to functionalities and limits, SCADE Suite Code Generators are characterized by the following:

1 KCG/MCG do not check that the produced C or Ada code is compilable. This check can only be performed by a C/Ada compiler.

> **Note**
>
> The generated code is correct but it may rely on external code which is beyond the scope of the tool.

2 KCG/MCG are limited respectively to C and/or Ada functions. The execution of these functions requires a calling program (main) and interfaces (acquisition of inputs and sending of outputs) that must be produced by the user.

Input and Output Files

SCADE Suite Code Generators produce all output files in a target directory.

- By default, `kcg` is the target directory in the working directory
- The target directory can be specified by users with the `-target_dir` command line option (see "Option -target_dir" on page 291).

Note

In case of command-line error unrelated to the input model, the KCG log file is created in the current working directory (see Table 9.4). If the target directory does not exist, KCG/MCG create the directory and its parent directories.

When code generation completes, Code Generators return an exit status:

0: no failure / no error / no warning

1: no failure / no error / at least a warning found

2: no failure / at least an error found / any warning

3: at least a failure found / any error/ any warning (if `kcg.log` correctly generated)

4: `kcg.log` not correctly generated

Note

If `-warn_error` option is set, KCG/MCG return either 1 or 2 exit status (see "Option -warn_error" on page 292). For code generation error codes and messages, see Chapter 12 about "Code Generation Messages".

Read about the nature, content, and names of input and output files, as well as possible checks to perform after completion of code generation.

- "Schema of Input/Output Files"
- "Generated Files"
- "User-Provided Input Files"

For final checks and recommendations, consult the Tool Operational Requirement (TOR) and Tool Accomplishment Summary (TAS) in your Certification Kit.

Schema of Input/Output Files

SCADE Suite code generation is based on a set of various input and output files for C or Ada code generation.

- "Input/Output Files for C Code Generation"
- "Input/Output Files for Ada Code Generation"

INPUT/OUTPUT FILES FOR C CODE GENERATION

The following figure represents all input files/information required for C code generation as well as all output files generated by SCADE Suite KCG and Multicore Code Generator.

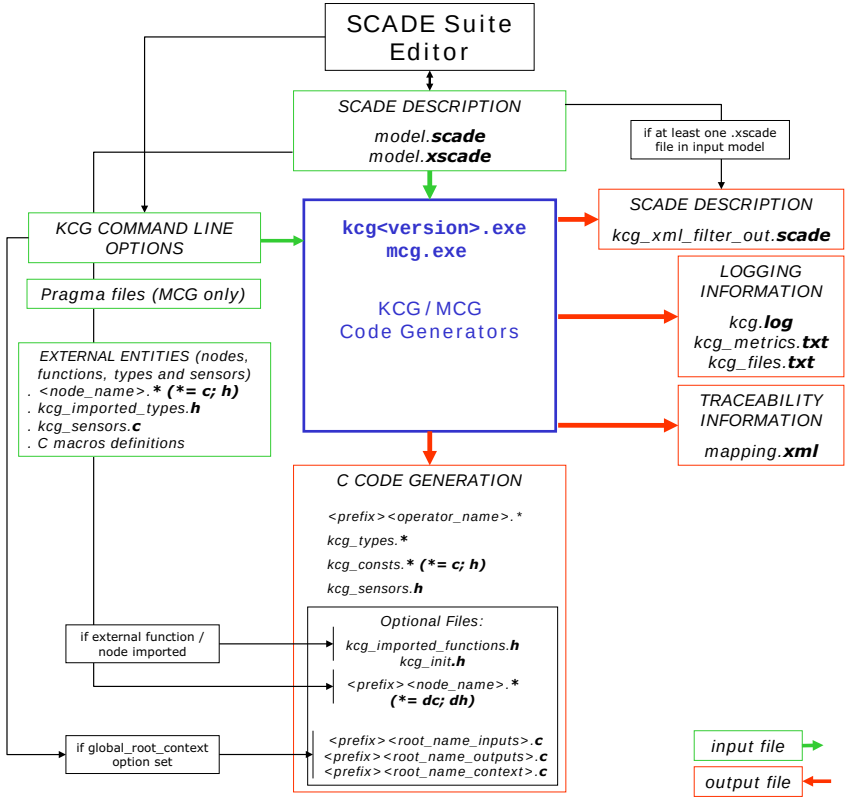


Figure 9.2: Input and output files for C code generation with KCG/MCG

INPUT/OUTPUT FILES FOR ADA CODE GENERATION

The following figure represents all input files/information required for Ada code generation as well as all output files generated by SCADE Suite KCG.

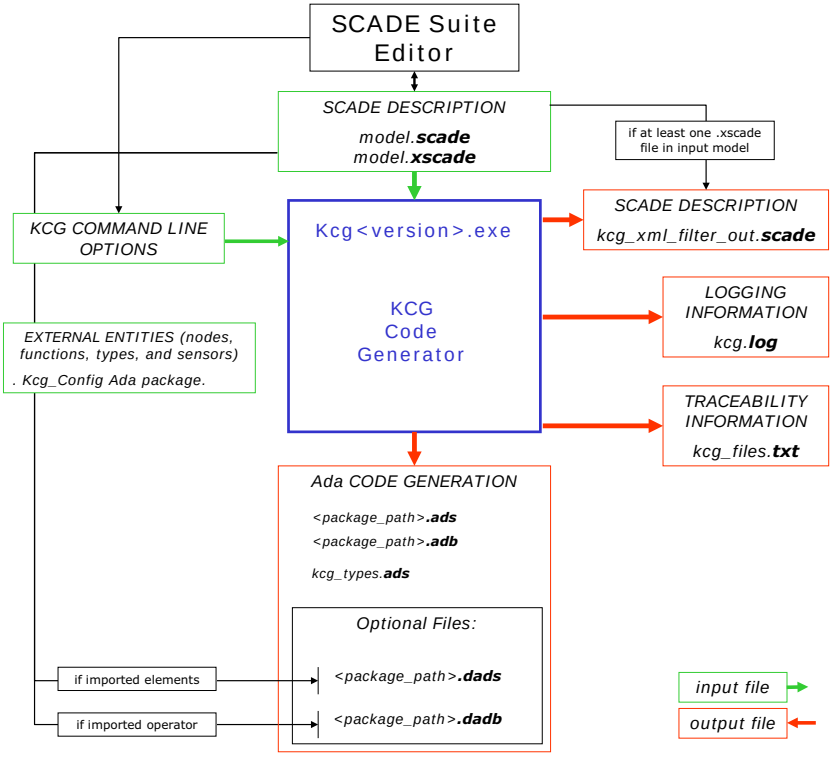


Figure 9.3: SCADE Suite KCG input and output files for Ada code generation

User-Provided Input Files

Depending on the input model (sensors, imported nodes, constants, types, or functions), several files must be provided. Check below the input files to be provided by users.

- "User Input Files for C Code Generation"
- "User Input Files for Ada Code Generation"

USER INPUT FILES FOR C CODE GENERATION

The `kcg_assign` macro must be provided either through `-user_config` option mechanism or through a `kcg_assign.h` file. The macro is used by the generated C code to perform low-level copy of non-scalar data. It uses the C `memcpy` library function for simulation purposes and has to be redefined by users according to final target environment. Other input files may be required depending on model content as listed in the table below.

Table 9.2: Input files provided by users for C code generation

If the model contains...	The user must ...
non-imported user types	provide a definition for each named type
sensors	provide the sensor definition in `kcg_sensors.c`
imported types	provide for each imported type: • the imported type definition in `kcg_imported_types.h` • the copy and compare functions if the type is not scalar the initialization function
imported constants	provide the imported constant definitions in `kcg_consts.h`
imported nodes	• complete the template files generated by KCG • rename the `.dh` and `.dc` templates using `.h` and `.c` extensions (*e.g.*, `<node_name>.h` and `<node_name>.c`)

For instructions about code generation with imported C code, consult "Implementing Imported C Code" on page 861 in *SCADE Suite User Manual*.

Users may also need to provide the definition of C macros (KCG/MCG generate their names) in the file given as argument for `-user_config` option to be able to compile code. Users must provide the `kcg_assume` and `kcg_guarantee` macros if the `-keep_asserts` option is set.

USER INPUT FILES FOR ADA CODE GENERATION

Table 9.3: Input files provided by users for Ada code generation

If the model contains...	The user must ...
base types	provide the base type definitions in `Kcg_Config` Ada package: • Kcg_Char corresponding to **char** type • Kcg_IntN corresponding to **int**N types (int8, int16, int32, int64) • Kcg_UintN corresponding to **Uint**N types (uint8, uint16, uint32, uint64) • Kcg_Size corresponding to **size** type • Kcg_Shift_Left_UintN, Kcg_Shift_Right_UintN, and Kcg_Lnot_UintN corresponding to the dynamic semantics of bitwise operators • Kcg_Assume corrresponding to procedure called for Assume contracts • Kcg_Guarantee corrresponding to procedure called for Guarantee contracts
sensors	complete the Ada code for the sensors. The `.dads` file generated by KCG can be used as a template.
imported nodes/ functions	provide the Ada code for the imported operator. The `.dads` and `.dadb` files generated by KCG can be used as a template.
imported types/ constants	provide the Ada type definition of the type. The `.dads` file generated by KCG can be used as a template.

For instructions about code generation with imported Ada code, consult "Implementing Imported Ada Code" on page 867 in *SCADE Suite User Manual*.

Generated Files

As shown on Figure 9.1, Figure 9.2, and Figure 9.3, SCADE Suite KCG and Multicore Code Generator automatically generate the C or Ada code associated with the input model, as well as a number of other information. Check below the lists all generated files.

- "Output Files with Code Generation Information"
- "Output Files in C Code Generation"
- "Output Files in Ada Code Generation"

Note

Depending on the options set for KCG/MCG, the number of generated output files differs (see "KCG/MCG Execution and Compatible Options" on page 284). A line in the generated code cannot exceed 2048 characters unless composed of a single identifier longer than 2048 characters. Generated Ada-specific file names are prefixed according to the `-root_package` option.

OUTPUT FILES WITH CODE GENERATION INFORMATION

Table 9.4: List of KCG/MCG-generated information files

Information Type	File name	Description
SCADE Suite model description	kcg_xml_filter_out.scade	Contains the translation of the input model in graphical format into the Scade textual format
Logging information	kcg.log	Summarizes the execution of the code generation
	kcg_files.txt	Contains the list of all generated files
	kcg_metrics.txt	Contains metrics information for the data structures, control flow, and program size in the generated C code files (**not for Ada code**)
Traceability information	mapping.xml	Mapping file in XML format enabling full traceability between source model and generated code

OUTPUT FILES IN ADA CODE GENERATION

Table 9.5: List of SCADE Suite KCG-generated files for Ada code

File name	Description
<package_path>.ads	Contains Ada code of non-imported SCADE declarations for each Scade package in the cone of influence[1] of the root operator.
<package_path>.adb	Contains Ada code of non-imported SCADE operators for each Scade package containing at least one operator in the cone of influence of the root operator(s).
kcg_types.ads	Contains Ada type declarations related to the types defined in the input model. This package contains "base types" which are either: • enum types • structured (arrays, records) types • user-defined types declared with a pragma manifest All other user-defined types are declared as "subtypes" inside their declaration package (<package_path>.ads).
<package_path>.dads	Contains template Ada code for the specification of imported elements.
<package_path>.dadb	Contains template Ada code for each monomorphic instance of imported operator.

1. The cone of influence of an operator is the set of operators called by this operator or any operator in the cone.

OUTPUT FILES IN C CODE GENERATION

Table 9.6: List of KCG/MCG-generated files for C code

File name	Description
<root_name>.h <root_name>.c	Contains C code for each root operator
<operator_name>.h <operator_name>.c	Contains C code for each monomorphic[1] instance of a non-expanded user-defined operator in the cone of influence of the root operator
kcg_types.c kcg_types.h	Contains C code related to the types defined in the input model
kcg_consts.c kcg_consts.h	Contains C code related to the constants defined in the input model
kcg_sensors.h	Contains C code related to the sensors defined in the input model
kcg_imported_functions.h	Contains C code declaration of imported functions contained in the cone of influence of the root operator
kcg_init.h	Contains C code declaration of initialization and reset functions for each monomorphic instantiation of an imported node in the cone of influence of a root operator
<node_name>.dc <node_name>.dh	Contains template C code for each monomorphic instance of imported nodes, if any
<root_name>_inputs.c <root_name>_outputs.c <root_name>_context.c	C code files generated when setting the option -global_root_context which contain respectively the declaration of inputs, outputs, and root operator's context

1. A monomorphic instantiation is an instantiation with a monomorphic signature (signature without type parameters). Operator monomorphization applies to generic operators and creates a specific non-generic instance for any instantiation of the generic operator defined with non-generic types.

Description of Generated Information Files

SCADE Suite KCG and Multicore Code Generator (MCG) generate various files containing different level of information regarding the model and the code generation run. These files are explained in details below.

- "kcg_xml_filter_out.scade"
- "kcg.log"
- "kcg_files.txt"
- "kcg_metrics.txt"

RELATED TOPICS

- "File Header Banner" on page 231
- "File Footer Banner" on page 232

KCG_XML_FILTER_OUT.SCADE

By default, KCG/MCG generate a `kcg_xml_filter_out.scade` file containing the translation of the graphical format of the input model into Scade textual format.

Note

If the `-xml_basename` option is set, KCG/MCG replaces the default name by the name specified in the option argument.

KCG.LOG

KCG/MCG generate a `kcg.log` file containing:

- list of raised failures, errors, and warnings (if any)
- total number of warnings
- total number of errors
- presence or absence of failures

If a command line error is raised, the log file is generated in the working directory. If no error, the log file is generated in the target directory as other files. The log format is always made of a message with a summary.

KCG_FILES.TXT

KCG/MCG generates a `kcg_files.txt` file containing the list of all files generated during code generation. Unlike other files, this one has no header/footer banner info.

KCG_METRICS.TXT

KCG/MCG provides several metrics on the generated C code (not for Ada code). Such information is gathered into the `kcg_metrics.txt` file. The list of available metrics is detailed below.

	The metric represents...
Data structures:	
Max members in a single structure	the maximum number of fields that any **`typedef struct`** is holding
Max number of dimensions for an array	the maximum of the number of dimension in any array declaration (the number of dimension of an array declaration based on another array is the number of dimensions of this second array incremented by one)
Maximum size for a dimension	the greatest size (i.e. number of elements it contains) of an array declaration
Max of enum constants in a single enumeration	the maximum number of enumerated values for any **`typedef enum`**
Control structures:	
Max numbers of case labels in a switch	the maximum number of case labels in any **`switch`** statement
Max numbers of nesting levels in compound statement	the maximum level of imbricated **`switch`**, **`if`**, and **`for`** constructions
Program size:	
Max number of function parameters in definition	the maximum number of parameter present in any function or macro definition
Max number of function arguments in a call	the maximum number of arguments in a function call

Max number of nesting levels of parenthesized expression in a full expression	the maximum level of nested parenthesis in any expression
Max number of identifiers with block scope declared in one block	the maximum amount of local variables declared in any block
Max number of macro identifiers simultaneously defined in one preprocessing translation unit	the maximum number of macro definitions simultaneously defined in one preprocessing translation unit
Max number of extern identifiers in one translation unit	the maximum number of extern declarations of identifiers in one translation unit

Description of Generated Files for C Code

This section describes the structure and content of generated files for C code:

- "Standard Generated Files in C"
- "Additional Generated Files in C"
- "File Inclusion Protection"

Note

All figures and examples in this section are based on files generated for the RollControl input model (see project examples from SCADE startup page).

The figure below presents the list of generated C files as they appear in the Output Window of SCADE Suite after successful code generation (see "Launching KCG Code Generation from UI" on page 855 in *SCADE Suite User Manual*):

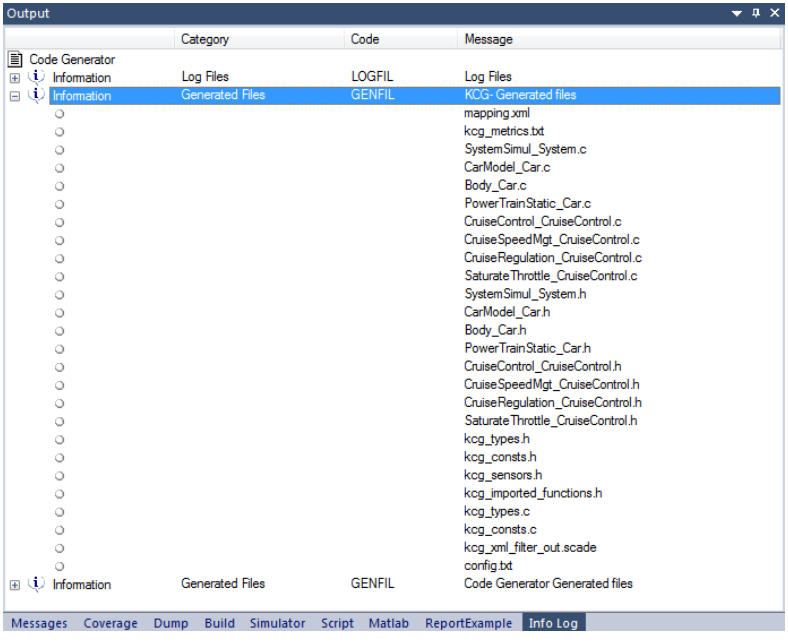


Figure 9.4: List of generated C files (Output Window in SCADE Suite)

Standard Generated Files in C

All standard files generated by KCG/MCG for C code integration are explained in details below. For each file, read a description of their content with illustrations and examples.

- "<root_name>.h"
- "<root_name>.c"
- "<operator_name>.h"
- "<operator_name>.c"
- "kcg_types.h"
- "kcg_types.c"
- "kcg_consts.h"
- "kcg_consts.c"
- "kcg_sensors.h"

FILE NAMING SCHEME AND LENGTH

Among the standard files generated, the C and H files based on the name of any operator or root operator in the model follow a naming pattern for C identifiers. By convention, a Scade path P1::P2::..::Pn::ID becomes as C identifier ID_P1_P2_..Pn.

In addition, the identifiers are modified in the following cases:

- Identifiers are prefixed if needed to ensure that two identifiers cannot have the same n first characters, n being the significance length value.
- Identifiers are truncated according to the name length value.
- When `-globals_prefix` option is set, they are prefixed.

Note

In case of name collision, a new non-conflicting name is generated by KCG/MCG.

DEFINITION OF OPERATOR CONTEXT

The context of an operator is the set of variables composed of all its memories and recursively the contexts of sub-operator instances (except the ones with an attached `C:task` pragma). Depending on code generation options, the context may contain all outputs, all probes, some local variables, all assertions, and variables containing the clock value of observable variables. The context needs the following functions:

- *initialization function* to initialize the context by taking a pointer as unique parameter and setting the pointed value to a valid value of its type. Memories are initialized as in reset function and other fields are initialized to a valid value of their type. Initialization functions can be used by the integration code to initialize root operator contexts. The prototype and definition of these functions are enclosed in preprocessor directives as:

```
#ifndef KCG_USER_DEFINED_INIT
...
#endif /* KCG_USER_DEFINED_INIT */
```

- *reset function* to reinitialize instance memories at initial tick by taking the context of an operator as unique parameter. Reset functions may be called by the cycle function generated by KCG/MCG or by user-provided code to reset the application. For each function not called directly or indirectly by a cycle function, the prototype and definition of the function are enclosed as follows in preprocessor directives:

```
#ifndef KCG_NO_EXTERN_CALL_TO_RESET
...
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */
```

- *cycle function* to compute outputs and new memories of an instance from a set of inputs and previous memories (if any) encoding the state of Scade instances. An operator always requires at least a cycle function.

A valid context is a value of the context that can be reached, starting with a context initialized as the reset method does, by applying the cycle method on an input sequence.

A memory is a variable used to encode the value of a flow at previous instants (pre, fby, last, times constructs). It can be a memorized output, a variable memorizing the state of an automaton, or a memorized probe.

<ROOT_NAME>.H

KCG/MCG generate for each root operator a `<root_name>.h` containing:

1. Header (see "File Header Banner" on page 231)

3 Directives for file inclusion protection at start/end of file

2 Inclusion of `kcg_types.h` and header files of operators called by the root operator

3 "Input structure" named `inC_<root_name>` that collects root operator inputs (if root operator has at least one input, not marked for separate I/O (see Table 11.4), and `-global_root_context` option is not set)

4 "Context type" structure named `outC_<root_name>` (if corresponding generated structure not empty) and made of:

- outputs (if no `-global_root_context` option set and root operator is not marked for separate I/O)
- root operator probes (if **`-probes`** option is set)
- initialization variables
- root operator memories
- context for each non-expanded sub-operators requiring a context (except the ones with an attached `C:task` pragma)

5 "State vector type" structure (if `-state_vector` option is set and if the root operator has a memory) named SV `<root name>` and made of:

- memorized outputs
- root operator memorized probes (if `-probes` option is set)
- initialization variables
- root operator memories
- state vector of each non-expanded sub-operator needing C state (one state vector per instance)

6 Functions to save and load the state vector (if `-state_vector` option is set and if the root operator has a memory):

```
extern void kcg_save_SV_<root_name>(SV_<root_name> *SV, outC_<root_name> *outC);
extern void kcg_load_SV_<root_name>(outC_<root_name> *outC, SV_<root_name> *SV);
```

The first one copies the state variables of the context structure into a state vector. The second one restores the state variables of a state vector into a context. Notice that the corresponding cycle functions has to be called at least once after loading to ensure the fact that all observable variables hold valid values.

7 Prototypes of initialization function (if any), reset function (if any), and cycle function(s):

```
Initialization function: extern void <root_name>_init<_path>(outC_<root_name> *outC);
Reset function: extern void <root_name>_reset<_path>(outC_<root_name> *outC);
Cycle function (KCG): extern void <root_name><_path>(inC_<root_name> *inC, outC_<root_name>
*outC);
Cycle function (MCG): extern void <root_name><_path><_suffix>(inC_<root_name> *inC,
outC_<root_name> *outC);
```

where <root_name> is the name of the root operator given as argument of the **-root** option. The <_path> corresponds to the package hierarchy leading to this operator. The inC (resp. outC) parameter does not appear in the prototypes if the root operator has no input (resp. and empty context).

8 Footer (see "File Footer Banner" on page 232)

Check the description of a root operator generated header file hereafter.

Note

The content of header files is different if -global_root_context option is set.

Example: Find below the generated `RollControl_RollControl.h` header file (RollControl root operator of `RollControl` package)

```
/* $************** KCG Version 6.4 beta (build i6) **************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */
#ifndef _RollControl_RollControl_H_
#define _RollControl_RollControl_H_

#include "kcg_types.h"
#include "RollRateCalculate_RollRate.h"
#include "RollRateWarning_RollRate.h"
#include "RollMode_RollMode.h"
#include "RisingEdge_digital.h"

/* ========================  input structure  ====================== */
typedef struct {
  kcg_real /* RollControl::RollControl::joystickCmd */ joystickCmd;
  kcg_real /* RollControl::RollControl::leftAdverseYaw */ leftAdverseYaw;
  kcg_real /* RollControl::RollControl::rightAdverseYaw */ rightAdverseYaw;
  kcg_bool /* RollControl::RollControl::onOffPressed */ onOffPressed;
} inC_RollControl_RollControl;

/* ========================  context type  ========================== */
typedef struct {
  /* ---------------------------  outputs  --------------------------- */
  kcg_real /* RollControl::RollControl::rollRate */ rollRate;
  kcg_bool /* RollControl::RollControl::leftWarning */ leftWarning;
  kcg_bool /* RollControl::RollControl::rightWarning */ rightWarning;
  TRollMode_RollMode /* RollControl::RollControl::mode */ mode;
  /* ----------------------  no local probes  ---------------------- */
  /* -----------------  no initialization variables  ----------------- */
  /* ----------------------  no local memory  ---------------------- */
  /* --------------------  sub nodes' contexts  -------------------- */
  outC_RisingEdge_digital /* 1 */ _1_Context_1;
  outC_RollMode_RollMode /* 1 */ Context_1;
  /* -----------------  no clocks of observable data ----------------- */
} outC_RollControl_RollControl;

/* ===========  node initialization and cycle functions  =========== */
/** "Title_1" {Title = "Title : ROLL RATE CONTROL APPLICATION"} */
/** "Author_1" {Author = "Created by : ESTEREL TECHNOLOGIES"} */
/* RollControl::RollControl */
extern void RollControl_RollControl(
  inC_RollControl_RollControl *inC,
  outC_RollControl_RollControl *outC);

extern void RollControl_reset_RollControl(outC_RollControl_RollControl *outC);
extern void RollControl_init_RollControl(outC_RollControl_RollControl *outC);

/* ======================  state vector type  ====================== */
typedef struct {
  /* ---------------------  no memorised outputs  -------------------- */
  /* ----------------------  no local memory  ---------------------- */
  /* ----------------------  no local probes  ---------------------- */
  /* -----------------  no initialization variables  ----------------- */
  /* --------------------  sub nodes' contexts  -------------------- */
  SV_RollMode_RollMode Context_1;
  SV_RisingEdge_digital _1_Context_1;
} SV_RollControl_RollControl;

extern void kcg_save_SV_RollControl_RollControl(
  SV_RollControl_RollControl *SV,
  outC_RollControl_RollControl *outC);
extern void kcg_load_SV_RollControl_RollControl(
  outC_RollControl_RollControl *outC,
  SV_RollControl_RollControl *SV);

#endif /* _RollControl_RollControl_H_ */
/* $************** KCG Version 6.4 beta (build i6) **************
** RollControl_RollControl.h
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */
```

1. Trace header
2. Directives
3. Header files
4. Input structure
5. Context type
6. Cycle, Reset and Initialization functions
7. State vector type
8. Save and restore functions
9. Trace footer

<ROOT_NAME>.C

KCG/MCG generate for each root operator a `<root_name>.c` source file containing:

1. Header (see "File Header Banner" on page 231)
2. Inclusion of header files: `kcg_consts.h`, `kcg_sensors.h`, and `<root_name>.h`
3. Definition of initialization function enclosed in #ifndef KCG_USER_DEFINED_INIT /#endif preprocessor directives
4. Definition of reset function
5. Definition of cycle function(s)
6. Save and restore functions (if the state vector data structure is generated in the header file):

```
void kcg_save_SV_<root_name>(SV_<root_name> *SV, outC_<root_name> *outC){ ... }
void kcg_load_SV_<root_name>(outC_<root_name> *outC, SV_<root_name> *SV){ ... }
```

7. Footer (see "File Footer Banner" on page 232)

Check the description of a root operator generated source file hereafter.

Example: Find below the generated `RollControl_RollControl.c` source file (RollControl root operator of `RollControl` package).

```c
/* $************** KCG Version 6.4 beta (build i6) **************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RollControl_RollControl.h"

void RollControl_init_RollControl(outC_RollControl_RollControl *outC)
{
  outC->mode = off_RollMode;
  outC->rightWarning = kcg_true;
  outC->leftWarning = kcg_true;
  outC->rollRate = 0.0;
  RollMode_init_RollMode(&outC->Context_1);
  RisingEdge_init_digital(&outC->_1_Context_1);
}


void RollControl_reset_RollControl(outC_RollControl_RollControl *outC)
{
  /* 1 */ RollMode_reset_RollMode(&outC->Context_1);
  /* 1 */ RisingEdge_reset_digital(&outC->_1_Context_1);
}

/** "Title_1" {Title = "Title : ROLL RATE CONTROL APPLICATION"} */
/** "Author_1" {Author = "Created by : ESTEREL TECHNOLOGIES"} */
/* RollControl::RollControl */
void RollControl_RollControl(
  inC_RollControl_RollControl *inC,
  outC_RollControl_RollControl *outC)
{
  kcg_real tmp;

  outC->rollRate = /* 1 */
    RollRateCalculate_RollRate(
      inC->joystickCmd,
      inC->leftAdverseYaw,
      inC->rightAdverseYaw);
  /* 1 */ RisingEdge_digital(inC->onOffPressed, &outC->_1_Context_1);
  if (0. <= outC->rollRate) {
    tmp = outC->rollRate;
  }
  else {
    tmp = - outC->rollRate;
  }
  /* 1 */
  RollMode_RollMode(tmp, outC->_1_Context_1.RE_Output, &outC->Context_1);
  outC->mode = outC->Context_1.mode;
  /* 1 */
  RollRateWarning_RollRate(
    outC->rollRate,
    &outC->leftWarning,
    &outC->rightWarning);
}

void kcg_save_SV_RollControl_RollControl(
  SV_RollControl_RollControl *SV,
  outC_RollControl_RollControl *outC)
{
  kcg_save_SV_RollMode_RollMode(&SV->Context_1, &outC->Context_1);
  kcg_save_SV_RisingEdge_digital(&SV->_1_Context_1, &outC->_1_Context_1);
}

void kcg_load_SV_RollControl_RollControl(
  outC_RollControl_RollControl *outC,
  SV_RollControl_RollControl *SV)
{
  kcg_load_SV_RollMode_RollMode(&outC->Context_1, &SV->Context_1);
  kcg_load_SV_RisingEdge_digital(&outC->_1_Context_1, &SV->_1_Context_1);
}

/* $************** KCG Version 6.4 beta (build i6) **************
** RollControl_RollControl.c
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */A
```

1. Trace header
2. Header files
3. Initialization function
3. Reset function
4. Cycle function
5. Save and restore functions
6. Trace footer

<OPERATOR_NAME>.H

KCG/MCG generate for each non-expanded monomorphic instance of a node in the cone of influence of the root operator an `<operator_name>.h` file containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file
3. Inclusion of header files: `kcg_types.h` and all `<operator name>.h` corresponding to non-expanded sub-operators called by the operator
4. "Context type" structure named `outC_<operator_name>` (if corresponding generated structure not empty) and made of:
   - outputs (if operator not marked for separate I/O, see Table 11.4)
   - operator probes (if **`-probes`** set), initialization variables, and operator memories
   - context for each non-expanded sub-operator instance requiring a context
5. "State vector type" structure (if the operator has a memory, namely if declared as a node without warning at type check) named `<SV_<root_name>` and made of:
   - memorized outputs and memorized probes (if **`-probes`** set)
   - initialization variables and operator memories
   - state vector of each non-expanded sub-operator needing C state (1 vector/instance)
6. Functions to save and restore the state vector (if operator has a memory):

```
extern void kcg_save_SV_<operator_name>(SV_<operator_name> *SV, outC_<operator_name> *outC);
extern void kcg_load_SV_<operator_name>(outC_<operator_name> *outC, SV_<operator_name> *SV);
```

The first one copies the state variables of the context structure into a state vector. The second one restores the state variables of a state vector into a context. Notice the corresponding cycle functions have to be called at least once after loading to ensure that all observable variables hold valid values.

7 Prototype of initialization function enclosed in #ifndef KCG_USER_DEFINED_INIT /#endif preprocessor directives for non-expanded or imported operator:

```
extern void <operator_name>_init<_path> (outC_<operator_name> *outC);
```

The initialization function is generated for the contexts of nodes and assumes such functions exist for the contexts of imported nodes and for objects of imported types which appear in the contexts of generated nodes.

8 Prototype of reset function for non-expanded or imported operator:

```
extern void <operator_name>_reset<_path> (outC_<operator_name> *outC);
```

The reset function is not generated if the operator has an empty context.

9 Prototype of cycle function(s) (for non-expanded operator, imported operator, or root operator with `separate_io` pragma):

- Case of operator not marked for separate I/O (see Table 11.4):

```
extern void <operator_name> <_path> (<inputs>, outC_<operator_name> *outC) ; (KCG)
extern void <operator_name> <_path> <_suffix> (<inputs>, outC_<operator_name> *outC) ; (MCG)
```

- Case of operator marked for separate I/O (see Table 11.4):

```
extern void <operator_name> <_path> (<inputs>, <outputs>, outC_<operator_name> *outC) ; (KCG)
extern void <operator_name> <_path> <_suffix> (<inputs>, <outputs>, outC_<operator_name>
*outC) ; (MCG)
```

Example: Find below the code of the cycle function generated for the RollMode node

```
extern void RollMode_RollMode(
  /* RollMode::RollMode::absRollRate */kcg_real absRollRate,
  /* RollMode::RollMode::onOffPressed */kcg_bool onOffPressed,
  outC_RollMode_RollMode *outC);
```

- Case of a Scade function returning a single scalar:

```
extern <output_type> <operator_name><_path>(<inputs>) ; -- when there are some inputs
extern <output_type> <operator_name><_path>(void) ; -- when there are no inputs
```

Example: The following shows the code of the cycle function generated for a function returning a scalar (RollRateCalculate function)

```
extern kcg_real RollRateCalculate_RollRate(
  /* RollRate::RollRateCalculate::joystickCmd */kcg_real joystickCmd,
  /* RollRate::RollRateCalculate::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::RollRateCalculate::rightAdverseYaw */kcg_real rightAdverseYaw);
```

- Any Scade function with outputs:

```
extern void <operator_name><_path>(<inputs>, <outputs>) ;
```

Example: The following shows the code of the cycle function generated for RollRateWarning function

```
extern void RollRateWarning_RollRate(
  /* RollRate::RollRateWarning::rollRate */kcg_real rollRate,
  /* RollRate::RollRateWarning::leftWarning */kcg_bool *leftWarning,
  /* RollRate::RollRateWarning::rightWarning */kcg_bool *rightWarning);
```

- Any other Scade function:

```
extern void <operator_name><_path>(<inputs>) ;
```

  - `<outputs>`: comma-separated list of pointers to operator outputs along with their type.
  - `<inputs>`: comma-separated list of parameters, each parameter i the C counterpart of the operator input with its type. They are passed by values when they belong to a scalar type, by address otherwise
  - `<operator_name>`: name of current operator and `<_path>` is the package hierarchy leading to the operator. The `outC` parameter does not appear in the prototype if the operator has an empty context.

10 Footer (see "File Footer Banner" on page 232)

Note

Depending on the options set in batch command line, the previously described structure can vary. Involved options are: -input_threshold, -exp, and -expall

<OPERATOR_NAME>.C

KCG/MCG generate for each `<operator_name>.h` file an `<operator_name>.c` source file containing:

1. Header (see "File Header Banner" on page 231)
2. Inclusion of header files: `kcg_consts.h`, `kcg_sensors.h`, and `<operator_name>.h`
3. Definition of initialization function, enclosed in #ifndef KCG_USER_DEFINED_INIT /#endif preprocessor directives, if the operator requires a context
4. Definition of reset function if the operator requires a context
5. Definition of cycle function(s) for the operator
6. Save and load functions (if the state vector data structure is generated in the header file) for the source file of non-expanded operator:

```
void kcg_save_SV_<operator_name>(SV_<operator_name> *SV, outC_<operator_name> *outC){...}
void kcg_load_SV_<operator_name>(outC_<operator_name> *outC, SV_<operator_name> *SV){...}
```

7. Footer (see "File Footer Banner" on page 232)

Check the description of an operator generated source file hereafter.

Example: Find below the generated `RisingEdge_digital.c` source file




```
/* $************** KCG Version 6.4 beta (build i6) **************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RisingEdge_digital.h"

void RisingEdge_init_digital(outC_RisingEdge_digital *outC)
{
  outC->init = kcg_true;
  outC->rem_RE_Input = kcg_true;
  outC->RE_Output = kcg_true;
}


void RisingEdge_reset_digital(outC_RisingEdge_digital *outC)
{
  outC->init = kcg_true;
}

/* digital::RisingEdge */
void RisingEdge_digital(
  /* digital::RisingEdge::RE_Input */ kcg_bool RE_Input,
  outC_RisingEdge_digital *outC)
{
  kcg_bool tmp;

  if (outC->init) {
    tmp = !RE_Input;
  }
  else {
    tmp = !outC->rem_RE_Input;
  }
  outC->RE_Output = tmp & RE_Input;
  outC->rem_RE_Input = RE_Input;
  outC->init = kcg_false;
}

void kcg_save_SV_RisingEdge_digital(
  SV_RisingEdge_digital *SV,
  outC_RisingEdge_digital *outC)
{
  SV->init = outC->init;
  SV->rem_RE_Input = outC->rem_RE_Input;
}

void kcg_load_SV_RisingEdge_digital(
  outC_RisingEdge_digital *outC,
  SV_RisingEdge_digital *SV)
{
  outC->init = SV->init;
  outC->rem_RE_Input = SV->rem_RE_Input;
}

/* $************** KCG Version 6.4 beta (build i6) **************
** RisingEdge_digital.c
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */
```

KCG_TYPES.H

KCG/MCG generate a `kcg_types.h` header file containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file
3. Directive for:
   - `stddef.h` conditional inclusion (in **KCG 6.6.4 only**) controlled by the absence of KCG_NO_STDDEF_H as follows:

```
#ifndef KCG_NO_STDDEF_H
#include "stddef.h"
#endif /*KCG_NO_STDDEF_H*/
```

- `stddef.h` inclusion (in **KCG 6.6.2/MCG**)

4 Empty definition of KCG_MAPW_CPY macro

5 Include directive for a user header file specified by `-user_config` (if any)

6 Definition for basic types:

7 Inclusion of `kcg_imported_types.h` if model with imported types

8 Macro definitions for wrapped operators if necessary:

- all predefined numeric types (intN, uintN, size, floatN) in `kcg_lit_t` macro
- all predefined unsigned types (uintN) in `kcg_lnot_t` and `kcg_lsl_t` macros

If `-wrap_c_ops` option is set:

- the following macros are defined for all predefined numeric types:

`kcg_diff_t`	`kcg_gt_t`	`kcg_mult_t`
`kcg_div_t`	`kcg_le_t`	`kcg_plus_t`
`kcg_eq_t`	`kcg_lt_t`	`kcg_uminus_t`
`kcg_ge_t`	`kcg_minus_t`	

- the `kcg_mod_t` macro for all predefined integer types (intN, uintN)
- the `kcg_land_t`, `kcg_lor_t`, `kcg_lxor_t`, and `kcg_lsr_t` macros for all predefined unsigned types (uintN)
- the `kcg_t1_to_t2` macros where `t1` and `t2` are distinct predefined numeric types
- the `kcg_cond`, `kcg_plus_size`, `kcg_lit_size`, and `kcg_le_size` macros for any sharp operator
- the `kcg_plus_size`, `kcg_minus_size`, `kcg_uminus_size`, `kcg_mod_size`, `kcg_eq_size`, `kcg_lt_size`, `kcg_le_size`, and `kcg_gt_size` macros for the size type

- all comparison operators in `kcg_eq`, `kcg_diff`, `kcg_eq_bool`, `kcg_diff_bool`, `kcg_eq_char`, and `kcg_diff_char` macros
- all Boolean operators in `kcg_and`, `kcg_or`, `kcg_xor`, and `kcg_not` macros
- other macros: `kcg_cond` to convert a `false` value of type `kcg_bool` to 0 and a `true` value of type `kcg_bool` to 1, `kcg_index(a, i)` to return the value at index `i` (of type `kcg_size`) in the array `a`, and `kcg_incr` to take as input a variable of type `kcg_size` and increment it by one if the result fits in the range of the `kcg_size` type

9 Definition of Boolean predefined values (`kcg_true` and `kcg_false`) in the following format:

if no -wrap_c_ops option:

```
#ifndef kcg_false
#define kcg_false ( ( kcg_bool ) 0)
#endif /* kcg_false */

#ifndef kcg_true
#define kcg_true ( ( kcg_bool ) 1)
#endif /* kcg_true */
```

if -wrap_c_ops option:

```
#ifndef kcg_false
#define kcg_false ( ( kcg_bool ) kcg_int_lit ( 0 ) )
#endif /* kcg_false */

#ifndef kcg_true
#define kcg_true ( ( kcg_bool ) kcg_int_lit ( 1 ) )
#endif /* kcg_true */
```

10 Assignment macros: protected inclusion of `kcg_assign.h`, and definitions of `kcg_assign_struct` and `kcg_assign_array` macros

Warning

Please note that a definition of `kcg_assign` must be provided by users either by referencing their own `kcg_assign.h` file or by providing their own definition of the macro through the `-user_config` option as explained further down.

11 Two enumerated type definitions for each state machine in the cone of influence of the root operator (one for collecting all states and one for collecting all transitions)

12 Enumerated type for each **enum** type present in the input model

13 Types declarations, macro definitions, and function declarations for array and structure copies and comparisons in topological order:

- if the model contains structure type definitions, `kcg_types.h` contains for each structure type equivalence class:

- the definition of C structure type named `structname`
- the macro definition of a copy function named `kcg_copy_structname`
- the prototype of a comparison function named `kcg_comp_structname`
- if the comparison function is not used in the generated code, the prototype is surrounded by a `#ifdef kcg_use_structname` directive.

- if the model contains array type definitions, `kcg_types.h` contains for each array type equivalence class:

- the definition of C array type named `arrayname`
- the macro definition of a copy function named `kcg_copy_arrayname`
- the prototype of a comparison function named `kcg_comp_arrayname`
- if the comparison function is not used in the generated code, the prototype is surrounded by a `#ifdef kcg_use_arrayname` directive.

- the definitions of the C types corresponding to non-enumeration user-defined types without `manifest` pragma

14 Footer (see "File Footer Banner" on page 232)

Check the description of a `kcg_types.h` generated header file hereafter.

Example: Find below the `kcg_types.h` header file generated for the RollControl model

```
/* $************** KCG Version 6.4 beta (build i6) **************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-17T09:34:37
*************************************************************$ */
#ifndef _KCG_TYPES_H_
#define _KCG_TYPES_H_

#define KCG_MAPW_CPY

#include "./user_macros.h"

#ifndef kcg_int
#define kcg_int kcg_int
typedef int kcg_int;
#endif /* kcg_int */

#ifndef kcg_bool
#define kcg_bool kcg_bool
typedef unsigned char kcg_bool;
#endif /* kcg_bool */

#ifndef kcg_real
#define kcg_real kcg_real
typedef double kcg_real;
#endif /* kcg_real */

#ifndef kcg_char
#define kcg_char kcg_char
typedef char kcg_char;
#endif /* kcg_char */

#ifndef kcg_false
#define kcg_false ((kcg_bool) 0)
#endif /* kcg_false */

#ifndef kcg_true
#define kcg_true ((kcg_bool) 1)
#endif /* kcg_true */

#ifndef kcg_assign
#include "kcg_assign.h"
#endif /* kcg_assign */

#ifndef kcg_assign_struct
#define kcg_assign_struct kcg_assign
#endif /* kcg_assign_struct */

#ifndef kcg_assign_array
#define kcg_assign_array kcg_assign
#endif /* kcg_assign_array */
```

1. Trace header
2. Header files
3. Empty definition of MAPW_CPY
4. Definition of predefined types
5. Definition of Boolean constants
5. Definition of assignment macros

(continued on next page)

```
/* truthtables::TruthTableValues */
typedef enum kcg_tag_TruthTableValues_truthtables {
  T_truthtables,
  F_truthtables,
  X_truthtables
} TruthTableValues_truthtables;
/* RollMode::TRollMode */
typedef enum kcg_tag_TRollMode_RollMode {
  off_RollMode,
  nominal_RollMode,
  failsoft_RollMode
} TRollMode_RollMode;
/* RollMode::RollMode::SMRollMode */
typedef enum kcg_tag_SSM_TR_SMRollMode {
  SSM_TR_no_trans_SMRollMode,
  SSM_TR_Off_1_SMRollMode,
  SSM_TR_On_1_SMRollMode
} SSM_TR_SMRollMode;
/* RollMode::RollMode::SMRollMode */
typedef enum kcg_tag_SSM_ST_SMRollMode {
  SSM_st_Off_SMRollMode,
  SSM_st_On_SMRollMode
} SSM_ST_SMRollMode;
/* RollMode::RollMode::SMRollMode::On::SMOn */
typedef enum kcg_tag_SSM_TR_SMOn_SMRollMode_On {
  SSM_TR_no_trans_SMOn_SMRollMode_On,
  SSM_TR_Nominal_1_SMOn_SMRollMode_On,
  SSM_TR_Failsoft_1_SMOn_SMRollMode_On
} SSM_TR_SMOn_SMRollMode_On;
/* RollMode::RollMode::SMRollMode::On::SMOn */
typedef enum kcg_tag_SSM_ST_SMOn_SMRollMode_On {
  SSM_st_Nominal_SMRollMode_On_SMOn,
  SSM_st_Failsoft_SMRollMode_On_SMOn
} SSM_ST_SMOn_SMRollMode_On;
/* RollRate::TRealLeftRight */
typedef struct kcg_tag_TRealLeftRight_RollRate {
  kcg_real left;
  kcg_real right;
} TRealLeftRight_RollRate;

/* lut::LutIndex */
typedef struct kcg_tag_LutIndex_lut { kcg_int k; kcg_real f; } LutIndex_lut;

#ifndef kcg_copy_LutIndex_lut
#define kcg_copy_LutIndex_lut(kcg_C1, kcg_C2) (kcg_assign_struct((kcg_C1),
(kcg_C2), sizeof (LutIndex_lut)))
#endif /* kcg_copy_LutIndex_lut */

#ifndef kcg_copy_TRealLeftRight_RollRate
#define kcg_copy_TRealLeftRight_RollRate(kcg_C1, kcg_C2)
(kcg_assign_struct((kcg_C1), (kcg_C2), sizeof (TRealLeftRight_RollRate)))
#endif /* kcg_copy_TRealLeftRight_RollRate */

#ifdef kcg_use_LutIndex_lut
#ifndef kcg_comp_LutIndex_lut
extern kcg_bool kcg_comp_LutIndex_lut(
  LutIndex_lut *kcg_c1,
  LutIndex_lut *kcg_c2);
#endif /* kcg_comp_LutIndex_lut */
#endif /* kcg_use_LutIndex_lut */

#ifdef kcg_use_TRealLeftRight_RollRate
#ifndef kcg_comp_TRealLeftRight_RollRate
extern kcg_bool kcg_comp_TRealLeftRight_RollRate(
  TRealLeftRight_RollRate *kcg_c1,
  TRealLeftRight_RollRate *kcg_c2);
#endif /* kcg_comp_TRealLeftRight_RollRate */
#endif /* kcg_use_TRealLeftRight_RollRate */

#endif /* _KCG_TYPES_H_ */
/* $************** KCG Version 6.4 beta (build i6) **************
** kcg_types.h
** Generation date: 2013-01-17T11:26:41
*************************************************************$ */
```

Definition of enumerated types (including SSM States and Transitions)

Definition of structured types, and copy/compare functions

6. Trace footer

KCG_TYPES.C

KCG/MCG generate the `kcg_types.c` source file including the `kcg_types.h` header file. The `kcg_types.c` source file contains:

- the definition for each structure comparison function declared in `kcg_types.h`
- the definition for each array comparison function declared in `kcg_types.h`

In addition, for each array class, the `kcg_types.c` file also contains a definition of the `kcg_comp_<class_name>` function.

Example: Find below a `kcg_types.c` generated source file

```
/* $************** KCG Version 6.4 beta (build i6) **************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-17T11:26:41
*************************************************************$ */

#include "kcg_types.h"

#ifdef kcg_use_LutIndex_lut
kcg_bool kcg_comp_LutIndex_lut(LutIndex_lut *kcg_c1, LutIndex_lut *kcg_c2)
{
  kcg_bool kcg_equ;

  kcg_equ = kcg_true;
  kcg_equ = kcg_equ & (kcg_c1->f == kcg_c2->f);
  kcg_equ = kcg_equ & (kcg_c1->k == kcg_c2->k);
  return kcg_equ;
}
#endif /* kcg_use_LutIndex_lut */

#ifdef kcg_use_TRealLeftRight_RollRate
kcg_bool kcg_comp_TRealLeftRight_RollRate(
  TRealLeftRight_RollRate *kcg_c1,
  TRealLeftRight_RollRate *kcg_c2)
{
  kcg_bool kcg_equ;

  kcg_equ = kcg_true;
  kcg_equ = kcg_equ & (kcg_c1->right == kcg_c2->right);
  kcg_equ = kcg_equ & (kcg_c1->left == kcg_c2->left);
  return kcg_equ;
}
#endif /* kcg_use_TRealLeftRight_RollRate */

/* $************** KCG Version 6.4 beta (build i6) **************
** kcg_types.c
** Generation date: 2013-01-17T11:26:41
*************************************************************$ */
```

KCG_CONSTS.H

KCG/MCG generate a `kcg_consts.h` header file containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file

3 Inclusion of `kcg_types.h` header file

4 Definition for each non-imported constant declared in the model:

- if the constant has no `const` pragma and its type is a scalar type:

```
#define <const_name> <C_expr>
```

- if the constant has a `const` pragma and its type is not a scalar type:

```
extern const <C-type> <const_name>;
```

where `<C-type>` is a type identifier defined in `kcg_types.h`.

5 Footer (see "File Footer Banner" on page 232)

Note

For imported constants definitions in `kcg_const.h` file, see "Description of External Files for C Code" on page 221.

Check the description of a generated `kcg_consts.h` header file hereafter.

Example: Find below the `kcg_consts.h` header file generated for the RollControl model

```
/* $*************** KCG Version 6.1.2 (build i5) ***************
** Command: s2c612 -config
D:/examples/GettingStarted/final/RollControl/KCG\kcg_s2c_config.txt
** Generation date: 2010-10-11T16:06:48
*****************************************************************$ */
#ifndef _KCG_CONSTS_H_
#define _KCG_CONSTS_H_

#include "kcg_types.h"

/* RollRate::kRollRateWarning */
extern const TRealLeftRight_RollRate kRollRateWarning_RollRate;

/* RollMode::kFailSoftRoll */
#define kFailSoftRoll_RollMode 20.0

#endif /* _KCG_CONSTS_H_ */
/* $*************** KCG Version 6.1.2 (build i5) ***************
** kcg_consts.h
** Generation date: 2010-10-11T16:06:48
*****************************************************************$ */
```

1. Trace header
2. Directives
3. Included kcg_types.h
4. Definition of constants
5. Trace footer

KCG_CONSTS.C

KCG/MCG generate a `kcg_consts.c` source file containing:

1. Header (see "File Header Banner" on page 231)
2. Inclusion of kcg_consts.h header file
3. For each non-imported constant declared in `kcg_consts.h` file the following definition:

```
const <C-type> <const_name> = <expr>
```

where `<C-type>` is a type identifier defined in `kcg_types.h` and `<expr>` the C translation of the Scade expression.

4. Footer (see "File Footer Banner" on page 232)

Check the description of a `kcg_consts.c` generated source file hereafter.

Example: Find below the `kcg_consts.c` source file generated for the RollControl model.

```
/* $*************** KCG Version 6.1.2 (build i5) ***************
** Command: s2c612 -config
D:/examples/GettingStarted/final/RollControl/KCG\kcg_s2c_config.txt
** Generation date: 2010-10-11T16:06:48
*************************************************************$ */

#include "kcg_consts.h"

/* RollRate::kRollRateWarning */
const TRealLeftRight_RollRate kRollRateWarning_RollRate = { - 15.0, 15.0 };

/* $*************** KCG Version 6.1.2 (build i5) ***************
** kcg_consts.c
** Generation date: 2010-10-11T16:06:48
*************************************************************$ */
```

1. Trace header
2. Included kcg_types.h
3. Global constant definition
4. Trace footer

KCG_SENSORS.H

KCG/MCG generate a `kcg_sensors.h` header file containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file
3. Inclusion of `kcg_types.h` header file

4 For each declared sensor in input model, a C global variable definition with this pattern:

```
#ifndef <sensor_name>
extern <C-type> <sensor_name>;
#endif
```

where <C-type> is a type identifier defined in kcg_types.h.

5 Footer (see "File Footer Banner" on page 232)

Additional Generated Files in C

- "kcg_imported_functions.h"
- "kcg_init.h"
- "<node_name>.dh"
- "<node_name>.dc"
- "kcg_imported_types.dh"

KCG_IMPORTED_FUNCTIONS.H

If the cone of influence of the root operator contains an imported operator declared as a function, KCG/MCG generate a kcg_imported_functions.h containing:

1 Header (see "File Header Banner" on page 231)

2 Directives for file inclusion protection at start/end of file

3 Inclusion directive for kcg_types.h header file

4 Prototype of a cycle function with one of the following patterns:

- For each Scade function returning a single scalar value when there are some inputs

```
#ifndef <operator_name>
extern <output_type> <operator_name>(<inputs>);
#endif /* <operator_name> */
```

- For each Scade function returning a single scalar value when there are no inputs

```
#ifndef <operator_name>
extern <output_type> <operator_name>(void);
#endif /* <operator_name> */
```

- For any other Scade function

```
#ifndef <operator_name>
extern void <operator_name>(<inputs>, <outputs>);
#endif /* <operator_name> */
```

where `<inputs>` is the list of function inputs along with their type (passed by values when they belong to a scalar type and by address otherwise) and `<outputs>` is the list of pointers to function outputs along with their type. This includes the case where the called function has no output.

5 Footer (see "File Footer Banner" on page 232)

Check the description of a `kcg_imported_functions.h` generated header file hereafter.

Example: Consider the Pilot example including a `Rising_FL_imported` function. Find below the generated `kcg_imported_functions.h` header file.

```
/* $*************** KCG Version 6.1.2 (build i5) ***************
** Command: s2c612 -config D:/examples/Pilot/Embedded\kcg_s2c_config.txt
** Generation date: 2010-10-13T18:32:01
*************************************************************$ */
#ifndef _KCG_IMPORTED_FUNCTIONS_H_
#define _KCG_IMPORTED_FUNCTIONS_H_

#include "kcg_types.h"

#ifndef SqrtR_mathext
/* mathext::SqrtR */
extern kcg_real SqrtR_mathext(/* mathext::SqrtR::Input1 */kcg_real Input1);
#endif /* SqrtR_mathext */

#ifndef Rising_FL_imported_Pilot
/* Pilot::Rising_FL_imported */
extern void Rising_FL_imported_Pilot(
  /* Pilot::Rising_FL_imported::VelIn */tVelocity_Pilot *VelIn,
  /* Pilot::Rising_FL_imported::VelOut */tVelocity_Pilot *VelOut);
#endif /* Rising_FL_imported_Pilot */

#endif /* _KCG_IMPORTED_FUNCTIONS_H_ */
/* $*************** KCG Version 6.1.2 (build i5) ***************
** kcg_imported_functions.h
** Generation date: 2010-10-13T18:32:01
*************************************************************$ */
```

1. Trace header
2. Included kcg_types.h
3. Definition of prototype
4. Trace footer

KCG_INIT.H

If the cone of influence of the root operator contains at least one imported operator declared as node, SCADE Suite KCG generates a `kcg_init.h` containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file
3. Preprocessor directive as follows:

```
#define <operator_name>_init <operator_name>_reset
```

4. Footer (see "File Footer Banner" on page 232)

Note

If the reset function of all imported nodes is also a valid initialization function, the declaration and definition of such initialization function can be replaced by the usage of the generated `kcg_init.h` file as an argument of **`-user_config`** option.

<NODE_NAME>.DH

KCG/MCG generate, for each monomorphic instance of imported node in the cone of influence of the root operator, a `<node_name>.dh` template file containing:

1. Header (see "File Header Banner" on page 231)
2. Directives for file inclusion protection at start/end of file
3. Inclusion of `kcg_types.h` header file
4. Empty "Context type" structure named `outC_<node_name>`
5. If `-state_vector` option is set:
   - A state vector type definition named `SV_<node name>`

- Two functions for saving/loading a context into/from a state vector:

```
extern void kcg_save_SV_<operator_name>(SV_<operator_name> *SV, outC_<operator_name> *outC);
extern void kcg_load_SV_<operator_name>(outC_<operator_name> *outC, SV_<operator_name> *SV);
```

6 Prototype of initialization function

7 Prototype of reset function

8 Prototype of cycle function

9 Footer (see "File Footer Banner" on page 232)

Note

Once completed by the user, this template file must be suffixed as `.h` instead of `.dh` to be integrated in the whole generated code.

<NODE_NAME>.DC

KCG/MCG generate, for each monomorphic instance of imported node in the cone of influence of the root operator, a `<node_name>.dc` template file containing:

1 Header (see "File Header Banner" on page 231)

2 Inclusion of header files `kcg_consts.h`, `kcg_sensors.h`, and `<node_name>.h`

3 Empty definitions of initialization, reset and cycle functions

4 If `-state_vector` option is set, empty definitions of the save and load functions

5 Footer (see "File Footer Banner" on page 232)

Note

Once completed by the user, this template file must be suffixed as `.c` instead of `.dc` to be integrated in the whole generated code.

KCG_IMPORTED_TYPES.DH

When the model contains at least one imported type, SCADE Suite KCG generates a `kcg_imported_types.dh` template file containing:

1. Empty definition of the type (supplied by user)
2. Prototype of C type initializer used to initialize value of imported type:
   - If no `C:initializer` pragma attached to type declaration

```
extern void <type_name>_init (type_name *);
```

   - If name set as argument of `C:initializer` pragma attached to type declaration

```
extern void <pragma_argument> (pragma_argument *);
```

3. For non-scalar types, the declarations of the associated copy and comparison functions
4. For numeric types: template definition of `kcg_lit_t` macro
5. For unsigned types: template definition of `kcg_lnot_t` and `kcg_lsl_t`
6. If `-wrap_c_ops` option is set, template definitions of macros listed below:

When type is ...	provided macros are
Scalar	`kcg_eq_t` and `kcg_diff_t`
Numeric	`kcg_ge_t`, `kcg_gt_t`, `kcg_le_t`, `kcg_lt_t`, `kcg_plus_t`, `kcg_minus_t`, `kcg_mult_t`, `kcg_div_t`, and `kcg_uminus_t`
	`kcg_u_of_t` for all predefined numeric types u so that a cast from type `t` to type u appears in generated code
	`kcg_u_to_t` for all numeric types u (either predefined or imported) so that a cast from type u to type `t` appears in generated code
Integer	`kcg_mod_t`
Unsigned	`kcg_land_t`, `kcg_lor_t`, `kcg_lxor_t`, and `kcg_lsr_t`

File Inclusion Protection

All header files generated by SCADE Suite KCG contain a file inclusion protection made of preprocessor directives at the beginning and at the end of the file:

- File starting with:

```
#ifndef _BASENAME_H_
#define _BASENAME_H_
```

- File ending with:

```
#endif /*_BASENAME_H_ */
```

If `-globals_prefix` option is set, the name of the macros used for protection against multiple inclusions is prefixed and uppercased as follows for `kcg_types.h`, `kcg_consts.h`, `kcg_sensors.h`, `kcg_imported_functions.h`, and `kcg_init.h` files:

- File starting with:

```
#ifndef _<prefix>_KCG_TYPES_H_
#define _<prefix>_KCG_TYPES_H_
```

- File ending with:

```
#endif /*_<prefix>_KCG_TYPES_H_ */
```

For other files, BASENAME is the basename of the file.

Description of External Files for C Code

Imported code used in SCADE Suite models must be provided by users. Specific files and formats are expected for proper integration of imported types and constants in C generated code.

KCG_IMPORTED_TYPES.H

The `kcg_imported_types.h` header file must contain the following:

1. Header (see "File Header Banner" on page 231)
2. C definition of each imported type used in the model
3. Prototype or macro definition of an initialization function named for each imported type (directly or indirectly used in the context declaration):
   - `<type_name>_init` if no initializer pragma on declaration of corresponding Scade type
   - according to name given by initializer pragma
4. For each non-scalar imported type:
   - prototype or macro definition of a copy function: `kcg_copy_<C_type_name>`
   - prototype or macro definition of a comparison function: `kcg_comp_<C_type_name>`
5. Footer (see "File Footer Banner" on page 232)

KCG_CONSTS.H

The kcg_consts.h header file can also contain for each imported constant declaration in the model an empty definition with the following pattern:

- if the constant has `const` pragma or its type is not scalar

```
extern const <C-type> <const_name>;
```

- in other cases: `<C-type>` is a type identifier defined in `kcg_types.h`

```
#ifndef <const_name>
#define <const_name> <const_name>
extern const <C-type> <const_name>;
#endif
```

Description of Generated Files for Ada Code

The structure and content of generated files for Ada code is described below:

- "Standard Generated Files in Ada"
- "Additional Generated Files in Ada"

Note

All generated files are SPARK 95 compliant and suitable for analysis with SPARK toolset version 10.1 or above. Supported SPARK annotations are those of SPARK Examiner of SPARK Pro 10.0.2 release.

SCADE SUITE MODELS AND SPARK COMPLIANCE

The prerequisites for model compliance with SPARK are the following:

- No use of # operator
- No use of conversion from float32/float64 to integer
- constants declarations using lsl, lsr, and lnot operators are statically evaluable in the domain constrained by the dynamic semantics of arithmetic operators
- No use of unary operators with unsigned operands

The figure below presents the list of generated Ada files as they appear in the Output Window of SCADE Suite after successful code generation (see "Launching KCG Code Generation from UI" on page 855 in *SCADE Suite User Manual*):

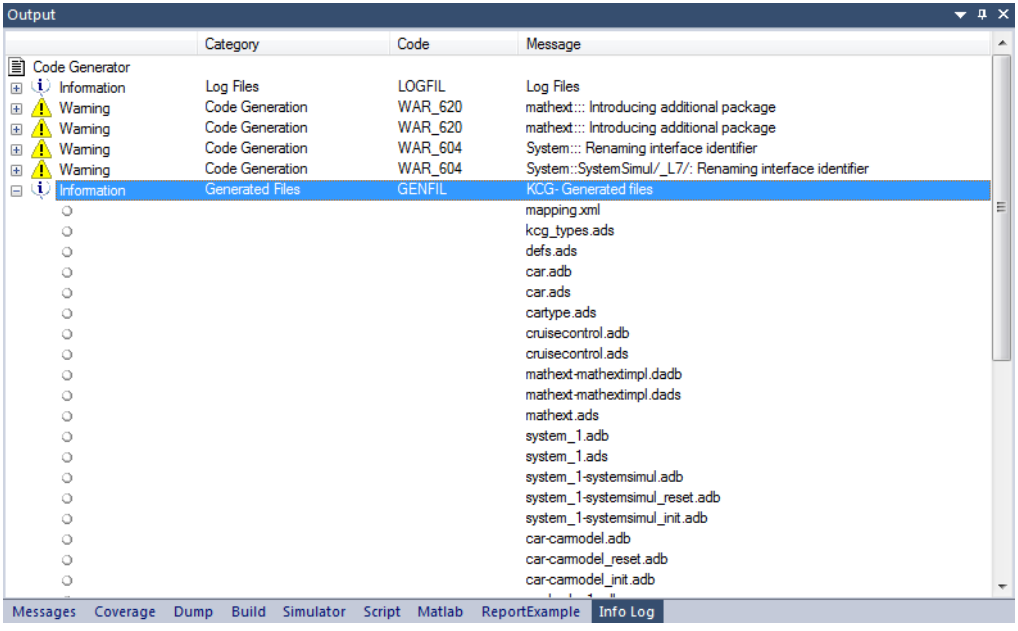


Figure 9.5: List of KCG-generated Ada files (Output Window in SCADE Suite)

Standard Generated Files in Ada

All standard files generated by KCG for Ada code integration are explained in details below. For each file you can find a description of their content with some illustrations and examples.

- "kcg_types.ads"
- "<package_path>.ads"
- "<package_path>.adb"

Note

Each package in the call graph of a root operator is translated into a package in the generated Ada code (and SCADE Suite sub-packages as child packages in Ada).

CONTENT OF ADA PARENT PACKAGE

The Ada parent package contains the definition of the Ada counterparts for Scade declarations (operator, constant, type, sensor or package).

An operator requires:

- *Ada cycle subprogram* (Ada conterpart of cycle C function)
- and, if the operator requires a context, the following items:
  - *Ada context* defined by an Ada record containing context variables
  - *Ada initialization procedure*
  - *Ada reset procedure*: generated if and only if the context contains memories initialized with a static value.

INITIALIZATION PROCEDURE

The Ada initialization procedure generated for any non-expanded Scade operator initializes each field of the context based on the actual type of the operator as summarized below:

Table 9.7: Initial value set by Ada init procedure for non-expanded operators

Type	Initial Value
bool	true
char	' '
intN, uintN, size	0
floatN	0.0
enumeration	first value in the **`enum`** declaration
imported type T	reference to the Ada constant of **`imported`** type or to nullary function returning a value of **`imported`** type
array	array aggregate with (the same) matching initial value for all elements
record	record aggregate with a matching initial value for each record field

KCG_TYPES.ADS

KCG generates all type declarations of the SCADE Suite model into `kcg_types.ads`. For each type declaration in the model (modulo type equivalence), a subtype declaration in Ada is generated in `Kcg_Types` package. All base types (including enumerations, arrays, and records) as well as the user-defined manifest types are declared in `Kcg_Types`. All other (non-manifest) user-defined types are declared as subtypes in their respective parent package.

<PACKAGE_PATH>.ADS

KCG generates the Ada specification for non-imported elements of the input model into ADS files (`.ads` extension). File naming is done with respect to the package hierarchy mechanism (see "Package Hierarchy and SPARK Compliance" on page 234). KCG translates non-imported elements as follows:

- a constant into a constant in Ada code
- a named type as a subtype of a (sub)type defined in `Kcg_Types` package
- an Ada function if the SCADE function is not the root operator and returns a single output; otherwise an Ada procedure
- a node: init/reset/cycle procedures + context type

<PACKAGE_PATH>.ADB

KCG generates the Ada implementation (body) for non-imported operators (nodes or functions) of the input model into ADB files (`.adb` extension). File naming is done with respect to the package hierarchy mechanism (see "Package Hierarchy and SPARK Compliance" on page 234). The cycle, reset and init procedures are generated as separate compilation units.

KCG translates non-expanded operators as follows:

Cycle procedure for the Ada cycle subprogram

The procedure implements the behavior of the Scade operator by processing inputs and using values of the context (if any) for storing its state to produce the outputs.

- If a non-root operator has only one output and does not require a context, an Ada function whose returned value is the operator output and whose formal parameters are:
  - one **in** parameter for each input of the Scade operator in order from the input model
- Otherwise, an Ada procedure whose formal parameters are in order:
  - one **in** parameter for each input of the Scade operator in order from the input model
  - if the operator is marked for separate I/O, one **out** parameter for each output of the Scade operator in order from the input model
  - if the operator requires a context, an Ada *context record* parameter with **out** mode value (context containing neither memories, nor clocked observable variables) or with **in out** mode value (context containing memories or clocked observable variables)

If the context is empty, the Ada cycle subprogram contains no context parameter.

Init procedure (optional) for the Ada initialization procedure

- An Ada procedure whose formal parameter is an Ada *context record* parameter with **out** mode value. Initializes the fields of the context with such parameter.

Reset procedure (optional) for the Ada reset procedure

- An Ada procedure whose formal parameter is an Ada *context record* with **out** or **in out** mode value. The context parameter is declared with **in out** mode if at least one field of the context is not updated by the reset procedure. Sets to **true** all reset flags and indicates the operator is in initial state.

Ada context record type (optional)

- Ada record containing the context variables:
  - reset flags used to compile **init** operators (either explicitly used in the model or implied by higher-level constructions)
  - memories used to compile **pre** and **fby** operators (explicitly used or implied)
  - probes when `-probes` option is set
  - fields corresponding to the *context* records of sub-operators

Additional Generated Files in Ada

SCADE Suite KCG generates templates for the Ada counterpart of imported declarations and sensors in dedicated files.

- "<package_path>.dads"
- "<package_path>.dadb"

<PACKAGE_PATH>.DADS

KCG generates the *Ada specification* for *imported elements* of the input model into template DADS files (`.dads` extension). File naming is done with respect to the package hierarchy mechanism (see "Package Hierarchy and SPARK Compliance" on page 234).

KCG translates imported elements as follows:

- a sensor into a global variable in Ada code
- Scade imported functions into an Ada counterpart defined by Ada cycle subprogram
- Scade imported node into an Ada counterpart defined by Ada context, cycle subprogram, initialization procedure, and reset procedure

<PACKAGE_PATH>.DADB

KCG generates the *Ada implementation* for *imported operators (nodes or functions)* of the input model into template DADB files (`.dadb` extension). File naming is done with respect to the package hierarchy mechanism (see "Package Hierarchy and SPARK Compliance" on page 234). The cycle, reset and init procedures are generated as separate compilation units.

KCG translates imported nodes as follows:

- The interface of an imported node in the Ada code is the same as for a non-imported Scade node having the same signature.

- By default, KCG assumes that the Ada implementation of an imported node makes use of both reset flags and memories, but contains no probes.

KCG translates imported functions as follows:

- The interface of an imported function in the Ada code is the same as for a non-imported Scade function having the same signature.
  - By default, KCG assumes that the Ada implementation of an imported function contains no probes.

Description of External Files for Ada Code

Specific files may be necessary for providing user definitions with Ada generated code.

KCG_CONFIG.ADS AND KCG_CONFIG.ADB

KCG does not generate these files.

The `Kcg_Config` configuration package is the top-level package when `-root_package` option is not used or the direct child of the package specified when using the option.

Users must provide the following procedures when the `-keep_asserts` option is used:

```
procedure Kcg_Assume (A: Boolean);
procedure Kcg_Guarantee (G: Boolean);
```

These procedures are called by KCG-generated code with the Boolean values for each Assume/Guarantee operator present in the SCADE Suite input model.

Traceability from Specification Models

KCG/MCG facilitate traceability between the generated C code and the various variables or constructs in the model with information added in comments, headers, and footers of the generated .c and .h files. In addition, KCG/MCG generate an XML mapping file.

- "Traceability Information"
- "Mapping File"
- "Package Hierarchy and SPARK Compliance" (only for Ada code)

Note

Generated code is considered traceable if any user is able to link it to the input model without knowing about the KCG/MCG code generation process.

Traceability Information

The traceability of the generated code facilitates the understanding of the correspondence between any variable or construct in models and the generated code. Traceability is implemented through the following items in generated files:

- "Trace Comment"
- "File Footer Banner"
- "File Header Banner"
- "Naming for Ada Counterparts"

TRACE COMMENT

For each C/Ada element being the counterpart of one element of the model, KCG/MCG provide a trace explaining the origin of the C/Ada variable. The trace is a fully qualified path added as comment next to the declaration.

Starting from the global package, the path indicates:

- the operator names
- in each operator, the automaton names
- in each automaton, the state names

The separator between names is "**::**".

A comment is added in the generated code before each call to a cycle function. It contains the occurrence pragma of the corresponding instance if any, or the *fully qualified path* of the called operator.

Information provided in doc pragma is kept as comments in the generated code when possible. If the pragma is attached to an operator instance expanded in the generated code, the information cannot be kept as comment. If the pragma is attached to a Scade declaration of the input model, the string is added as a C comment to the corresponding entity in the generated code (only if the entity is in the C code).

Example 1: Sample of trace (bold) in C code for a memorized output from CruiseSpeed located in CruiseControl of CruiseControl package.

```
tSpeed_CarType /* CruiseControl::CruiseControl::CruiseSpeed */ CruiseSpeed;
```

Example 2: Sample of trace (in bold) in C code for a type associated with the states of automaton SSM2 located in state Enabled of automaton SM1 in CruiseControl of CruiseControl package.

```
/* CruiseControl::CruiseControl::SM1::Enabled::SM2 */
typedef enum {
  SSM_st_Active_SM1_Enabled_SM2,
  SSM_st_Interrupt_SM1_Enabled_SM2
} SSM_ST_SM2_SM1_Enabled;
```

Example 3: Sample of trace comment in Ada code.

```
-- CruiseControl::CruiseControl
  procedure CruiseControl_1(
    -- CruiseControl::CruiseControl::On
    On : in Boolean;
    -- CruiseControl::CruiseControl::Off
    Off : in Boolean;
    ...
    -- CruiseControl::CruiseControl::ThrottleCmd
    ThrottleCmd : out CarType.tPercent;
    -- CruiseControl::CruiseControl::CruiseState
    CruiseState : out Kcg_Types.tCruiseState_49;
    Ctx : in out Context_CruiseControl_1)
  is
    -- CruiseControl::CruiseControl::SM1::Enabled::SM2::Active::SM3
    SM3_state_sel : Kcg_Types.SSM_ST_SM3_620;
    -- CruiseControl::CruiseControl::SM1::Enabled::SM2::Active::SM3
    SM3_state_act : Kcg_Types.SSM_ST_SM3_620;
  ...
    SM1_state_act : Kcg_Types.SSM_ST_SM1_582;
    -- CruiseControl::CruiseControl::SM1::Enabled::SM2::Active::SM3::On::_L3
    L3 : CarType.tPercent;
    -- CruiseControl::CruiseControl::SM1::Enabled::SM2::Active::SM3
    SM3_reset_act : Boolean;
```

FILE HEADER BANNER

KCG/MCG insert a header banner at the beginning of each generated file (except in `kcg_files.txt`). The file header banner contains:

- SCADE Suite KCG version used to generate C/Ada code or MCG version used to generate C code
- Each line of Ada file header starts with -- (KCG only)
- The command line
- The date and time of code generation in `YYYY-MM-DDTHH:MM:SS` format. If `-no_timestamp` option is set, no generation date is added to the header banner.

Example: Sample of trace header banner (no use of `-no_timestamp`)

```
/* $*************** KCG Version 6.4 (build i6) ****************
** Command: kcg64.exe -config C:/Dev Projects/SCADE Suite/RollControl/KCG/config.txt
** Generation date: 2013-01-14T11:50:42
*************************************************************$ */
```

FILE FOOTER BANNER

KCG/MCG insert a footer banner at the end of each generated file (except in `kcg_files.txt`). The file footer banner contains:

- SCADE Suite KCG version used to generate C/Ada code or MCG version used to generate C code
- Each line of Ada file footer starts with `--` (KCG only)
- Name of generated file or name of file specified by `-xml_basename`
- The date and time of code generation in `YYYY-MM-DDTHH:MM:SS` format. If `-no_timestamp` option is set, no generation date is added to the header banner.

Example: Sample of trace footer banner (no use of `-no_timestamp`)

```
/* $*************** KCG Version 6.4 (build i6) ***************
** RollMode_RollMode.h
** Generation date: 2013-01-14T11:50:42
************************************************************$ */
```

NAMING FOR ADA COUNTERPARTS

In the usual cases, SCADE Suite KCG names the Ada counterparts of Scade declarations with the identifiers declared in the SCADE Suite input model. This is impossible when:

- Scade identifiers conflict with SPARK reserved keywords
- Scade identifiers conflict with each other
- Scade identifiers conflict with identifiers from the “Standard library”
- Scade identifiers are case-sensitive while Ada identifiers are not case-sensitive
- Scade identifiers are not valid identifiers in Ada

For resolving conflicts, SCADE Suite KCG renames Scade declarations in Ada with discriminating numbers, prefixes, or suffixes based on the following patterns:

- If the Scade declaration has no `Ada:name` pragma attached

```
<prefix><d><decl_name><suffix>
```

  `<d>` is an empty string or a strictly positive integer. `<decl_name>` is the string of the Scade declaration truncated to fit maximum name length. All consecutive sequences of underscores are replaced by single underscore. All leading and trailing underscores are also removed.

- If the Scade declaration has an `Ada:name` pragma attached

```
<prefix><name_pragma><suffix>
```

  `<name_pragma>` is the value specified by `Ada:name` pragma

Ada identifiers are renamed with the following prefixes and suffixes:

Ada Element	**Ada Name**
Ada context	Operator name with `Context` as prefix
Ada cycle subprogram	Operator name
Ada initialization procedure	Operator name with `Init` as suffix
Ada reset procedure	Operator name with `Reset` as suffix
Ada type initializer	Type name with `Init` as suffix if no `Ada:initializer` pragma on type or type name as specified by `Ada:initializer` pragma
Ada counterpart of constants	Constant name
Ada counterpart of types	Type name
Ada counterpart of sensors	Sensor name
Ada counterpart of I/Os	I/O name

Whenever renaming occurs on interface identifiers, SCADE Suite KCG raises a warning.

Package Hierarchy and SPARK Compliance

When possible, SCADE Suite KCG provides direct correspondence between the package hierarchy of the SCADE Suite input model and the package hierarchy of the generated Ada code. Due to differences between Scade language and SPARK, this is not always possible. In identified cases, additional packages ensure the generated code remains SPARK-compliant. A warning message systematically informs users when this occurs.

- "Direct Package Hierarchy Correspondence"
- "Case of Top-Level Declaration in SCADE Suite Model"
- "Case of Package with Imported and Non-Imported Declarations"
- "Case of References to Sub-Packages"
- "Case of Circular Dependencies between Packages"

DIRECT PACKAGE HIERARCHY CORRESPONDENCE

SCADE Suite KCG objective is to provide a direct correspondence between the package hierarchy of the SCADE Suite input model and that of the generated Ada code. In some well-identified cases, SCADE Suite KCG has to introduce additional packages to ensure that the generated code is valid SPARK. Every time a package needs to be introduced by SCADE Suite KCG to generate valid SPARK code, a warning message displays that information.

The package hierarchy of the SCADE Suite model has a direct counterpart in the generated code when the model fulfills all of the following properties:

- each Scade declaration belongs to a SCADE Suite package
- a package with imported declarations (and sensors) does not contain non-imported declarations
- no package contains a global declaration that depends on one of its sub-packages (referring to a sibling package is allowed)
- there are no circular dependencies between packages

When the input model fulfills all these properties, no additional packages need to be introduced to obtain valid SPARK code. In compliance with SPARK, SCADE Suite KCG adds an `--#inherit` annotation with every Ada child package specification it generates. In addition, SCADE Suite KCG provides the ability to specify which parts of the model are out of Spark Examiner scope:

- If a Scade package has a `spark hide` pragma, the `--# hide P;` directive is added on top of the declarative part of the corresponding Ada package body.
- If a Scade operator has a `spark hide` pragma, the `--# hide op;` directive is added on top of the declarative part of the corresponding Ada cycle subprogram. The hide directive is also added at the beginning of the init and reset procedures.

When setting `-root_package` option, the hierarchy of packages specified in `<package_path>` is considered as the top-level of the model and all SPARK code generated by SCADE Suite KCG is in the `<package_path>` package. For instance, use `-root_package P1.P2` and the counterpart of a Scade package `P` becomes the Ada package `P1.P2.P` (provided that `P` has a direct translation into valid SPARK code).

CASE OF TOP-LEVEL DECLARATION IN SCADE SUITE MODEL

If the SCADE Suite input model contains global declarations that do not belong to any package, SCADE Suite KCG introduces an additional top-level package named `Kcg_Top` and the counterpart SPARK code of such global Scade declarations is generated within package `Kcg_Top`. For instance, use `-root_package P1.P2`, and a new package `P1.P2.Kcg_Top` is created if there are top-level Scade declarations.

CASE OF PACKAGE WITH IMPORTED AND NON-IMPORTED DECLARATIONS

If a SCADE Suite model package contains an imported declaration and its parent package contains both imported and non-imported declarations, SCADE Suite KCG introduces an additional `Kcg_Imported` sub-package.

- The SPARK code corresponding to imported declarations (including sensors) is generated within `<package_path>.Kcg_Imported`. `<package_path>` package is the counterpart of the SCADE Suite package taking into account `-root_package` and the introduction of other packages, if need be.
- The SPARK code corresponding to non-imported declarations is generated directly within package `<package_path>`.

Example 1: SCADE Suite package mixing imported and non-imported declarations

```
package P
  type imported T;
  const K: int16 = 42;
  const imported N: int16;
  sensor S: float64;
  function F (x1, x2: T) returns (y: T) ...
  function imported G (x: T) returns (y: T);
end;
```

After re-structuring by SCADE Suite KCG

```
package P
  package Kcg_Imported
    type imported T;
    const imported N: int16;
    sensor S: float64 ;
    function imported G (x: T) returns (y: T);
  end;
  const K: int16 = 42;
  function F (x1, x2: T) returns (y: T) ...
end;
```

CASE OF REFERENCES TO SUB-PACKAGES

If a SCADE Suite model package contains at least a global declaration referring to one of its sub-packages, SCADE Suite KCG introduces an additional `Kcg_Top` sub-package. The counterpart SPARK code of all global Scade declarations in the SCADE Suite model package (those not already in a sub-package) are generated within `<package_path>.Kcg_Top` package. `<package_path>` is the counterpart of the SCADE Suite model package.

Example 1: SCADE Suite package with reference to sub-packages

```
package P
  package MyTypes
    type T1 = int32;
    type T2 = T1^10;
  end;
  function F (x: MyTypes::T1) returns (y: MyTypes::T2) ...
end;
```

After re-structuring by SCADE Suite KCG

```
package P
  package MyTypes
    type T1 = int32;
    type T2 = T1^10;
  end;
  package Kcg_Top
    function F (x: MyTypes:: T1) returns (y : MyTypes :: T2) ...
  end;
end;
```

CASE OF CIRCULAR DEPENDENCIES BETWEEN PACKAGES

If a SCADE Suite model input contains circular dependencies between global declarations (of different packages), SCADE Suite KCG introduces additional packages to break such dependencies.

For each global Scade declaration `P::decl` that belongs to a dependency cycle (this includes the declarations that depend on a declaration belonging to a dependency cycle, and recursively), a package named `Kcg_P_decl` is introduced at the same level in the package hierarchy as the counterpart of package P. The SPARK code corresponding to the single declaration `P::decl` is generated in the `Kcg_P_decl` package.

Note

Exception: if the global Scade declaration `P::decl` is already the one and only declaration in P, no new package is created to extract that declaration.

Example 1: SCADE Suite package with circular dependencies between packages

```
package P
  const K1: Q::T1 = 42;
  const K2: int32 = K1 + 1;
  const K3: int32 = 42;
end;
package Q
  type T1 = int32;
  type T2 = T1 ^ P::K1;
  type T3 = float32;
end;
package R
  type T = Q::T3 ^ 10;
  function F (x: Q::T3) returns (y: T) ...
end;
```

The global declarations involved in a dependency cycle are: P::K1, P::K2, Q::T1 and Q::T2.

After re-structuring by SCADE Suite KCG

```
package P
  const K3: int32 = 42;
end;
package Kcg_P_K1
  const K1: Kcg_Q_T1::T1 = 42;
end;
package Kcg_P_K2
  const K2: int32 = Kcg_P_K1::K1 + 1;
end;
package Q
  type T3 = float32;
end;
package Kcg_Q_T1
  type T1 = int32;
end;
package Kcg_Q_T2
  type T2 = Kcg_Q_T1::T1 ^ Kcg_P_K1::K1;
end;
package R
  type T = Q::T3 ^ 10;
  function F (x: Q::T3) returns (y: T) ...
end;
```

Mapping File

In addition to trace comments, or headers and footers added into the generated code, an XML traceability file called mapping file is generated by KCG/MCG. This file, named `mapping.xml` (in KCG/MCG), enables you to

trace elements between the generated code and the SCADE Suite input model. See details about KCG/MCG mapping file from "Mapping File Access for SCADE Suite KCG/MCG/ACG" on page 38 in *SCADE Python API Guide*.

Note

For further details, consult section about mapping file generation in *SCADE Suite KCG Tool Operational Requirements* (TOR) document relative to the KCG version in use.

The following table sums up entities traced in the mapping file:

Table 9.8: Entities traced during code generation

Types	Operator control blocks	Operator automatons
Sensors	Constants	Operator iterators
Observable Variables	Operators	Memory elements
Operator instances	Assumes and Guarantees	

For each of the above mentioned entities, the Scade name in the input model and the name of the corresponding structure in the target code is given. The type of the target code structure is also given whenever relevant.

Example: The following shows a sample of the `mapping.xml` mapping file for the ABC_N model.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--$************** SCADE Suite KCG 32-bit 6.6 (build i19) **************
** Command: kcg66.exe -config C:/Users/Public/Documents/ANSYS Inc/v201/SCADE/examples/CruiseControl/CruiseControl/KCG/config.txt
** Generation date: 2019-08-21T09:59:08
*****************************************************************$-->
<mapping>
  <config>
    <option name="use_types" value="#off#"/>
    <option name="imported_root_package" value="#off#"/>
    <option name="root_package" value="#off#"/>
    <option name="xml_basename" value="kcg_xml_filter_out.scade"/>
    <option name="state_vector" value="#off#"/>
    <option name="static" value="#off#"/>
    <option name="separate_io_all" value="#off#"/>
    <option name="no_separate_io" value="#off#"/>
    <option name="separate_io" value="#off#"/>
    <option name="warn_error" value="#off#"/>
    <option name="short_circuit" value="#off#"/>
    <option name="wrap_c_ops" value="#off#"/>
    <option name="input_file" value="C:/Program Files/ANSYS Inc/v201/SCADE/SCADE/libraries/SC65/libverif/libverif.xscade"/>
```

Pragma Syntax and Usage

Pragmas allow to pass information to SCADE Suite KCG Code Generator and Multicore Code Generator (MCG). Pragma information is attached to specific objects in SCADE Suite models with respect to a given syntax. SCADE Suite Code Generators recognize this syntax and take the information into account for code generation. Pragmas may be used also to save information for tools that parse and/or produce Scade files. Even if syntactically valid such pragmas are ignored by SCADE Suite Code Generators.

- "General Pragmas"
- "Ada-Specific Pragmas"
- "C-Specific Pragmas"
- "MCG-Specific Pragmas"
- "Sequence of Pragmas"
- "Pragmas by Code Generator Version"

Important Note

For attaching pragma information within SCADE Suite models, consult "Setting Pragmas on Model Elements" on page 850 in *SCADE Suite User Manual*. About pragmas lexical elements in Scade language, consult "Pragmas" on page 6 of *Scade Language Reference Manual*.

General Pragmas

The following pragmas are available:

- "Occurrence Pragma"
- "default Pragma"
- "doc Pragma"
- "enum_val Pragma"
- "expand Pragma"
- "noexpand Pragma"
- "initializer Pragma"
- "keep Pragma"
- "name Pragma"
- "manifest Pragma"
- "separate_io Pragma"
- "no_separate_io Pragma"

OCCURRENCE PRAGMA

Such pragma carries the instance name information attached to Scade expressions represented with design elements in the model. It identifies within the mapping file any call tagged by users with a specific ID.

Syntax

```
#id, where id = ('_' | [a-z] | [A-Z] | [0-9])+
```

The pragma must be attached to a Scade expression as described below:

Design elements	BNF expression
Last value of variables/signals	`#id last expr`
Previous value of flows (no value at first cycle)	`#id pre expr`
Initial value of flows (at first cycle)	`expr #id -> expr`
Previous value of initialized flows	`#id fby ( list ; expr ; list )`
Count down of repeated occurrence	`expr #id times expr`
Filtering of flows on given cycles or flow synchronization at different rates	`expr #id when clock_expr`
Merging of incoming flows into faster flow	`#id merge ( expr ; list {{ ; list }} )`
Sign inversion of numeric flows or conversion of numeric expressions	`#id unary_arith_op expr`
Arithmetic operations: sum, difference, multiplication, division, modulo)	`expr #id bin_arith_op expr`
Comparison of numeric flow values	`expr #id bin_relation_op expr`
Logical NOT	`#id not expr`
Logical AND, OR, XOR	`expr #id bin_bool_op expr`
Logical sharp (exclusive)	`#id ( list )`
Reverse of array values	`#id reverse expr`
Concatenation of array flows	`expr #id @ expr`
Static projection of arrays	`expr #id index`

Design elements	BNF expression
Static slice of arrays	`expr #id [ expr .. expr ]`
Dynamic projection of arrays	`( expr #id . {{ label_or_index }}+ default expr )`
Swap of array dimensions	`#id transpose ( expr ; integer ; integer )`
Array construction by value repetition	`expr #id ^ expr`
Array construction from list of elements	`#id [ list ]`
Projection of a structure	`expr #id . id`
Data structure from different flows	`#id { label_expr {{ , label_expr }} }`
Structure/array copy with value modification	`( expr #id with {{ label_or_index }}+ = expr )`
Decision based on Boolean condition and decision branches	`#id if expr then expr else expr`
Decision based on condition and decision pattern	`( #id case expr of {{ case_expr }}+ )`
User-defined operator	`#id operator ( list )`
Parameterized operator	`( #id prefix << [[ expr {{ , expr }} ]] >> )`
Composition of output flow of structure type from input flows of different types	`( #id make path_id )`
Decomposition of input flow of array/ structure types into output flows of different types	`( #id flatten path_id )`
Array iteration (map, fold, mapfold) and array iteration with access to index (mapi, foldi)	`( #id iterator operator << expr >> )`
Clocked activation of operator	`( #id activate operator every clock_expr )`
Boolean activation with default value	`( #id activate operator every expr default expr )`
Boolean activation with default initial value	`( #id activate operator every expr initial default expr )`
Resetting of operator instantiation	`( #id restart operator every expr )`
Array iteration with partial mapping	`( #id mapw operator << expr >> if expr default expr )`
Array iteration with partial mapping with access to index of iteration	`( #id mapwi operator << expr >> if expr default expr )`

Design elements	BNF expression
Array iteration with partial folding	`( #id foldw operator << expr >> if expr )`
Array iteration with partial folding with access to index of iteration	`( #id foldwi operator << expr >> if expr )`

Example:

```
x = #aNodeCall_1 aNode(b, c);
```

DEFAULT PRAGMA

When `default` pragma is attached to a state in a state machine or to an enumeration value, the generated C **switch/case** (resp. Ada **case/when**) constructs for the state machine or the enumeration handles the corresponding state or enumeration value in the **default** branch (for C code) or in **others** branch (for Ada code).

Syntax

```
#pragma kcg default #end
```

Example 1: Consider the following Scade sample with state machine

```
node Root (In1 , In2: int32) returns (Out1: int32)
 let
   automaton A
     initial state S1
       Out1 = In1;
     until if true resume S2;
     #pragma kcg default #end
     state S2
       Out1 = In2;
     until if true resume S1;
   returns Out1;
tel;
```

Depending on pragma usage, the generated C code is:

without **`default`** pragma:

```
/* act_A */ switch (A_state_act) {
  case SSM_st_S1_A :
    outC->A_state_nxt = SSM_st_S2_A;
    outC->Out1 = inC->In1 ;
    break ;
  case SSM_st_S2_A :
    outC->A_state_nxt = SSM_st_S1_A;
    outC->Out1 = inC->In2 ;
    break ;
  default :
    /* this default branch is unreachable */
    break ;
}
```

with **`default`** pragma:

```
/* act_A */ switch (A_state_act) {
  case SSM_st_S1_A :
    outC->A_state_nxt = SSM_st_S2_A;
    outC->Out1 = inC->In1 ;
    break ;
  case SSM_st_S2_A :
  default :
    outC->A_state_nxt = SSM_st_S1_A;
    outC->Out1 = inC->In2 ;
    break ;
}
```

Example 2: Consider the following Scade sample with enumeration

```
type T2 = enum {#pragma kcg default #end D, E, F };
y = (case x of | D: 1 | E: 2 | F: 3 );
```

Depending on pragma usage, the generated Ada code is:

without **`default`** pragma:

```
case (InS.x) is
  when Kcg_Types .D =>
    OutS.y := 1;
  when Kcg_Types .E =>
    OutS.y := 2;
  when Kcg_Types .F =>
    OutS.y := 3;
end case;
```

with **`default`** pragma:

```
case (InS.x) is
  when Kcg_Types .E =>
    OutS.y := 2;
  when Kcg_Types .F =>
    OutS.y := 3;
  when others =>
    OutS.y := 1;
end case;
```

DOC PRAGMA

Such pragma contains comments about a language element. When `doc` pragma is attached to an element, the corresponding comment appears in the generated code if, and only if, the element has a counterpart in the code. A `doc` pragma attached to an instance of an expanded operator does not translate into a comment in the code.

Syntax

```
#pragma kcg doc lines of documentation... #end
```

The pragma is attached to the following elements when preceding related identifier:

Element	Identifier	Element	Identifier
packages	package ID	signals	signal ID
operators	operator ID	local flows	flow ID
imported operator	operator ID	state machines	**`automaton`** keyword
types	type ID	states	**`state`** keyword
inputs	input ID	enum value (in type declaration)	value ID
outputs	output ID	structure field name (in type declaration)	the label
constants	constant ID	instance of operator	instantiated operator ID
sensors	sensor ID		

When `-target C` is used, the doc pragma cannot contain the `*/` character sequence. Use a double ## sign to distinguish it from `#end` as illustrated below:

```
const #pragma kcg doc foo ##end #end A : int32 = 42;
function Root() returns (Out1: int32)
  Out1 = A;
```

Example: Check the appropriate sample by language element

```
type
#pragma kcg doc a structure #end t7=
  {#pragma kcg doc x field doc #end x: int32};
#pragma kcg doc enum1 definition #end
t9 = enum1 {#pragma kcg doc AAA doc #end AAA,
     BBB, CCC};

const
#pragma kcg doc constant C1 is initialized
with K
#end
  c1: int32 = K;

sensor #pragma kcg doc documentation of s1
#end
  s1: int32;
  s2: int32;
  #pragma kcg doc documentation of s3 #end
  s3, s4, s5: int32;
```

```
node
  #pragma kcg doc description of f1 #end
  f1 (a,#pragma kcg doc description of b #end
b,
  c: int32; d: float64) returns (w,x,y: int32;
  #pragma kcg doc description of z #end
  z: float64 last = K)

sig
  s1,
  #pragma kcg doc description of s2 #end s2,
  s3;

var
  #pragma kcg doc description of v4 #end
  v4: float64 last = K;
let
 #pragma kcg doc description of state machine
#end
  automaton
    #pragma kcg doc state S1 doc #end
    state S1
```

ENUM_VAL PRAGMA

The `enum_val` pragma is used to specify the value of an enumeration item. The value is copied as is in the generated code and is not used by KCG/MCG/ACG for internal computation. The pragma must be located before the corresponding enumeration value.

Syntax

```
#pragma kcg C:enum_val number #end (KCG/MCG/ACG)
#pragma kcg Ada:enum_val number #end (KCG)
```

If `-target C` is used, `C:enum_val` pragma has the following characteristics:

- Pragma value meets ISO IEC 9899 constraints, section “Enumeration specifiers”.
- C enumeration constant values (taking into account pragma values and ISO IEC 9899 semantics) of the enumerated type generated by KCG/MCG are all different.

If `-target Ada` is used, `Ada:enum_val` pragma has the following characteristics:

- Either all enumerations of a type have an `Ada:enum_val` pragma, or no enumeration has the pragma.
- Pragma values for a type are valid values: values are static and specify an integer code for each value of enumeration type; the associated integer codes satisfy the predefined ordering relation of the type.

Example:

```
type bgcolor = enum {
  #pragma kcg C:enum_val 0 #end grey,
  #pragma kcg C:enum_val 1 #end yellow, green } ;
```

EXPAND PRAGMA

When `expand` pragma is attached to a user-defined operator, the code is inlined within the code generated for the first non-expanded operator where it is instantiated. The -noexp option is stronger than the `expand` pragma: the code is not inlined if the user-defined operator is specified in the list of non-expanded operators. The pragma must be located before the operator it is attached to in the operator declaration.

Syntax

```
#pragma kcg expand #end
```

Example:

```
function #pragma kcg expand #end N(foo: int8) -- N is expanded
returns (y: int8)
let
  y = foo;
tel
```

NOEXPAND PRAGMA

When `noexpand` pragma is attached to a user-defined operator, the code is not inlined. Two files (<operator_name>.h and <operator_name>.c) are generated for each monomorphic and non-inlined instance in the cone of influence of the root operator. The -exp option is stronger than the `noexpand` pragma: the code is inlined if the user-defined operator is specified in the list of expanded operators. The pragma must be located before the operator it is attached to in the operator declaration.

Syntax

```
#pragma kcg noexpand #end
```

Example:

```
function #pragma kcg noexpand #end N(foo: int8) -- N is not expanded
returns (y: int8)
let
  y = foo;
tel
```

INITIALIZER PRAGMA

When `initializer` pragma is attached to an imported type, the initialization function of the associated type is named based on the pragma definition. If an imported type has no pragma, the default name `<type_name>_init` is used. The pragma must be located before the imported type it is attached to in the type declaration.

Syntax

```
#pragma kcg C:initializer string #end (KCG/MCG)
#pragma kcg Ada:initializer string #end (KCG)
```

The specified string must be a valid C/Ada identifier.

Example 1: Ada pragma

```
type
  imported #pragma kcg Ada:initializer Init_T1 #end T1 ;
  imported #pragma kcg Ada:initializer Init_T2 #end T2 ;
```

Example 2: C pragma

```
type
  imported #pragma kcg C:initializer Init_T1 #end T1 ;
  imported #pragma kcg C:initializer Init_T2 #end T2 ;
```

KEEP PRAGMA

Any declaration of a variable or signal with the keep pragma in the input model has a counterpart in the generated code for all optimization levels, except when the variable or signal is only used in a dead branch optimized by KCG/MCG. The pragma must be located before the identifier of the variable, signal, or imported function.

Syntax

```
#pragma kcg keep #end
```

Example:

```
sig
  s1,
  #pragma kcg keep #end s2,          -- pragma keep on the signal s2
  s3;
var
  k: bool;
  #pragma kcg keep #end v: bool;    -- pragma keep on the variable v
  i: int16;
function
  imported #pragma kcg keep #end F (i: int16) returns (o: int16); -- pragma
keep on the function F
```

If an imported operator declaration in the input model has a keep pragma, each non-specialized instance of the operator has a corresponding function call in the generated code. If the operator call is in a dead branch optimized by the tool, it may not appear in the generated code even if the operator has the keep pragma. A C variable corresponding to a variable or signal defined in SCADE Suite models with a keep pragma cannot be optimized. The same applies for an imported operator with a keep pragma: its instances cannot be optimized even if its outputs are not used, except instances present in a non-reachable part of the code. This may require to keep an instance of a non-imported operator containing an instance of an imported operator with a keep pragma.

Example: Consider the following example

```
activate A
   if false
   then _ = #1 F (...);
   else _ = #2 F (...);
returns ..;
```

Instance #1 of F is in a dead branch, it may be optimized by KCG/MCG, even if the declaration of F has the keep pragma attached to. Provided the whole control block A can be activated, instance #2 of F can be activated (it is always activated when A is activated). However, its output is not used. Without keep pragma, it could be removed by optimizations. With keep pragma, it is not removed and appears in the generated code.

MANIFEST PRAGMA

The `manifest` pragma controls the name of the types generated in `kcg_types.h`. When attached to a named type, the pragma forces to use the name of the named type (or its `name` pragma) when naming the type generated for the type equivalence class. The pragma must be located before the type it is attached to in the type declaration.

Syntax

```
#pragma kcg manifest #end
```

Example: Manifest pragma definition in a SCADE model

```
type A = {f1: int32};
type B = A;
type C = {f1: int32};
type #pragma kcg manifest #end D = C;
```

Result of manifest pragma in `kcg_types.h` file

```
typedef struct kcg_tag_D { kcg_int f1; } D;
typedef D A;
typedef D B;
typedef D C;
```

NAME PRAGMA

When `name` pragma is attached to a global identifier[1], this name is present as is in the generated code with the exception of imported polymorphic operators where only the `name` pragma of their specializations is kept. KCG/MCG/ACG does not modify this pragma information: identifiers set with this pragma are ignored by the -globals_prefix option.

Syntax

```
#pragma kcg C:name string #end (KCG/MCG/ACG)
#pragma kcg Ada:name string #end (KCG)
```

The pragma is attached to Scade elements when preceding the following identifiers:

Element	Identifier	Element	Identifier
operators	operator ID	local flows	flow ID
groups	group ID	assertions	assertion ID
types	type ID	named state machine	**`automaton`** keyword
inputs	input ID	named activate if	**`activate`** keyword
outputs	output ID	named activate when	**`activate`** keyword
size parameters	parameter ID	anonymous state machine	**`automaton`** keyword
type variables	type variable ID	anonymous activate if	**`activate`** keyword
constants	constant ID	anonymous activate when	**`activate`** keyword
sensors	sensor ID	states	**`state`** keyword
signals	signal ID	enum value (in type declaration)	value ID

1. Global C identifiers are used as counterparts for sensors, types, constants, operators, and any imported declaration.

The specified string must be a valid C/Ada identifier. During code generation checks on `name` pragmas, errors may occur if a pragma is too long, not unique, not a valid C identifier, or if the global prefix is too long.

Example:

```
function #pragma kcg Ada:name MyName #end MyNode(x: int16) returns (y: int16)
  y = x;
function imported PolyEx (a: 'T) returns (b: 'T) ;
function #pragma kcg C:name Example #end MonoEx(a: int16) returns (b: int16)
         specialize PolyEx
  b = a + 1;
```

SEPARATE_IO PRAGMA

When `separate_io` pragma is attached to an operator declaration (including root operators and imported operators), the code generated for the cycle function is different. The outputs are no more in the context but passed as separate parameters. The inputs are no more passed in an input context for the root operator except if the number of parameter of the function exceeds the value set by the -input_threshold option. The -no_separate_io option is stronger than the `separate_io` pragma: the code of the cycle function is not modified in case the operator is specified in the list of operators not marked for separate I/O. The pragma must be located before the operator it is attached to in the operator declaration.

Syntax

```
#pragma kcg separate_io #end
```

Example:

```
function #pragma kcg separate_io #end
  N(x: int8) returns (y: int8)
   y = x ;
function imported #pragma kcg separate_io #end
  N2(x: int8) returns (y: int8);
```

NO_SEPARATE_IO PRAGMA

When `no_separate_io` pragma is attached to an operator declaration (including root operators), the code generated for the cycle function is without separate I/O. The -separate_io option is stronger than the `no_separate_io` pragma: the code of the cycle function is modified in case the operator is specified in the list of operators marked for separate I/O. The pragma must be located before the operator it is attached to in the operator declaration.

Syntax

```
#pragma kcg no_separate_io #end (KCG/MCG)
```

Ada-Specific Pragmas

SPARK HIDE PRAGMA

When a `spark hide` pragma is attached to a package or an operator, the `--# hide` directive is added respectively on top of the declarative part of the corresponding Ada package body or of the corresponding Ada cycle subprogram in the generated Ada code.

Syntax

```
#pragma kcg spark hide #end (KCG)
```

Example:

```
package #pragma kcg spark hide #end P
  ...
end;
```

PRAGMA PRAGMA

The `pragma` pragma generates pragmas in target code. If attached to a type, sensor, constant, or operator declaration in SCADE Suite model and if the declaration has an Ada counterpart (not a sub-type), the pragma value is added as Ada pragma after the corresponding entity in the generated code.

Syntax

```
#pragma kcg Ada:pragma string #end (KCG)
```

The pragma is attached to an element declaration in SCADE Suite when it precedes the following text:

Element	Operators	Imported operators	Types	Constants	Sensors	Signals	Local flows
Text	operator ID	operator ID	type ID	constant ID	sensor ID	signal ID	flow ID

The specified string must be a valid Ada pragma.

Example: Pragma on type declaration used with manifest pragma to force the type name

```
type
  #pragma kcg Ada:pragma Pack (ipair) #end
  #pragma kcg manifest #end
  ipair = {re: int32, im: int32};
```

The following Ada code is generated (record type declaration in `kcg_types.ads`:

```
type ipair is
  record
    re : Kcg_Config.Kcg_Int32;
    im : Kcg_Config.Kcg_Int32;
  end record;
pragma Pack (ipair);
```

C-Specific Pragmas

CONST PRAGMA

When the `const` pragma is attached to a scalar constant, it prevents the usage of the `#define` directive. It is generated in the code as a non-scalar constant. The pragma must be located before the constant identifier.

Syntax

```
#pragma kcg C:const #end (KCG/MCG)
```

Example:

```
const N: int32 = 10;                              -- no pragma on N
const #pragma kcg const #end K: int32 = -1;       -- a pragma attached to K
```

SCALAR PRAGMA

When attached to an imported type, this pragma specifies that all type values are considered as scalar, and that a simple C assignment is used for copy. The pragma must be located before the imported type it is attached to in the type declaration.

Syntax

```
#pragma kcg C:scalar #end (KCG/MCG)
```

Example:

```
type
  imported #pragma kcg scalar #end TYPE_1; -- TYPE_1 has a scalar pragma
  imported TYPE_2;                         -- TYPE_2 has no pragma
```

MCG-Specific Pragmas

TASK PRAGMA

When the `C:task` pragma is attached to any operator instance, it identifies the operator instances to be extracted as separate tasks during MCG code generation. For attaching such pragma in SCADE Suite models, consult "Specifying Task Pragmas" on page 888 in *SCADE Suite User Manual*.

Syntax

```
#pragma kcg C:task string #end
```

The specified string must be a valid C identifier.

Pragmas can only be attached to instances with an occurrence pragma. They cannot be attached to elements inside anonymous control blocks.

Example: Consider an operator declared with the following pragma information:

```
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);
```

The same could be declared equivalently as:

```
function root (i: int32) returns (o: int32)
  o = #1 F1(i) - #1 F2(i);
```

with the following pragma definition in the external pragma file:

```
#pragma kcg C:task F1_task #end root/(F1#1)/
#pragma kcg C:task F2_task #end root/(F2#1)/
```

The path of an instance must use the following syntax:

`decl_path / {{control}} (op_path# instance_name)/`

- `decl_path` is the path of the operator containing the instance
- `{{control}}` is the local path of the scope containing the instance
- `op_path` is the path of the called operator
- `instance_name` is the instance name (i.e., occurrence pragma)

UNROLL PRAGMA

When `unroll` pragma is attached to an iterator instance, it identifies the iterator instances to be extracted as separate tasks during MCG code generation (to use such pragma in SCADE Suite models, consult "Declaring Iterator Instances as Unrolled" on page 894 in *SCADE Suite User Manual*). This pragma can also be used to replace a 'for' loop in the generated code with n calls (possibly more efficient for a small n).

Syntax

```
#pragma kcg unroll integer #end
```

Sequence of Pragmas

It is possible to use multiple pragmas preceding the same identifier as follows:

```
const
  #pragma kcg const #end -- 1st pragma
  #pragma kcg name MyPragmaName #end -- 2nd pragma
  #pragma kcg doc documentation about K #end -- 3rd pragma
  K : int32 = -1 ; -- attached to K
```

Pragmas by Code Generator Version

The pragmas are not available with all Code Generator versions.

Table 9.9: List of pragmas by Code Generator version

Pragma	KCG		MCG C	ACG C
	C	Ada		
const	✓		✓	✓
default	✓	✓	✓	✓
doc	✓	✓	✓	✓
enum_val	✓	✓	✓	✓
expand	✓	✓	✓	✓
hide		✓		
initializer	✓	✓	✓	✓
keep	✓	✓	✓	✓
manifest	✓	✓	✓	✓
name	✓	✓	✓	✓
no_separate_io	✓	✓	✓	✓
noexpand	✓	✓	✓	✓
occurrence	✓	✓		✓
pragma		✓		
scalar	✓		✓	✓
separate_io	✓	✓	✓	✓
spark		✓		
unroll			✓	
task			✓	

10 / Multicore Target Integration of Task-Based Code

This chapter provides dedicated guidelines for using task-based code generated with Multicore Code Generator (MCG) to write integration code for multicore targets.

- "Traceability and Mapping File"
- "Method Splitting"
- "Implementation of Communication Channels"
- "Advanced Multicore Implementation"
- "Examples of Integration"

For instruction about task-based code generation, see Chapter 22 about "Generating and Integrating Task-Based Code" in *SCADE Suite User Manual*.

Traceability and Mapping File

The generation of mapping file by Multicore Code Generator takes into account tasks generation. The following is a brief introduction for using the content of the mapping file. Each model instance extracted into a task is linked to the C counterparts of the generated task (like context type, reset/init functions), using the same roles as for an operator.

The generated Kahn Process Network (KPN) model described in the **`<architecture>`** section of the mapping file represents the generated tasks and channels, as well as their mappings with the input model and the generated code. This formal model specifies the usage conditions of the code, for instance the dependencies between tasks and how communications should be implemented. Tasks are represented as a tree of **`<task>`** tags (the parent is the task where the operator calling the task is executed). Each task contains one **`<method>`** tag for each method, with a `deps` attribute that lists the methods it depends on (*i.e.,* it must be executed after these methods). Each method is mapped to the corresponding C function with a `"method-function"` role. A clocked task is linked with its clock like context fields (`"clock"` and `"clock-value"` roles). Each channel is described in a **`<channel>`** tag. Its attributes list its sender and receiver. The channel is linked with the corresponding C type, initial token constant (if any), and the context fields in the sender and receiver. These fields can also be clocked like other context fields (*e.g.,* probes).

Method Splitting

A task can have several cycle methods. Method splitting has *several objectives*:

- Ensuring that communications are performed outside of the sequential body of methods.
- Preventing the addition of dependencies between generated tasks. If two instances have no data-flow dependencies in the input model, then the corresponding tasks should have no dependencies in the generated KPN. This ensures that code generation does not arbitrarily restrict the possible schedules.

Method splitting also ensures some invariants to simplify integration:

- Inputs of an operator are read-only by one method
- Each channel is only read/written by one method

The algorithm is inspired by the modular scheduling algorithm. For each `eq` equation, the following sets are computed:

- `R(eq)` is the set of input channels that `eq` depends on.
- `S(eq)` is the set of output channels that depends on `eq`.

Note that channels with initial values are ignored, because they do not introduce dependencies. The equations are split according to these sets:

- A first method that contains all equations such that $R(eq) = \varnothing$; namely the part of the operator that does not depend on any channel.
- Another method with equations such that $R(eq) \neq \varnothing$ and $S(eq) = \varnothing$; namely the equations that do not influence any channel.
- The other equations are split according to their R sets: two equations are in the same method if they have the same R set.

Then a second pass ensures the invariants:

- If an input is read by a method other than the first (the one such that R(.)=Ø), then the input is copied to a temporary variable in the context which is read by the other methods.
- All reads of a given channel are grouped in the same method. This may require to copy some fields to a temporary variable in the context, to avoid introducing new dependencies between tasks. Writes to each output channel are processed the same way.

Each temporary variable introduces a dependency between the method that reads it and the method that defines it.

Example: Consider the following code

```
node root_split (i:int32) returns (o:int32)
var x:int32;
let
  x = #pragma kcg C: task F1_task #end F1(i);
  o = #pragma kcg C: task F2_task #end F2(G(x)) + i;
tel
```

The following methods are generated:

```
// R(.) = empty
void root_split_1(inC_root_split *inC, outC_root_split *outC)
{
  (*outC->F1_task_in_ch).i1 = inC->i;
  outC->i = inC->i;
}
// R(.) = {F1_task_out_ch}
void root_split_2(outC_root_split *outC)
{
  (*outC->F2_task_in_ch).i2 = /* o=(G)/ */ G((*outC->F1_task_out_ch).o1);
}
// S(.) = empty
void root_split_3(outC_root_split *outC)
{
  outC->o = (*outC->F2_task_out_ch).o2 + outC->i;
}
```

The first method `root_split_1` reads the input, stores a copy in the context, and sends to the input channel of `F1_task`. The second method `root_split_2` contains the code that depends on `F1_task` that sends on the input channel of `F2_task`. Finally, the last method receives from

F2_task and computes the output. Note that root_split_3 depends on root_split_1, because it reads the field outC->i that is written by it. This dependency is represented in the mapping file by deps attribute:

```
<task id="33">
  <method id="34"/>           -- root_split_1
  <method id="35"/>           -- root_split_2
  <method id="36" deps="34"/> -- root_split_3
</task>
```

Implementation of Communication Channels

For target-specific integration, users must provide the implementation of channel operations. Communication channels are single-producer, single-consumer FIFOs of size one. It means that a buffer of size one is sufficient to avoid deadlocks in the KPN. Implementing this data-structure is a very common activity in multicore programming (a.k.a. *bounded producer-consumer problem* in literature). The mapping file provided by Multicore Code Generator contains all information necessary for implementing communication channels.

Some classical approaches to communication implementation using generic multi-core primitives are proposed hereafter.

- "Implementation with Semaphores"
- "Implementation with Condition Variables"
- "Special Implementation Cases"
- "Implementation with Initial Token"
- "Local Communications"

Implementation with Semaphores

A common way to solve the *bounded producer-consumer problem* is to use two semaphores: one to represent the empty FIFO and another to represent the full FIFO.

```
in_ch_ty in_ch;
semaphore in_ch_empty=1; -- number of empty slots
          in_ch_full=0; -- number of produced items
```

Producer	Consumer
`wait(in_ch_empty);`	`wait(in_ch_full);`
`P(); -- write to in_ch`	`C(); -- read from in_ch`
`signal(in_ch_full);`	`signal(in_ch_empty);`

The producer waits for the FIFO to be empty, writes a new data, and signals that the FIFO is full. Similarly, the consumer waits for the FIFO to be full, reads the data, and signals that the FIFO is empty. In this case, the method P directly writes/reads the shared `in_ch` variable, so that the semaphore is signaled after the execution of P. It is also possible to copy the data to a local variable and then signal before executing P.

Producer	Consumer
`in_ch_ty in_ch_loc;`	`in_ch_ty in_ch_loc;`
`P(); -- write to in_ch_loc`	`wait(in_ch_full);`
`wait(in_ch_empty);`	`in_ch_loc = in_ch;`
`in_ch = in_ch_loc;`	`signal(in_ch_empty);`
`signal(in_ch_full);`	`C(); -- read from in_ch_loc`

Another advantage of this second version is that accesses to shared variables are clearly separated from computations. This can be useful to avoid interferences in the access to shared resources.

Implementation with Condition Variables

Another common implementation of the *producer-consumer problem* relies on condition variables (semaphores and condition variables are similar and one can be implemented using the other). It is necessary to use a mutex to guarantee mutual exclusion, one condition variable for each event to wait for (FIFO full, FIFO empty) and a Boolean to store the state of the FIFO (handled by semaphore count in previous implementation).

```
in_ch_ty in_ch;
mutex m;
condvar empty, full;
bool full; -- state of the FIFO
```

Producer	Consumer
`lock(m);`	`lock(m);`
`while(full) wait(m, in_ch_empty);`	`while(not full) wait(m, in_ch_full);`
`P(); -- write to in_ch`	`C(); -- read from in_ch`
`signal(in_ch_full);`	`signal(in_ch_empty);`
`unlock(m);`	`unlock(m);`

Special Implementation Cases

The previous implementations check if the FIFO is empty before writing a new value. This is not necessary in some cases, because this property always holds when the channel sender is executed. This is the case if the KPN is *feed-forward*, that is, there is no cycle, and there is no task without input channels (such a task is obtained from an instance of operator with no inputs or only constant inputs). The implementation can use only one semaphore.

```
in_ch_ty in_ch;
semaphore in_ch_full=0;
```

Producer	Consumer
`P(); -- write to in_ch` `signal(in_ch_full);`	`wait(in_ch_full);` `C(); -- read from in_ch`

Beware that this implementation is incorrect if:

- A task has no inputs: so-called *back-pressure* is needed to ensure that this task is executed at the same rate as others. Back-pressure is provided by the `in_ch_empty` semaphore. Such back-pressure is needed for all tasks which do not receive (directly or indirectly) a value from the root task.
- A channel has an initial token: ensure the initial value is read before writing a new one.

Implementation with Initial Token

In case of a channel with initial token, if a buffer of size one is used, the sender cannot be executed until the receiver is done. While this is correct and sufficient to avoid deadlocks, it restricts the possible schedules. This restriction can be lifted by using a size two buffer, implemented for instance using a circular buffer.

```
in_ch_ty in_ch[2];
in_ch[0] = v0;
semaphore in_ch_empty=1; -- number of empty slots
          in_ch_full=1; -- number of produced items

int w_idx = 1; -- index of producer
int r_idx = 0; -- index of consumer
```

Producer

```
wait(in_ch_empty);
P(); -- write to in_ch[w_idx]
signal(in_ch_full);
w_idx = (w_idx+1) % 2;
```

Consumer

```
wait(in_ch_full);
C(); -- read from in_ch[r_idx]
signal(in_ch_empty);
r_idx = (r_idx+1) % 2;
```

A variant with local copies of the channels is as follows:

Producer

```
in_ch_ty in_ch_loc;

P(); -- write to in_ch_loc
wait(in_ch_empty);
in_ch[w_idx] = in_ch_loc;
w_idx = (w_idx+1) % 2;
signal(in_ch_full);
```

Consumer

```
in_ch_ty in_ch_loc;

wait(in_ch_full);
in_ch_loc = in_ch[r_idx];
r_idx = (r_idx+1) % 2;
signal(in_ch_empty);
C(); -- read from in_ch_loc
```

Local Communications

If the sender and receiver of a channel are executed in the same thread, there is no need to use any synchronization. The channel can be implemented using a local variable and the sender must be executed before the receiver. If the sender and receiver of a channel with initial token, are executed in the same thread, then there are two cases:

- If the receiver is executed before the sender, then one local variable is sufficient:

```
in_ch_ty in_ch_loc = v0;
C(); -- read from in_ch_loc
P(); -- write to in_ch_loc
```

- Otherwise, it is necessary to use two variables, for instance:

```
in_ch_ty in_ch_loc = v0, in_ch_pre;

in_ch_pre = in_ch_loc;
P(); -- write to in_ch_loc
C(); -- read from in_ch_pre
```

Advanced Multicore Implementation

Writing integration code for multicore targets may require more advanced implementation and approaches.

- "Clocked Tasks"
- "Tasks and Code Expansion"
- "Channel Optimization"
- "Pipelining"

Clocked Tasks

Code integration gets more complex when a task is extracted from an instance inside an activate block or a state machine. In that case, the generated task is not active at every step and its parent task only sends/receives when the task is active.

Example: Consider the following where the tasks are only active when the input is positive

```
function F1(i1:int32) returns (o1:int32)
  o1 = i1*i1;

function F2(i2:int32) returns (o2:int32)
  o2 = (i2-1)*(i2+1);

node root_clocked (i:int32) returns (o:int32 last =0)
  activate if i>0 then
    o = #pragma kcg C: task F1_task #end F1(i)
      - #pragma kcg C: task F2_task #end F2(i);
else
returns ..;
```

The code generated for `F1_task` and `F2_task` is the same as presented in "Example of Generated Code" on page 884 in *SCADE Suite User Manual*. Here is the code generated for `root_clocked`:

```
/* root_clocked */
void root_clocked_1 (inC_root_clocked *inC , outC_root_clocked *outC)
{
   outC-> cb_clock = inC->i > kcg_lit_int32 (0);
   outC->o1 = outC->o;
   if (outC->cb_clock) {
      (* outC->F1_task_in_ch). i1 = inC->i;
      (* outC->F2_task_in_ch). i2 = inC->i;
   }
   else {
      outC->o = outC->o1;
   }
}

void root_clocked_2 (outC_root_clocked *outC)
{
   if (outC->cb_clock) {
       outC->o = (*outC->F1_task_out_ch).o1 - (*outC->F2_task_out_ch).o2;
   }
}
```

We can see that the context fields associated with the input (resp. output) channels are only read (resp. written) when their clock (`cb_clock` field) is active. It means the integration code should only perform the corresponding synchronization/communication at these instants:

```
/* call root_clocked_1 */
root_clocked_1 (&inCtx_root_clocked, &outCtx_root_clocked);
if ((&outCtx_root_clocked)->cb_clock == kcg_true) {
   CHANNEL_SEND(F2_task_in_ch);
   CHANNEL_SEND(F1_task_in_ch);
}
/* call root_clocked_2 */
if ((&outCtx_root_clocked)->cb_clock == kcg_true) {
   CHANNEL_RECV(F2_task_out_ch);
   CHANNEL_RECV(F1_task_out_ch);
}
root_clocked_2(&outCtx_root_clocked);
```

The clock of these channel fields can be found in the mapping file as other context fields like probes. The activation clock of a task is given relatively to its parent task in the mapping file. It means this clock is a context field in the parent task. When using dynamic scheduling (see "Dynamic Scheduling" on page 277), there is no need to consider the activation clock of tasks, since their execution is triggered by the availability of data on input channels. When using static scheduling (see "Sequential Scheduling" on page 278), a method must only be executed when the clock of the task is active. Here is the result in the example:

```
/* call root_clocked_1 */
root_clocked_1 (&inCtx_root_clocked, &outCtx_root_clocked);
/* call F2_task */
if ((&outCtx_root_clocked)->cb_clock == kcg_true) {
   F2_task((&outCtx_F2_task));
}
/* call F1_task */
if ((&outCtx_root_clocked)->cb_clock == kcg_true) {
   F1_task((&outCtx_F1_task));
}
/* call root_clocked_2 */
root_clocked_2(&outCtx_root_clocked);
```

In the general case, the global activation clock of a task is the conjunction of its clock with respect to its parent task and the global clock of its parent task. The clocks of tasks form a tree, whose root is the activation clock of the root task which is always true. A clock `c1` is a descendant of `c2` if `c1` is a sub-clock of `c2`, that is, `c1` is obtained by sampling `c2` on a certain condition. If several tasks are grouped in the same worker[1], then the activation clock of this worker can be defined as the common ancestor of the clocks of the tasks in the worker. If a task is slower than the clock of the worker, its execution must be conditioned by the clock of the task with respect to the clock of the worker. Otherwise, the slower task blocks, waiting for its inputs, and prevents the execution of faster tasks in the worker. The sequential scheduling is a

1. A worker represents a set of tasks executed sequentially in the same core/thread/OS task depending on the target.

particular case of this general principle, where the clock of the (unique) worker is the base clock of the model and all channels are local. There can be an issue if this local clock requires access to a context in a different worker, which may not be possible/desirable.

Tasks and Code Expansion

An operator extracted into a task can be marked as expanded for code generation. In that case, the operator instance is expanded inside the generated task. For instance, if `F1` is expanded, then the code generated for `F1_task` is:

```
void F1_task ( outC_F1_task * outC)
{
  (*outC -> F1_task_out_ch). o1 = (*outC -> F1_task_in_ch). i1 *
   (*outC -> F1_task_in_ch). i1;
}
```

Channel Optimization

By default, each task has one input channel containing all the inputs of the operator, and one output channel containing all its outputs. The task receives and sends data to its parent task, which is the task in which the operator calling the task is executed (typically the root task if tasks are not nested).

Yet, if the output of one channel (or part of it) is directly used by another task, Multicore Code Generator ensures that both tasks communicate directly (instead of communicating via the parent task). This optimization also avoids communicating constant values (the task directly uses the constant). It can handle the case where the parent task only performs a simple operation (*e.g.,* arithmetics operation) on the output of a task before sending the result to

another task. In that case, the latter task receives directly from the former task and performs the simple operation inside the task. For instance, in the following example:

```
x = #pragma kcg C: task F1_task #end F1(i);
y = #pragma kcg C: task F2_task #end F2(i);
.. = #pragma kcg C: task G_task #end G(x+y, 3);
```

The code generated for `G_task` is:

```
(*outC->G_task_out_ch).o = /* o=(G)/ */
   G((*outC->F1_task_out_ch).o1 + (*outC->F2_task_out_ch).o1,
kcg_lit_int32(3));
```

The second input is replaced by the corresponding constant and the task receives directly from `F1_task` and `F2_task` and sums their results locally. As a result, a task can have multiple input and output channels. There is one channel for each pair of tasks that communicate directly.

Pipelining

In some cases, it may not be possible to parallelize an existing SCADE Suite model because of computation sequentiality. A classic solution is to use pipelining and add memories between the different parts of the computation. This represents more steps to compute a result, but the different parts of the pipeline can be executed in parallel. Pipelining can be achieved in SCADE Suite models by adding a unit delay (`fby` operator) between different operators. This results in a channel with an initial token containing the initial value of the `fby` operator.

Example: Consider these two tasks separated by an “fby” operator

```
node root_pipeline (i:int32) returns (o:int32)
var x:int32;
let
  x = fby(#pragma kcg C:task F1_task #end F1(i); 1; 0);
  o = #pragma kcg C:task F2_task #end F2(x);
tel
```

A basic implementation involves `F1_task` sending its result to the root task, which stores it in the context to send it to `F2_task` at the next instant. Multicore Code Generator generates a more efficient KPN (see Figure 10.1) where `F1_task` communicates directly through a channel with an initial token. It means that the FIFO contains a value before executing tasks. For details on the implementation of such channels see "Implementation of Communication Channels" on page 265. At a given step, `F2_task` reads either the initial value or the value produced by `F1_task` at previous step, and `F1_task` produces a new value. The initial token of a channel is a constant defined in `kcg_consts.h` and linked with the channel in the mapping file.

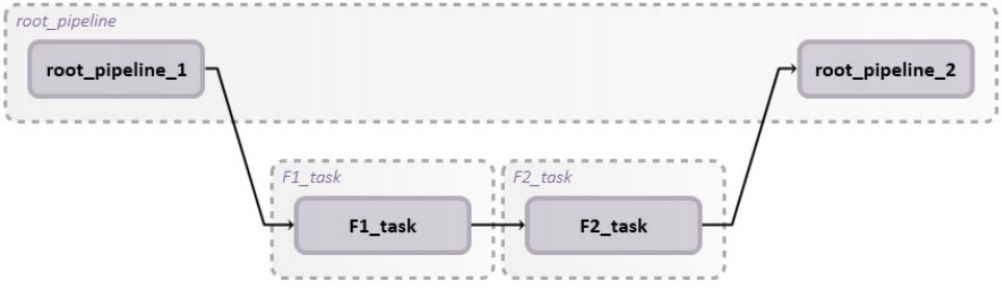


Figure 10.1: Efficient KPN generated for pipelining

Examples of Integration

Simple cases of integration illustrate below some basics of task scheduling.

- A *dynamic scheduling* where each task is executed in a different worker
- A *sequential scheduling* where all tasks are executed in the same worker
- A *classic list-scheduling* algorithm for allocating tasks to a multicore target

DYNAMIC SCHEDULING

The most straightforward way to implement the KPN is dynamic scheduling, done by the OS. This is done by creating a separate worker for each task. In a worker, methods can be executed in any order as long as the dependencies between methods are respected. A possible order is the one given by method numbering, which is also the order of methods in the mapping file.

Consider the following code for a `F2_task` task which has only one cycle method (Win32 semaphores are used for synchronization).

```
/* wait for the semaphore to be active (i.e. equal to 1) and then decrement it by one . */
#define CHANNEL_RECV (channel) \
   WaitForSingleObject ( channel ## _sem , INFINITE );
/* make the semaphore active (i.e. equal to 1) */
#define CHANNEL_SEND (channel) \
   ReleaseSemaphore (channel##_sem, 1, NULL);
/* Allocate a global variable and a semaphore for each channel */
/* F2_task_in_ch */
F2_task_in_ch_type F2_task_in_ch;
HANDLE F2_task_in_ch_sem ;
/* F2_task_out_ch */
F2_task_out_ch_type F2_task_out_ch;
HANDLE F2_task_out_ch_sem ;
...
DWORD WINAPI thread_2 (PVOID arg)
{
  size_t i;
  outC_F2_task outCtx_F2_task;
  /* initialize context fields for channels */
  (&outCtx_F2_task)-> F2_task_in_ch = & F2_task_in_ch;
  (&outCtx_F2_task)-> F2_task_out_ch = & F2_task_out_ch;
  F2_task_reset ((&outCtx_F2_task));
  while (1) {
    /* call F2_task */
    CHANNEL_RECV (F2_task_in_ch);
    F2_task ((&outCtx_F2_task));
    CHANNEL_SEND (F2_task_out_ch);
  }
  return 0;
}
```

SEQUENTIAL SCHEDULING

Another way to implement the generated KPN is to execute all tasks in the same worker. To compute a valid schedule, one can build a dependency graph of the different methods, where the nodes are the generated methods and there exists an edge from m_1 to m_2 if:

- m_2 receives from a channel written by m_1 without initial token
- m_1 and m_2 are methods of the same task and m_2 depends on m_1 (this information is available in the mapping file)

Then, any topological sort of dependency graph is a valid execution order. All channels can be implemented using local variables.

Consider the following generated code:

```
/* Allocate a global variable for each channel */
/* F2_task_in_ch */
F2_task_in_ch_type F2_task_in_ch ;
/* F2_task_out_ch */
F2_task_out_ch_type F2_task_out_ch ;
/* F1_task_in_ch */
F1_task_in_ch_type F1_task_in_ch ;
/* F1_task_out_ch */
F1_task_out_ch_type F1_task_out_ch ;

int main ()
{
   outC_F1_task outCtx_F1_task ;
   outC_root outCtx_root ;
   inC_root inCtx_root ;
   outC_F2_task outCtx_F2_task ;
   size_t i;
   /* initialize context fields for channels */
  (&outCtx_F1_task)->F1_task_in_ch = &F1_task_in_ch;
  (&outCtx_F1_task)->F1_task_out_ch = &F1_task_out_ch;
  (&outCtx_root)->F1_task_in_ch = &F1_task_in_ch;
  (&outCtx_root)->F1_task_out_ch = &F1_task_out_ch;
  (&outCtx_root)->F2_task_in_ch = &F2_task_in_ch;
  (&outCtx_root)->F2_task_out_ch = &F2_task_out_ch;
  (&outCtx_F2_task)->F2_task_in_ch = &F2_task_in_ch;
  (&outCtx_F2_task)->F2_task_out_ch = &F2_task_out_ch;

   /* execute main program loop */
   F1_task_reset ((&outCtx_F1_task));
   root_reset (&outCtx_root);
   F2_task_reset ((&outCtx_F2_task));

   while (1) {
     /* set inputs in inCtx_root */
     ...
     /* call root_1 */
     root_1 (&inCtx_root, &outCtx_root);
     /* call F2_task */
     F2_task ((&outCtx_F2_task));
     /* call F1_task */
     F1_task ((&outCtx_F1_task));
     /* call root_2 */
     root_2 (&outCtx_root);
     /* read outputs in outCtx_root */
     ...
   }
   return 0;
}
```

SEQUENTIAL SCHEDULING AND INITIAL TOKENS

The sequential scheduling algorithm described above does not consider channels with initial token. This may result in unnecessary copies, if the sender of such channel is scheduled before the receiver, even though it could have been scheduled before. This can be avoided by adding new edges in the dependency graph: an edge is added from m_1 to m_2 if m_1 receives from a channel written by m_2 without initial token. This ensures that the receiver is executed before the sender. However, adding such edges may introduce cycles in the dependency graph. If the edge cannot be added without creating a cycle, then it means that it is necessary to do a copy for the associated channel.

MULTICORE SCHEDULING

Let us apply a classic algorithm for task allocation to the KPN generated by MCG and illustrate its specificities.

Consider a task graph, which is a directed (acyclic) graph where the nodes are tasks, which have an associated cost (w_i is the cost of task t_i), and there exists an edge from t_1 to t_2 if t_2 depends on t_1 (edges can also have an associated cost, but there are ignored here).

We use here a list scheduling algorithm. The principle of this family of algorithms is to keep a queue of schedulable task, sorted by priority. The task with the highest priority is allocated to the core which allows the earliest start time for the task. Some tasks then become schedulable and are added to the queue, sorted by priority, and so on, until all tasks are allocated. The methods differ in the choice of the priority function.

In the HLFET heuristic (Highest Level First with Estimated Time), the priority of a task is its so-called static level (SL), which is the length of a longest path from the task to an exit node (that is, a node without successors in the task graph). This method ignores the cost of communications.

MCG application:

- Build a method graph, which is the reverse of the dependency graph used for sequential scheduling.
- Compute the static level of all methods, using for instance a sequential WCET as cost for each method.

$$SL(m_i) = w_i + \max_{m_j \in succ(m_i)} SL(m_j)$$

- Initialize priority queue with the methods having no dependencies, sorted by static level.
- While the queue is not empty:
  - Pop the method with the highest priority and allocate it:
    - If another method of the same task was already allocated, then the method must be allocated to the same core.
    - Otherwise, it is put in the core which allows the earliest start time. This can be computed from the end time of dependencies of the method and the end time of the last method allocated to each core.
  - Add in the priority queue all the methods whose dependencies are all scheduled.

To sum up, the specificities of the MCG-generated KPN are:

- The additional dependencies between methods of the same task are handled through the dependency graph: a task can only be scheduled when all its dependencies are scheduled. It is straightforward to add new dependencies.
- The methods of the same task must be allocated to the same core, which restricts some choices in the scheduling algorithm.

11 /Code Generators Command Line Options

This chapter explains how you can launch directly the SCADE Suite KCG or Multicore Code Generator (MCG) executables and what are the available command line options for.

Note

The content of this chapter is intended for an advanced use of code generation. In the framework of SCADE Suite KCG certification activities, this information is extracted and compiled from *SCADE Suite KCG Tool Operation Requirements (TOR)* to ease access directly from online user documentation.

- "KCG/MCG Execution and Compatible Options"
- "List of Available KCG/MCG Options"
- "Description of General Options"
- "Description of Optimization Options"
- "Description of Debug Options"
- "Description of Expansion Options"
- "Description of I/O Separation Options"
- "Description of Configuration Options"
- "Description of Additional Options"

Note

SCADE Suite KCG supports Ada and C code generation. Multicore Code Generator supports only C code generation.

KCG/MCG Execution and Compatible Options

The command lines used to execute KCG/MCG require options compatible with the execution objective. In case of incompatible options in a command line, KCG/MCG display an error message and stops any code or file generation.

Note

KCG/MCG also displays an error message when an option occurs several times in command line (and if no error is raised by other options).

With the appropriate set of options described in Table 11.2, you can run KCG/MCG for the following objectives:

- *Help*: displaying help about command lines
- *C generation*: generating C code from input model
- *Ada generation*: generating Ada code from input model (only KCG)
- *Checks*: running syntactic/semantic checks on input model without generating code

Provided one combines compatible options, KCG/MCG perform the specific tasks and produces different output files or information.

Table 11.1: KCG/MCG execution and compatible options

Objective	Options	Compatibility	Description
Help	-h or -help	no option	Help about the command line and options on the standard output. No file is generated. No other options.

Table 11.1: KCG/MCG execution and compatible options (Continued)

Objective	Options	Compatibility	Description
C generation	-target C -root <roots>	all C options no -h/-help	C code, logging and traceability info are generated. List of generated files in "Generated Files" on page 187 and list of options in Table 11.2.
Ada generation	-target Ada (KCG only) -root <roots> (KCG only)	all Ada options no -h/-help	Ada code and logging information are generated. List of generated files in "Generated Files" on page 187 and list of options in Table 11.2.
Checks	-root <roots>	all expansion options: -exp/-expall/-noexp	Syntactic and semantic checks are run only when -target option is not set in the command line. No code generation.

PATH TO SCADE ELEMENTS

The definition of SCADE elements when setting option arguments requires the use of fully qualified paths. A path represents the location of a Scade element in a model referring to the named constructions where it belongs. The *fully qualified path* of a Scade element indicates which elements are crossed to reach the element, starting from the top-level namespace of the model and not taking into account any `open` directives.

A path is built from the concatenation of the following elements:

- package identifier followed by `::`
- operator, type, constant, sensor, enumerator identifier, local flow, or signal followed by `/`

- state machine, state, or control block identifier, or control block branch name followed by " : "
- transition priority between < and >

The final optional element is an operator, type, enumerator, constant, sensor or variable identifier, followed by = in the case of equations (the variable is then the first left-hand side variable of the equation).

Note the trailing separator is optional: both `P1::P2::N/` and `P1::P2::N` are valid fully qualified paths for operator `N` contained in package `P2`, itself contained in package `P1`.

List of Available KCG/MCG Options

A number of options are available for generating code in command line from KCG/MCG executables. You find the list of available options summarized by category with a short description of their action in the following table.

Table 11.2: KCG/MCG command line options

Options	Category	Target Language		Description
		C	Ada	
-debug / -observe / -no_observe	Debug	X	X	Sets selected model elements as observable (*i.e.* accessible via the context of the operator they belong to) with sufficient accessible clock info to determine when their value is active
-exp /-noexp / -expall	Expansion	X	X	Expands the generated code of specified user-defined operators. As a consequence, modifies the amount of generated files. Exclusive options. By default, code generation performs no expansion on any operator.

Table 11.2: KCG/MCG command line options (Continued)

Options	Category	Target Language		Description
		C	Ada	
-global_root_context	Configuration	X		Modifies the global data types and architecture of execution of the C code
-globals_prefix	Configuration	X		Adds a prefix to all global names of a model
-header	Configuration	X	X	Adds at the beginning of each generated file a content header before the file header generated by the KCG (see "File Header Banner" on page 231)
-imported_root_package	Configuration		X	Generates all Ada imported code within a specified package
-input_threshold	Optimization	X		Sets the maximum number of arguments of the cycle function generated for a scade operator until a specific structure is created to collect them
-keep_asserts	Configuration	X	X	Calls definitions of `kcg_assume` and `kcg_guarantee` provided by user for each assume and guarantee assertion in the model
-name_length	Configuration	X		Limits the length of C identifiers
-no_timestamp	Configuration	X	X	Generates KCG files without file generation date in headers/footers
-root	General	X	X	Defines the root operator(s) of the input model
-probes	Configuration	X	X	Keeps all C or Ada variables corresponding to probes in input model in the contexts corresponding to node calls

Table 11.2: KCG/MCG command line options (Continued)

Options	Category	Target Language		Description
		C	Ada	
-root_package	Configuration		X	Generates all Ada code within a specified Ada package
-separate_io / -separate_io_all / -noseparate_io	I/O Separation	X	X	Specifies whether operators are marked or not for separate I/O during code generation
-short_circuit	Configuration	X	X	Specifies no bitwise C operator appears in the generated code
-significance_length	Configuration	X		Differentiates generated identifiers by their first significant characters
-state_vector	Configuration	X		Generates the memory state of the system in header files of the root operator and of any other operator influenced by the memory of the designed system
-static	Optimization	X		Adds the `static` C class storage specifier in front of C local variables
-target	General	X	X	Specifies the language generated (C or Ada)
-target_dir	General	X	X	Specifies the output directory of generated files (except kcg.log and kcg_files.txt)
-use_types	Configuration		X	Generates each specified type with use type clause in Ada code
-user_config	Configuration	X		Includes the `.h` file specified at the beginning of the generated `kcg_types.h` file
-warn_error	General	X	X	Treats warnings as errors
-wrap_c_ops	Configuration	X		Encapsulates arithmetics and relational operators into C macros

Table 11.2: KCG/MCG command line options (Continued)

Options	Category	Target Language		Description
		C	Ada	
-xml_basename	Additional option	X	X	Changes the default name of the `.scade` generated file

Description of General Options

The following KCG/MCG command line options are available.

-root (C and Ada)
specify root operator(s) for code generation

-target (C and Ada)
specify target code

-target_dir (C and Ada)
specify output directory for generated files

-warn_error (C and Ada)
modify the status of code generation warnings

"KCG General Options in UI Settings"

OPTION -ROOT

Option Format	`-root rootname1, rootname2,... (c and Ada)`
Code Generation	C and Ada
Argument Format	Node path: `PackageName::NodeName`
Description	This option takes valid user-defined operator path names that defines the **root operator(s)** of the current execution. Multiple operators are possible for C and Ada code generation. Specify paths to user-defined operators in comma-separated list (see 3.2 "Packages" from *Scade Language Reference Manual*).
Errors	An error is raised if: • the option does not appear in the command line • no argument is given to the option or not a path to exiting operator • the argument refers to an imported, polymorphic, parameterized, or non-valid user-defined operator name • an illegal root operator is specified
Mandatory	Yes

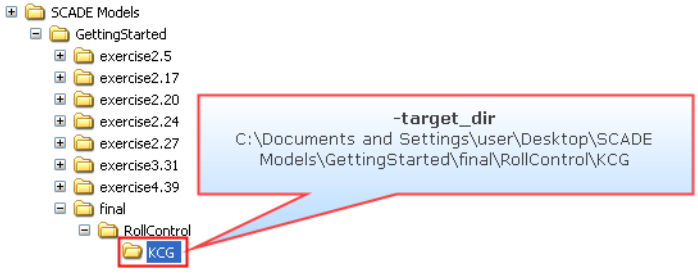
OPTION -TARGET

Option Format	`-target C` `-target Ada`
Code Generation	C and Ada
Argument Format	(Upper case) C or (string) Ada
Description	This option specifies that C or Ada code is generated. If not specified, checks are performed on the input model but no code is generated.
Errors	An error is raised if: • no argument is given to the option • an argument different from upper case C or Ada is given

OPTION -TARGET_DIR

Option Format	`-target_dir dirname`
Code Generation	C and Ada
Argument Format	Directory path
Description	This option specifies the output directory for all generated files for C/Ada generation (see "KCG/MCG Execution and Compatible Options"). By default, a "kcg" output directory is created as subdirectory in the directory used to launch KCG. If the directory does not exist, it is created with parent directories when needed.
Errors	An error is raised if no argument is given to the option

Example: Option `-target_dir` and argument in command line to generate code in KCG directory

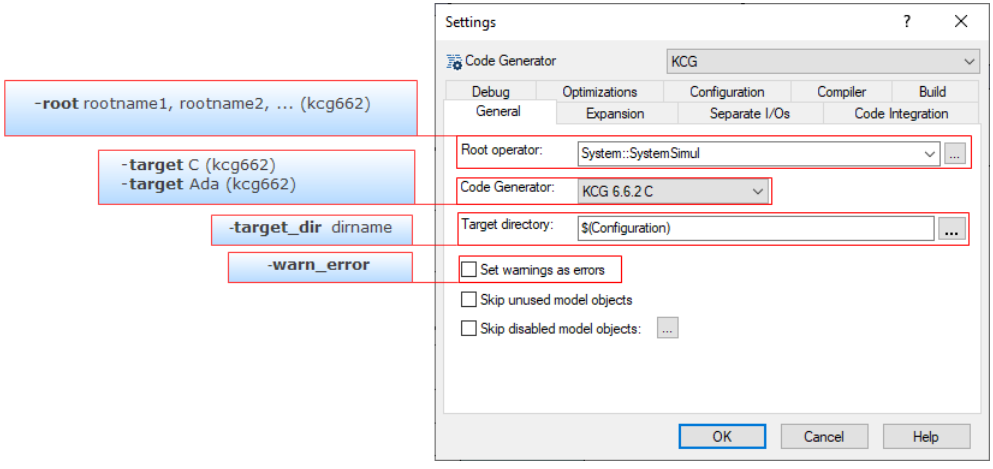


OPTION -WARN_ERROR

Option Format	`-warn_error`
Code Generation	C and Ada
Description	This option treats warnings as errors. If this option is set and there is no error but at least one warning, KCG stops with exit status 2 instead of 1 without generating code (for details about exit codes, see "Input and Output Files" on page 182). Also, a message is reported on the standard output and in the log file to remind the user that warnings are currently treated as errors. KCG generates only the following files: `kcg.log`, `kcg_files.txt`, and `kcg_xml_filter_out.scade` (or named according to -xml_basename option). **Note**: this option does not affect the format of warning messages, and the occurring warnings are still counted as warnings in the report.

KCG GENERAL OPTIONS IN UI SETTINGS

The general options available to run KCG/MCG in command line have an equivalent in the **General** tab on the Code Generator Settings window.



Description of Optimization Options

The following KCG/MCG command line options are available.

-input_threshold (C only)
set an input threshold for operators

-static (C only)
define local variables with static class storage specifier

"KCG Optimization Options in UI Settings"

In SCADE Suite KCG, generated code is always optimized except with `-debug`, `-observe`, and `-no_observe` options.

OPTION -STATIC

Option Format	`-static`
Code Generation	C
Description	This option defines local variables with a static C class storage specifier. When set, local variables are declared at global data level and have *static storage duration* - memory for these local variables is allocated when the program begins and is freed at termination. Stack size is reduced. In case of MCG, this option raises an error if tasks are generated.
Example	The code below illustrates the use of the -static option:

Example: Sample of generated C code with `-static` option set

```
/* $*************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T09:39:10
*************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RollRateCalculate_RollRate.h"

/** RollRateCalculate computes the aircraft roll rate due to stick input and adverse yaw effects. */
/* RollRate::RollRateCalculate */
kcg_real RollRateCalculate_RollRate(
  /* RollRate::RollRateCalculate::joystickCmd */kcg_real joystickCmd,
  /* RollRate::RollRateCalculate::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::RollRateCalculate::rightAdverseYaw */kcg_real rightAdverseYaw)
{
  /* pwlinear::LimiterSymmetrical::Lower_limit */
  static kcg_real Lower_limit_1;
  /* pwlinear::LimiterSymmetrical::Upper_limit */
  static kcg_real Upper_limit_1;
```

Local variable with static C class storage specifier

OPTION -INPUT_THRESHOLD

Option Format	`-input_threshold <n>`
Code Generation	C
Argument Format	Integer

Description	This option prevents the number of parameters of any function generated for non-imported operators to be greater than a number given as parameter of the option. If the number of inputs is greater than n, an input context is created containing all inputs. If adding the number of outputs as parameters still exceeds n, then an output structure containing all outputs is built.
	The prototype of the cycle function is:
	• Case of an operator requiring a context:
	  • if operator not marked for separate I/O (see Table 11.4): `extern void <operator_name> (inC_<operator_name> *inC, outC_<operator_name> *outC);`
	  • if operator marked for separate I/O (see Table 11.4) and if the number of outputs strictly greater than 0 and n-2: `extern void <operator_name> (inC_<operator_name> *inC, outS_<operator_name> *outS, outC_<operator_name> *outC); otherwise extern void <operator_name> (inC_<operator_name> *inC, outputs, outC_<operator_name> *outC);`
	• Case of a Scade function that returns a single scalar: `extern <output_type> <operator_name>(inC_<operator_name> *inC );`
	• In other cases:
	  • If the number of outputs is strictly greater than 0 and n-1: `extern void <operator_name>(inC_<operator_name> *inC , outS_<operator_name> *outS);`
	  • else: `extern void <operator_name>(inC_<operator_name> *inC , outputs);`

Errors	An error is raised if: • no argument is given to the option • the argument given to the option is neither positive nor an integer

Example: The input threshold is set to 2. The `RollRateCalculate` operator (from RollControl example) requires a number of inputs (3) superior to the input threshold defined (2). Below is a sample of generated code with `-input_threshold` option set: input structure created in the generated header file of the `RollRateCalculate` operator.

```
/* $**************** KCG Version 6.4 (build i21) ****************
** Command: kcg64.exe -config C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T09:48:56
*****************************************************************$ */
#ifndef _RollRateCalculate_RollRate_H_
#define _RollRateCalculate_RollRate_H_

#include "kcg_types.h"
#include "AdverseYaw_RollRate.h"

/* ========================  input structure  ====================== */
typedef struct {
  kcg_real /* RollRate::RollRateCalculate::joystickCmd */ joystickCmd;
  kcg_real /* RollRate::RollRateCalculate::leftAdverseYaw */ leftAdverseYaw;
  kcg_real /* RollRate::RollRateCalculate::rightAdverseYaw */ rightAdverseYaw;
} inC_RollRateCalculate_RollRate;

/* ======================  no output structure  ====================== */


/** RollRateCalculate computes the aircraft roll rate due to stick input and adverse yaw effects. */
/* RollRate::RollRateCalculate */
extern kcg_real RollRateCalculate_RollRate(inC_RollRateCalculate_RollRate *inC);

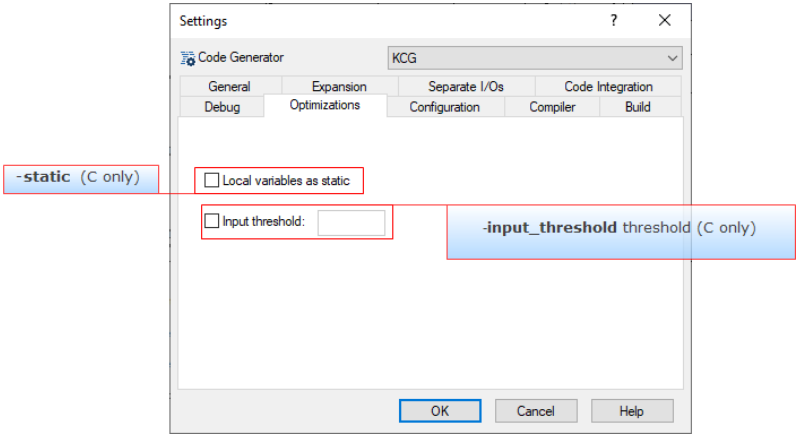
#endif /* _RollRateCalculate_RollRate_H_ */
/* $**************** KCG Version 6.4 (build i21) ****************
** RollRateCalculate_RollRate.h
** Generation date: 2015-08-11T09:48:56
*****************************************************************$ */
```

C structure created `inC_<operator_name>`

Cyclic function with reference to `inC_<operator_name>`

KCG OPTIMIZATION OPTIONS IN UI SETTINGS

The optimization options available to run KCG in command line have an equivalent in the **Optimizations** tab on the Code Generator Settings window.



Description of Debug Options

The following KCG/MCG command line options are available.

-debug (C and Ada)
decide to make all operators observable

-no_observe (C and Ada)
define list of non-observable operators

-observe (C and Ada)
define list of observable operators

"KCG Debug Options in UI Settings"

OPTION -DEBUG

Option Format	`-debug`
Code Generation	C and Ada
Description	This option specifies that all the operators in the model are observable. Observable operators are not optimized. See `-observe` for an explanation on observable operators.
Errors	An error is raised if the `-observe` or `-no_observe` option are present in the same command line

OPTION -NO_OBSERVE

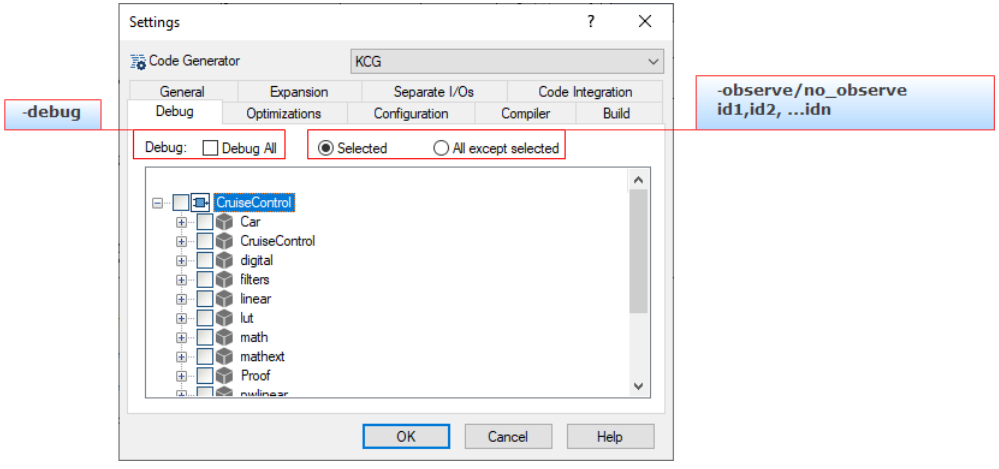
Option Format	`-no_observe operatorpathlist`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: PckgName1::OpName1, ..., PckgNameN::OpNameN
Description	This option specifies the list of operators you want to make non-observable. The operators out of the list are observable. Observable operators are not optimized. See `-observe` for an explanation on observable operators.
Errors	An error is raised if: • `-observe` or `-debug` options are present in same command line • one of the identifiers in the list is not found in the input files, is an imported operator • no argument is given

OPTION -OBSERVE

Option Format	**`-observe`** `operatorpathlist`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: `PckgName1::OpName1, ..., PckgNameN::OpNameN`
Description	This option sets the list of operators you want to make observable. Observable operators are not optimized. The operators out of the list are not observable. With this option, local variables with C/Ada counterpart, probes, and assertions are added in the generated code to their operator context. The following elements are also observable: state and transition variables, clocks of clocked blocks, and clocks of observed data. If an observable operator is not expanded, the following objects are added to the C context of the operator: • local variables with a C/Ada counterpart • C counterpart of the operator probes if -probes option is set • local variables used to define assertions • C/Ada counterpart of state and transition variables • clocks of clocked blocks • clocks of observed variables If the operator is expanded, the above elements are added to the context of the operator in which it is inlined. This applies even if the operator in which an observed operator is inlined is not itself observed.
Errors	An error is raised if: • `-no_observe` or `-debug` options are present in same command line • one of the identifiers in the list is not found in the input files, is an imported operator • no argument is given

KCG DEBUG OPTIONS IN UI SETTINGS

The debug options available to run KCG/MCG in command line have an equivalent in the **Debug** tab on the Code Generator Settings window.



Description of Expansion Options

The following KCG/MCG command line options are available.

-exp (C and Ada)
define list of expanded operators

-noexp (C and Ada)
define list of non-expanded operators

-expall (C and Ada)
decide to expand all operators

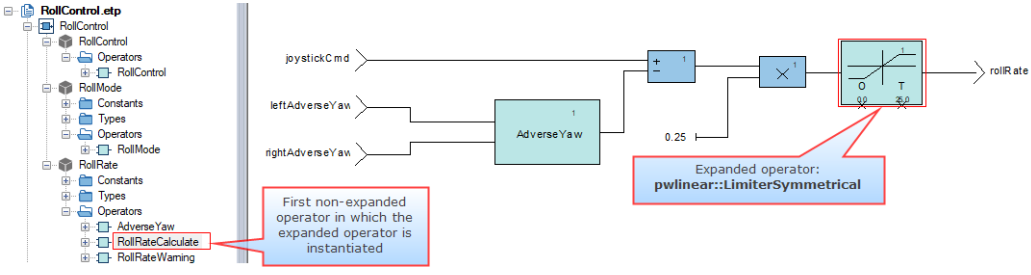
"KCG Expansion Options in UI Settings"

OPTION -EXP

Option Format	`-exp [pathlist]`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: `PckgName1::OpName1, ..., PckgNameN::OpNameN` Path names are described in section 3.2 "Packages" from *Scade Language Reference Manual*. The following operators are expanded: • Operators not present in list but with `expand` pragma attached to • Operators present in list (even if a `noexpand` pragma is attached to)
Description	This option specifies the operators to be expanded. For *expanded operators*, the corresponding C code is inlined inside the generated code of the first non-expanded operator where they are instantiated. For *non-expanded operators* (not present in the list and no expand pragma attached to), two files are generated for each monomorphic instance in the cone of influence of the operator (see "Generated Files" on page 187). By default, operators are not expanded.

Errors	An error is raised if: • one of the identifiers in the list is not found in the input • the root operator appears in the input list of identifiers • `-noexp` or `-expall` is also used If a root operator is marked for expansion (see Table 11.3), a warning is raised in one of the following conditions: • `-exp` option is set and the fully qualified path of the root operator is in the list of expanded operators • `expand` pragma attached to the root operator

Example: The figure below shows the data-flow representation of the `RollRateCalculate` operator containing the operator to be expanded (`LimiterSymmetrical`)



The code corresponding to the `LimiterSymmetrical` operator is inlined in the generated code of the `RollRateCalculate` operator (see above). Below are samples of code generated for the `RollRateCalculate` operator: expanded `AdverseYaw` operator, expanded `LimiterSymmetrical` operator (left), and non-expanded `LimiterSymmetrical` operator (right).

For C code:

```c
/* $*************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config
C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T10:02:34
*************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RollRateCalculate_RollRate.h"

/** RollRateCalculate computes the aircraft roll rate due to stick input and
adverse yaw effects. */
/* RollRate::RollRateCalculate */
kcg_real RollRateCalculate_RollRate(
  /* RollRate::RollRateCalculate::joystickCmd */kcg_real joystickCmd,
  /* RollRate::RollRateCalculate::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::RollRateCalculate::rightAdverseYaw */kcg_real
rightAdverseYaw)
{
  /* pwlinear::LimiterSymmetrical::Lower_limit */ kcg_real Lower_limit_1;
  /* pwlinear::LimiterSymmetrical::Upper_limit */ kcg_real Upper_limit_1;
  /* RollRate::RollRateCalculate::_L6 */ kcg_real _L6;
  /* RollRate::RollRateCalculate::rollRate */ kcg_real rollRate;

  _L6 = (joystickCmd - (leftAdverseYaw - rightAdverseYaw) * 0.1) * 0.25;
  Upper_limit_1 = 25.0 + 0.0;
  Lower_limit_1 = 0.0 - 25.0;
  /* 5 */ if (_L6 >= Upper_limit_1) {
    rollRate = Upper_limit_1;
  }
  else /* 8 */ if (_L6 <= Lower_limit_1) {
    rollRate = Lower_limit_1;
  }
  else {
    rollRate = _L6;
  }
  return rollRate;
}

/* $*************** KCG Version 6.4 (build i21) ***************
** RollRateCalculate_RollRate.c
** Generation date: 2015-08-11T10:02:34
*************************************************************$ */
```

C Code of the expanded operator is inlined in the generated C code of the RollRateCalculate operator

```c
/* $*************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config
C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T10:20:54
*************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "RollRateCalculate_RollRate.h"

/** RollRateCalculate computes the aircraft roll rate due to stick input and
adverse yaw effects. */
/* RollRate::RollRateCalculate */
kcg_real RollRateCalculate_RollRate(
  /* RollRate::RollRateCalculate::joystickCmd */kcg_real joystickCmd,
  /* RollRate::RollRateCalculate::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::RollRateCalculate::rightAdverseYaw */kcg_real rightAdverseYaw)
{
  /* RollRate::RollRateCalculate::rollRate */ kcg_real rollRate;

  rollRate = /* 1 */
    LimiterSymmetrical_pwlinear_real(
      (joystickCmd - (leftAdverseYaw - rightAdverseYaw) * 0.1) * 0.25,
      0.0,
      25.0);
  return rollRate;
}

/* $*************** KCG Version 6.4 (build i21) ***************
** RollRateCalculate_RollRate.c
** Generation date: 2015-08-11T10:20:54
*************************************************************$ */
```

C Code of the same operator in a non-expanded configuration

For Ada code:

```
-- $**************** KCG Version 6.3 (build i1) *****************
-- ** Command: kcg63.exe -config C:/Program Files/Esterel Technologies/Esterel
-- SCADE 6.3/examples/GettingStarted/final/RollControl/KCG/config.txt
-- ** Generation date: 2011-09-26T10:35:17
-- *************************************************************$
package body RollRate
-- RollRate
is
  --# accept F, 501, all, "SCADE KCG Generated" ;
  --# accept F, 23,  all, "SCADE KCG Generated" ;
  --# accept F, 504, all, "SCADE KCG Generated" ;
  --# accept F, 602, all, "SCADE KCG Generated" ;

  --- RollRateWarning computes the left warning and right warning alarms
  --- according to the plane roll rate.

  -- RollRate::RollRateWarning
  procedure RollRateWarning
  (-- RollRate::RollRateWarning::rollRate
   rollRate_1 : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateWarning::leftWarning
   leftWarning : out Boolean;
   -- RollRate::RollRateWarning::rightWarning
   rightWarning : out Boolean)
  is

  begin
    rightWarning := rollRate_1 > kRollRateWarning.right;
    leftWarning := rollRate_1 < kRollRateWarning.left;
  end RollRateWarning;

  --- AdverseYaw represents simplified coupling effects between roll and yaw.

  -- RollRate::AdverseYaw
  procedure AdverseYaw
  (-- RollRate::AdverseYaw::leftAdverseYaw
   leftAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::AdverseYaw::rightAdverseYaw
   rightAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::AdverseYaw::rollCoupling
   rollCoupling : out Kcg_Config.Kcg_Real)
  is

  begin
    rollCoupling := (leftAdverseYaw - rightAdverseYaw) * 0.1;
  end AdverseYaw;

  --- RollRateCalculate computes the aircraft roll rate due to stick input and
  --- adverse yaw effects.

  -- RollRate::RollRateCalculate
  procedure RollRateCalculate
  (-- RollRate::RollRateCalculate::joystickCmd
   joystickCmd : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::leftAdverseYaw
   leftAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::rightAdverseYaw
   rightAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::rollRate
   rollRate_1 : out Kcg_Config.Kcg_Real)
  is
    -- pwlinear::LimiterSymmetrical::Lower_limit
    Lower_limit_1 : Kcg_Config.Kcg_Real ;
    -- pwlinear::LimiterSymmetrical::Upper_limit
    Upper_limit_1 : Kcg_Config.Kcg_Real ;
    -- RollRate::RollRateCalculate::_L6
    xL6 : Kcg_Config.Kcg_Real ;
  begin
    -- 1
     AdverseYaw(leftAdverseYaw, rightAdverseYaw, Lower_limit_1);
    xL6 := (joystickCmd - Lower_limit_1) * 0.25;
    Lower_limit_1 := 0.0 - 25.0;
    Upper_limit_1 := 25.0 + 0.0;
    if (xL6 >= Upper_limit_1) then
      rollRate_1 := Upper_limit_1;
    else
      if (xL6 <= Lower_limit_1) then
        rollRate_1 := Lower_limit_1;
      else
        rollRate_1 := xL6;
      end if;
    end if;
  end RollRateCalculate;

end RollRate;

-- $**************** KCG Version 6.3 (build i1) *****************
-- ** rollrate.adb
-- ** Generation date: 2011-09-26T10:35:17
-- *************************************************************$
```

Ada code of the expanded operator is inlined in the generated Ada code of the RollRateCalculate operator

```
-- $**************** KCG Version 6.3 (build i1) *****************
-- ** Command: kcg63.exe -config C:/Program Files/Esterel Technologies/Esterel
-- SCADE 6.3/examples/GettingStarted/final/RollControl/KCG/config.txt
-- ** Generation date: 2011-09-26T13:16:33
-- *************************************************************$
with pwlinear;

package body RollRate
-- RollRate
is
  --# accept F, 501, all, "SCADE KCG Generated" ;
  --# accept F, 23,  all, "SCADE KCG Generated" ;
  --# accept F, 504, all, "SCADE KCG Generated" ;
  --# accept F, 602, all, "SCADE KCG Generated" ;

  --- RollRateWarning computes the left warning and right warning alarms
  --- according to the plane roll rate.

  -- RollRate::RollRateWarning
  procedure RollRateWarning
  (-- RollRate::RollRateWarning::rollRate
   rollRate_1 : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateWarning::leftWarning
   leftWarning : out Boolean;
   -- RollRate::RollRateWarning::rightWarning
   rightWarning : out Boolean)
  is

  begin
    rightWarning := rollRate_1 > kRollRateWarning.right;
    leftWarning := rollRate_1 < kRollRateWarning.left;
  end RollRateWarning;

  --- AdverseYaw represents simplified coupling effects between roll and yaw.

  -- RollRate::AdverseYaw
  procedure AdverseYaw
  (-- RollRate::AdverseYaw::leftAdverseYaw
   leftAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::AdverseYaw::rightAdverseYaw
   rightAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::AdverseYaw::rollCoupling
   rollCoupling : out Kcg_Config.Kcg_Real)
  is

  begin
    rollCoupling := (leftAdverseYaw - rightAdverseYaw) * 0.1;
  end AdverseYaw;


  --- RollRateCalculate computes the aircraft roll rate due to stick input and
  --- adverse yaw effects.

  -- RollRate::RollRateCalculate
  procedure RollRateCalculate
  (-- RollRate::RollRateCalculate::joystickCmd
   joystickCmd : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::leftAdverseYaw
   leftAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::rightAdverseYaw
   rightAdverseYaw : in Kcg_Config.Kcg_Real;
   -- RollRate::RollRateCalculate::rollRate
   rollRate_1 : out Kcg_Config.Kcg_Real)
  is
    tmp : Kcg_Config.Kcg_Real ;
  begin
    -- 1
     AdverseYaw(leftAdverseYaw, rightAdverseYaw, tmp);
    -- 1

    pwlinear.LimiterSymmetrical_Real((joystickCmd - tmp) * 0.25,
                                     0.0,
                                     25.0,
                                     rollRate_1);
  end RollRateCalculate;

end RollRate;

-- $**************** KCG Version 6.3 (build i1) *****************
-- ** rollrate.adb
-- ** Generation date: 2011-09-26T13:16:33
-- *************************************************************$
```

Ada code of the same operator in a non-expanded configuration

OPTION -EXPALL

Option Format	`-expall`
Code Generation	C and Ada
Description	This option specifies that all operators but the root operator are expanded during code generation. Any *non-root operator* is expanded for causality analysis and expanded in the generated code if marked for expansion (see Table 11.3 below) Any *root operator* is expanded for causality analysis (and not expanded in the generated code) if marked for expansion. See `-exp` option for an explanation on expanded operators.
Errors	An error is raised if `-noexp` or `-exp` is in the same command line as `-expall`

Table 11.3: Operators marked for expansion

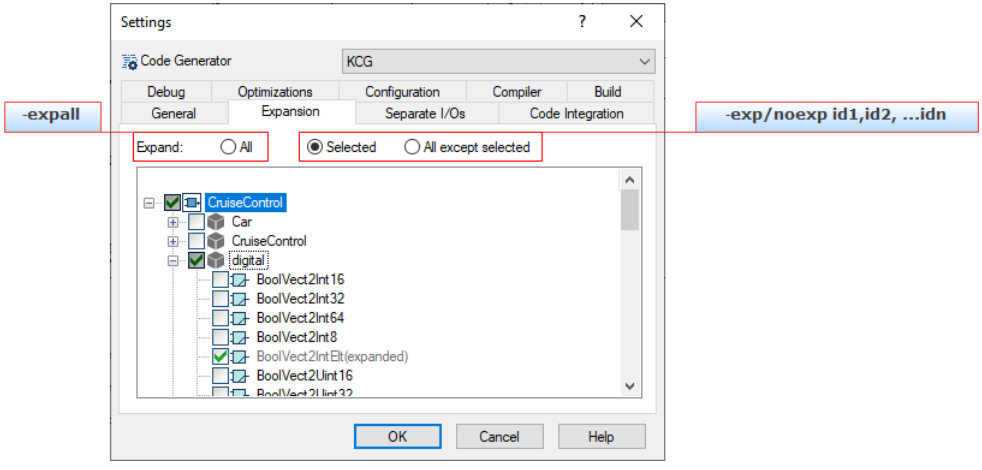
	no pragma	expand	noexpand
-exp and operator path in option list	Yes	Yes	Yes
-exp and no path in option list	No	Yes	No
-noexp and operator path in option list	No	No	No
-noexp and no path in option list	Yes	Yes	No
-expall	Yes	Yes	No
no expansion option	No	Yes	No

OPTION -NOEXP

Option Format	`-noexp [pathlist]`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: `PckgName1::OpName1, ..., PckgNameN::OpNameN` Path names are described in section 3.2 "Packages" from *Scade Language Reference Manual*. The following operators are not expanded: • Operators not present in list but with `noexpand` pragma attached to • Operators present in list, even if an `expand` pragma is attached to
Description	This option specifies the operators that you do not want to be expanded. See `-exp` option for an explanation on expanded and non-expanded operators and also for example. By default, operators are not expanded.
Errors	An error is raised if: • one of the identifiers in the list is not found in the input files • the root operator appears in the input list of identifiers • -exp or -expall is also used

KCG EXPANSION OPTIONS IN UI SETTINGS

The expansion options available to run KCG/MCG in command line have an equivalent in the **Expansion** tab on the Code Generator Settings window.



Description of I/O Separation Options

The following KCG/MCG command line options are available.

-separate_io (C and Ada)
list operators generated with separate I/O

-no_separate_io (C and Ada)
list operators generated without separate I/O

-separate_io_all (C and Ada)
generate all operators with separate I/O

"KCG Separate I/O Options in UI Settings"

OPTION -SEPARATE_IO

Option Format	`-separate_io [pathlist]`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: `PckgName1::OpName1, ..., PckgNameN::OpNameN` Path names are described in section 3.2 "Packages" from *Scade Language Reference Manual*. The following operators are marked for separate I/O: • Operators not present in list but `separate_io` pragma attached to • Operators present in list (even if a `no_separate_io` pragma is attached to)
Description	This option specifies the list of operators marked with separate I/O during code generation with respect to the usage of pragmas or not (see Table 11.4 below). See details about -separate_io pragma.
Errors	An error is raised if: • the path to an operator is not valid • the path to an imported operator is given

OPTION -SEPARATE_IO_ALL

Option Format	`-separate_io_all`
Code Generation	C and Ada
Description	This option specifies all operators are marked with separate I/O during code generation with respect to the usage of pragmas or not (see Table 11.4 below). See details about -separate_io and -no_separate_io pragmas.
Errors	An error is raised if `-no_separate_io` or `-separate_io` is present in the same command line as `-separate_io_all`

OPTION -NO_SEPARATE_IO

Option Format	`-no_separate_io [pathlist]`
Code Generation	C and Ada
Argument Format	Comma-separated list or operator identifiers: `PckgName1::OpName1, ..., PckgNameN::OpNameN` Path names are described in section 3.2 "Packages" from *Scade Language Reference Manual*. The following operators are not marked for separate I/O: • Operators not present in list but with `no_separate_io` pragma attached to • Operators present in list, even if `separate_io` pragma is attached to
Description	This option specifies which operators are not marked for separate I/O during code generation with respect to the usage of pragmas or not (see Table 11.4 below). Notice that imported operators can be compiled with separate I/O only using the `separate_io` pragma. See details about -no_separate_io pragma.
Errors	An error is raised if: • the path to an operator is not valid • the path to an imported operator is given

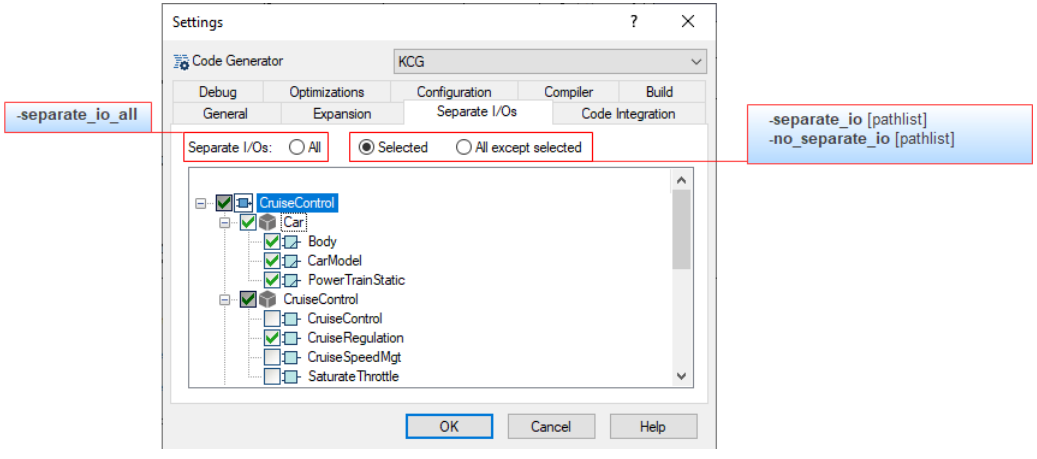
Table 11.4: Operators marked for separate I/O

	no pragma	separate_io	no_separate_io
-separate_io and operator path in option list	Yes[1]	Yes	Yes
-separate_io and no path in option list	No	Yes	No
-no_separate_io and path in option list	No	No	No
-no_separate_io and no path in option list	Not Imp[2]	Yes	No
-separate_io_all	Not Imp	Yes	No
no option	No	Yes	No

1. Yes (resp. No) stands for the operator is (resp. is not) marked for separate I/O.
2. Imp (resp. Not Imp) stands for the operator is marked for separate I/O if it is imported (resp not imported).

KCG SEPARATE I/O OPTIONS IN UI SETTINGS

The I/O separation options available to run KCG/MCG in command line have an equivalent in the **Separate I/Os** tab on the Code Generator Settings window.



Description of Configuration Options

The following KCG/MCG command line options are available.

-global_root_context (C only)
generate the context as global

-globals_prefix (C only)
generate all global identifiers with a prefix

-header (C and Ada)
insert header at beginning of generated files

-imported_root_package (Ada only)
specify package path for Ada imported code

-keep_asserts (C and Ada)
generate macro on assertions

-name_length (C only)
control the length of generated variables

-no_timestamp (C and Ada)
do not print generation date in generated files (KCG 6.6.4 only)

-probes (C and Ada)
keep probes in generated code

-root_package (Ada only)
specify package path for Ada code generation

-short_circuit (C and Ada)
generate Boolean operators without C bitwise operators

-significance_length (C only)
control name length of KCG-generated identifiers

-state_vector (C only)
generate the memory state of the system

-use_types (Ada only)
generate Scade types with `use_type` clauses

-user_config (C only)
include header file into generated `kcg_types.h` file

-wrap_c_ops (C only)
wrap C operators

"KCG Configuration Options in UI Settings"

OPTION -GLOBAL_ROOT_CONTEXT

Option Format	`-global_root_context`
Code Generation	C

Description	This option modifies the global data types and architecture of execution of the generated C code:
	• operator inputs/outputs and context variables of root operator are C global variables
	• the `inC_<root_name>` structure is not generated
	• the outputs do not belong to the `outC_<root_name>` context structure
	• the state vector structure and copy macro are not generated
	If this option is set, KCG/MCG generate for each root operator the `<root_name>_inputs.c`, `<root_name>_outputs.c` and `<root_name>_context.c` files that contain:
	• the inputs, outputs, and contexts of the root operator as global variables
	• the `<root_name>.h` file, with defined variables declared as external.
	If this option is set, the inputs, outputs, and context variables are not passed as arguments to root functions but as global C variables.
	• If the root operator requires a context, the global root context variable is declared as: `extern outC_<root_name_context> Ctxt_<root_name_variable>`
	• The root operator header file contains the following prototypes:
	• Initialization function: `extern void <root_name>_init<_path>(void)`
	• Reset function: `extern void <root_name>_reset<_path>(void)`
	• Cycle function: `extern void <root_name><_path>(void)`
	If this option and `-state_vector` are set, and if a root operator needs a C state, the state vector structure is generated and the save/restore functions use the global context and memorized output variables.
	The function prototypes are as follows:
	• Function for saving the global state to a state vector:
	`extern void kcg_save_SV_<root_name><_path>(SV_<root_name><_path> *SV);`
	• Function for restoring the global state from a state vector:
	`extern void kcg_load_SV_<root_name><_path>(SV_<root_name><_path> *SV);`
	
	... continued next page

Description (continued)	If this option is not set, the header file associated with a root operator contains the output context. If the output context is empty, it is not declared in the header file, since it is forbidden to declare empty structures in C.
	If this option is not set and if the root operator has at least one input and is not marked for separate I/O (see Table 11.4), the header file associated with a root operator contains an input context collecting its inputs.
	If `-global_root_context` option is not set:
	• The cycle function prototype of each root operator not marked for separate I/O (see Table 11.4)is:
	  • If there are no inputs and no context: extern void <root_name>(void);
	  • If there are no inputs and a context: extern void <root_name>(outC_<root_name>*outC);
	  • If there are inputs and no context: extern void <root_name>(inC_<root_name>*inC);
	  • If there are inputs and a context: extern void <root_name>(inC_<root_name>*inC , outC_<root_name>*outC);
	• The reset function prototype of each root operator is:
	  • If there is a context: extern void <root_name>_reset(outC_<root_name>*outC);
	  • If there is no context and the root operator is not marked for separate I/O (see Table 11.4): extern void <root_name>_reset(void)
	  • If no context and the root operator is marked for separate I/O (see Table 11.4): no reset function
	• The initialization function prototype of each root operator is:
	  • If there is a context: extern void <root_name>_init(outC_<root_name>*outC);
	  • If no context and the root operator is not marked for separate I/O (see Table 11.4): extern void <root_name>_init(void)
	  • If no context and the root operator is marked for separate I/O (see Table 11.4): no initialization function

OPTION -GLOBALS_PREFIX

Option Format	`-globals_prefix prefix`
Code Generation	C
Argument Format	String of characters <prefix>
Description	If this option is set, all global identifiers are prefixed by the <prefix> string: • sensors, types, constants • operators • any imported declaration The prefix is never used on the following identifiers: • identifiers with an attached name pragma • identifiers listed in Table 11.5
Errors	An error is raised if: • no argument is given to the option • the argument given to the option is not a valid C identifier

Table 11.5: Generated identifiers

kcg_char	kcg_bool	kcg_float32	kcg_float64	kcg_int8
kcg_int16	kcg_int32	kcg_int64	kcg_uint8	kcg_uint16
kcg_uint32	kcg_uint64	kcg_cond	kcg_incr	kcg_index
kcg_true	kcg_false	kcg_assume	kcg_guarantee	kcg_assign
kcg_assign_struct	kcg_assign_array	kcg_and	kcg_or	kcg_xor
kcg_not	kcg_eq	kcg_eq_bool	kcg_eq_char	kcg_size
kcg_diff	kcg_diff_bool	kcg_diff_char	kcg_mod_intN	kcg_mod_uintN
kcg_mod_size	kcg_lit_size	kcg_plus_size	kcg_minus_size	kcg_uminus_size
kcg_eq_size	kcg_lt_size	kcg_le_size	kcg_gt_size	kcg_*t*_to_*u* [1]
kcg_land_uintN	kcg_lnot_uintN	kcg_lor_uintN	kcg_lxor_uintN	kcg_lsl_uintN
kcg_lsr_uintN	kcg_eq_*t* [2]	kcg_diff_*t* [2]	kcg_lit_*t* [2]	kcg_ge_*t* [2]

Table 11.5: Generated identifiers

`kcg_gt_t` [2]	`kcg_le_t` [2]	`kcg_lt_t` [2]	`kcg_plus_t` [2]	`kcg_minus_t` [2]
`kcg_mult_t` [2]	`kcg_div_t` [2]	`kcg_uminus_t` [2]		

1. where t and u are distinct predefined numeric types
2. where t is intN, uintN, or floatN

OPTION -HEADER

Option Format	`-header filename`
Code Generation	C and Ada
Argument Format	File name (no restriction on extension)
Description	If this option is set, the content of the file <filename> is added on top of header banner of each generated file.
Errors	An error is raised if: • no argument is given to the option • the argument given to the option is not a valid file name • <filename> contains anything but valid C comments

OPTION -IMPORTED_ROOT_PACKAGE

Option Format	`-imported_root_package <package_path>`
Code Generation	Ada
Argument Format	Valid name for an Ada package.
Description	This option generates all Ada imported code within a specified package. For details about Ada packages, see "Package Hierarchy and SPARK Compliance" on page 234.
Errors	An error is raised if no valid name for an Ada package is given to the option

OPTION -KEEP_ASSERTS

Option Format	`-keep_asserts`
Code Generation	C and Ada
Description	If this option is set, and if the model contains **assume** (respectively **guarantee**) equations, the generated code contains the translation of these equations and one call per assertion to the `kcg_assume` (resp. `kcg_guarantee`) macro called with the value denoting the assertion expression. This value has `kcg_bool` as type. KCG/MCG does not provide a definition for these macros. For C code generation, users can provide the definitions in a file using `-user_config` option. For Ada code generation, users complete Ada generated code with definitions in `kcg_config` files.

OPTION -NAME_LENGTH

Option Format	`-name_length namelength`
Code Generation	C
Argument Format	Integer
Description	This option sets the maximum name length of KCG-generated identifiers to <namelength> characters. By default, <namelength> is set to 200. The names of the root operator's inputs and outputs are kept in the generated code when these names do not exceed the specified maximum identifiers length. The names of the root operator's probes are kept unless some of them coming from distinct subscopes collide: in this case, distinct C names are generated.
Errors	An error is raised if: • no argument is given to the option • no integer value given to the option • negative integer value given to the option • the integer value given to the option is lower than 16. In this case, KCG behaves as if "-name_length 16" was passed to the command line.

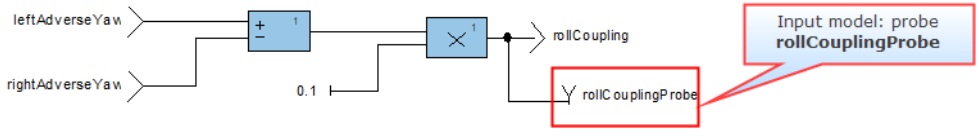
OPTION -NO_TIMESTAMP

Option Format	`-no_timestamp`
Code Generation	C and Ada (KCG 6.6.4 only)
Description	This option allows to generate KCG files without adding any file generation date in headers and footers.

OPTION -PROBES

Option Format	`-probes`
Code Generation	C and Ada
Description	If this option is set, all C/Ada variables corresponding to probes in the input model are kept in the contexts corresponding to node calls. Consequently, a call to a node computing only probes is kept in the generated code.

Example: Keep the `rollCouplingProbe` probe in the generated code for the `AdverseYaw` operator of the `RollRate` package



Below a sample of generated C code without `-probes` option

```
/* $*************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T15:59:11
*****************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "AdverseYaw_RollRate.h"

/** AdverseYaw represents simplified coupling effects between roll and yaw. */
/* RollRate::AdverseYaw */
kcg_real AdverseYaw_RollRate(
  /* RollRate::AdverseYaw::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::AdverseYaw::rightAdverseYaw */kcg_real rightAdverseYaw)
{
  /* RollRate::AdverseYaw::rollCoupling */ kcg_real rollCoupling;

  rollCoupling = (leftAdverseYaw - rightAdverseYaw) * 0.1;
  return rollCoupling;
}

/* $*************** KCG Version 6.4 (build i21) ***************
** AdverseYaw_RollRate.c
** Generation date: 2015-08-11T15:59:11
*****************************************************************$ */
```

Standard C Code for AdverseYaw operator

Below a sample of generated H file with `-probes` option set (`AdverseYaw_RollRate.h` with definitions of local probes)

```
/* $**************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T16:12:25
*****************************************************************$ */
#ifndef _AdverseYaw_RollRate_H_
#define _AdverseYaw_RollRate_H_

#include "kcg_types.h"

/* =====================  no input structure  ====================== */

/* =====================  no output structure  ====================== */

/* ========================  context type  ========================== */
typedef struct {
  /* --------------------------  outputs  --------------------------- */
  kcg_real /* RollRate::AdverseYaw::rollCoupling */ rollCoupling;
  /* ----------------------  local probes  ------------------------- */
  kcg_real /* RollRate::AdverseYaw::rollCouplingProbe */ rollCouplingProbe;
  /* -----------------  no initialization variables  ---------------- */
  /* ----------------------  no local memory  ---------------------- */
  /* --------------------- no sub nodes' contexts  -------------------- */
  /* ----------------- no clocks of observable data ------------------ */
} outC_AdverseYaw_RollRate;

/* ===========  node initialization and cycle functions  =========== */
/** AdverseYaw represents simplified coupling effects between roll and yaw. */
/* RollRate::AdverseYaw */
extern void AdverseYaw_RollRate(
  /* RollRate::AdverseYaw::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::AdverseYaw::rightAdverseYaw */kcg_real rightAdverseYaw,
  outC_AdverseYaw_RollRate *outC);

#ifndef KCG_NO_EXTERN_CALL_TO_RESET
extern void AdverseYaw_reset_RollRate(outC_AdverseYaw_RollRate *outC);
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

#ifndef KCG_USER_DEFINED_INIT
extern void AdverseYaw_init_RollRate(outC_AdverseYaw_RollRate *outC);
#endif /* KCG_USER_DEFINED_INIT */

#endif /* _AdverseYaw_RollRate_H_ */
/* $**************** KCG Version 6.4 (build i21) ***************
** AdverseYaw_RollRate.h
** Generation date: 2015-08-11T16:12:25
*****************************************************************$ */
```

C variable corresponding to probe in input model

Below a sample of generated C code with `-probes` option

```c
/* $*************** KCG Version 6.4 (build i21) ***************
** Command: kcg64.exe -config C:/Products/examples/GettingStarted/final/RollControl/KCG/config.txt
** Generation date: 2015-08-11T15:59:11
*****************************************************************$ */

#include "kcg_consts.h"
#include "kcg_sensors.h"
#include "AdverseYaw_RollRate.h"

#ifndef KCG_USER_DEFINED_INIT
void AdverseYaw_init_RollRate(outC_AdverseYaw_RollRate *outC)
{
  outC->rollCouplingProbe = 0.0;
  outC->rollCoupling = 0.0;
}
#endif /* KCG_USER_DEFINED_INIT */


#ifndef KCG_NO_EXTERN_CALL_TO_RESET
void AdverseYaw_reset_RollRate(outC_AdverseYaw_RollRate *outC)
{
}
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */

/** AdverseYaw represents simplified coupling effects between roll and yaw. */
/* RollRate::AdverseYaw */
void AdverseYaw_RollRate(
  /* RollRate::AdverseYaw::leftAdverseYaw */kcg_real leftAdverseYaw,
  /* RollRate::AdverseYaw::rightAdverseYaw */kcg_real rightAdverseYaw,
  outC_AdverseYaw_RollRate *outC)
{
  outC->rollCouplingProbe = (leftAdverseYaw - rightAdverseYaw) * 0.1;
  outC->rollCoupling = outC->rollCouplingProbe;
}

/* $*************** KCG Version 6.4 (build i21) ***************
** AdverseYaw_RollRate.c
** Generation date: 2015-08-11T16:07:10
*****************************************************************$ */
```

C Code for AdverseYaw operator with –probes option set

Below a sample of generated Ada code without `-probes` option

```
-- $******** SCADE Suite KCG 32-bit 6.6 (build i8) ********
-- ** Command: kcg66.exe -config C:/Dev Projects/R17 projects/GettingStarted/final/RollControl/KCG/config.txt
-- ** Generation date: 2015-08-27T09:45:27
-- ************************************************************$
package body RollRate
-- RollRate
is
  --# accept F, 501, all, "SCADE KCG Generated" ;
  --# accept F, 23,  all, "SCADE KCG Generated" ;
  --# accept F, 504, all, "SCADE KCG Generated" ;
  --# accept F, 602, all, "SCADE KCG Generated" ;

  --- RollRateWarning computes the left warning and right warning alarms
  --- according to the plane roll rate.
   -- RollRate::RollRateWarning/
  procedure RollRateWarning(
    -- RollRate::RollRateWarning/_L19/, RollRate::RollRateWarning/_L3/, RollRate::RollRateWarning/rollRate/
    rollRate_1 : in Kcg_Config.Kcg_Float64;
    -- RollRate::RollRateWarning/_L17/, RollRate::RollRateWarning/leftWarning/
    leftWarning : out Boolean;
    -- RollRate::RollRateWarning/_L18/, RollRate::RollRateWarning/rightWarning/
    rightWarning : out Boolean)
  is separate;

  --- AdverseYaw represents simplified coupling effects between roll and yaw.
    -- RollRate::AdverseYaw/
   function AdverseYaw(
    -- RollRate::AdverseYaw/_L4/, RollRate::AdverseYaw/leftAdverseYaw/
    leftAdverseYaw : in Kcg_Config.Kcg_Float64;
    -- RollRate::AdverseYaw/_L2/, RollRate::AdverseYaw/rightAdverseYaw/
    rightAdverseYaw : in Kcg_Config.Kcg_Float64) return Kcg_Config.Kcg_Float64
  is separate;

 --- RollRateCalculate computes the aircraft roll rate due to stick input and
  --- adverse yaw effects.
   -- RollRate::RollRateCalculate/
   function RollRateCalculate(
    -- RollRate::RollRateCalculate/_L1/, RollRate::RollRateCalculate/joystickCmd/
    joystickCmd : in Kcg_Config.Kcg_Float64;
    -- RollRate::RollRateCalculate/_L2/, RollRate::RollRateCalculate/leftAdverseYaw/
    leftAdverseYaw : in Kcg_Config.Kcg_Float64;
    -- RollRate::RollRateCalculate/_L3/, RollRate::RollRateCalculate/rightAdverseYaw/
    rightAdverseYaw : in Kcg_Config.Kcg_Float64) return Kcg_Config.Kcg_Float64
  is separate;

end RollRate;

-- $******** SCADE Suite KCG 32-bit 6.6(build i8) ********
-- ** rollrate.adb
-- ** Generation date: 2015-08-27T09:45:27
-- ************************************************************$
```

Standard Ada Code for AdverseYaw operator

Below a sample of generated Ada code with `-probes` option

```
-- $******** SCADE Suite KCG 32-bit 6.6 (build i8) ********
-- ** Command: kcg66.exe -config C:/Dev Projects/R17 projects/GettingStarted/final/RollControl/KCG/config.txt
-- ** Generation date: 2015-08-27T11:13:03
-- ************************************************************$
package body RollRate
-- RollRate
is
  --# accept F, 501, all, "SCADE KCG Generated" ;
  --# accept F, 23,  all, "SCADE KCG Generated" ;
  --# accept F, 504, all, "SCADE KCG Generated" ;
  --# accept F, 602, all, "SCADE KCG Generated" ;

  procedure AdverseYaw_Init(
    Ctx : out Context_AdverseYaw)
  is separate;

  procedure RollRateCalculate_Init(
    Ctx : out Context_RollRateCalculate)
  is separate;

  --- RollRateWarning computes the left warning and right warning alarms
  --- according to the plane roll rate.
   -- RollRate::RollRateWarning/
  procedure RollRateWarning(
   -- RollRate::RollRateWarning/_L19/, RollRate::RollRateWarning/_L3/, RollRate::RollRateWarning/rollRate/
   rollRate_1 : in Kcg_Config.Kcg_Float64;
   -- RollRate::RollRateWarning/_L17/, RollRate::RollRateWarning/leftWarning/
   leftWarning : out Boolean;
   -- RollRate::RollRateWarning/_L18/, RollRate::RollRateWarning/rightWarning/
   rightWarning : out Boolean)
  is separate;

  --- AdverseYaw represents simplified coupling effects between roll and yaw.
   -- RollRate::AdverseYaw/
  procedure AdverseYaw(
   -- RollRate::AdverseYaw/_L4/, RollRate::AdverseYaw/leftAdverseYaw/
   leftAdverseYaw : in Kcg_Config.Kcg_Float64;
   -- RollRate::AdverseYaw/_L2/, RollRate::AdverseYaw/rightAdverseYaw/
   rightAdverseYaw : in Kcg_Config.Kcg_Float64;
   Ctx : out Context_AdverseYaw)
  is separate;

 --- RollRateCalculate computes the aircraft roll rate due to stick input and
  --- adverse yaw effects.
   -- RollRate::RollRateCalculate/
   procedure RollRateCalculate(
    -- RollRate::RollRateCalculate/_L1/, RollRate::RollRateCalculate/joystickCmd/
    joystickCmd : in Kcg_Config.Kcg_Float64;
    -- RollRate::RollRateCalculate/_L2/, RollRate::RollRateCalculate/leftAdverseYaw/
    leftAdverseYaw : in Kcg_Config.Kcg_Float64;
    -- RollRate::RollRateCalculate/_L3/, RollRate::RollRateCalculate/rightAdverseYaw/
    rightAdverseYaw : in Kcg_Config.Kcg_Float64;
    Ctx : out Context_RollRateCalculate)
  is separate;

end RollRate;

-- $******** SCADE Suite KCG 32-bit 6.6 (build i8) ********
-- ** rollrate.adb
-- ** Generation date: 2015-08-27T11:13:03
-- ************************************************************$
```

Ada Code for AdverseYaw operator with `-probes` option set

OPTION -ROOT_PACKAGE

Option Format	`-root_package <package_path>`
Code Generation	Ada
Argument Format	Valid name for an Ada package.
Description	This option generates all Ada code within a specified package. For details about Ada packages, see "Package Hierarchy and SPARK Compliance" on page 234.
Errors	An error is raised if no valid name for an Ada package is given to the option

OPTION -SHORT_CIRCUIT

Option Format	`-short_circuit`
Code Generation	C and Ada
Description	This option specifies that no bitwise operator appears in the generated code. When this option is set, the translation of Scade Boolean operators into logical operators with default definitions for the Boolean macros (see `-wrap_c_ops`) use the C translations in Table 11.6 or the Ada translation in Table 11.7.

Table 11.6: Boolean Operators computation for C code depending on option

SCADE Operator	Expression		
	-short_circuit only	-wrap_c_ops only	none of the options
A or B	A || B	kcg_or(A, B)	A | B
A xor B	A !=B	kcg_xor(A, B)	A ^ B
A and B	A && B	kcg_and(A, B)	A & B
not A	! A	kcg_not(A)	! A

Table 11.7: Boolean Operators computation for Ada code depending on option

SCADE Operator	Expression	
	with -short_circuit	without -short_circuit
A or B	A or else B	A or B
A and B	A and then B	A and B

OPTION -SIGNIFICANCE_LENGTH

Option Format	`-significance_length significancelength`
Code Generation	C
Argument Format	Integer
Description	Identifiers in the generated code do not rely on the significance of more than <n> initial characters. If this option is set, two generated identifiers can be differentiated on their first <significancelength> characters. By default, <significancelength> is set to 31.
Errors	An error is raised if: • no argument is given to the option • no integer value given to the option • negative integer value given to the option • the effective value of the option is greater than the effective value of `-name_length` • the integer value of option is lower than 16. In this case, KCG behaves as if "-significance_length 16" was passed to the command line.

OPTION -STATE_VECTOR

Option Format	`-state_vector`
Code Generation	C
Description	This option generates the state vector structure and function prototypes in the header files of the root operator(s) and of operators in the cone of influence having at least one state variable. The state vector structure called SV_<root_name> is composed of: • memorized outputs • memorized probes of the operator (if `-probes` option set) • initialization variables (at initial tick) • operator memories • state vector of each non-expanded sub-operator needing C state (one state vector per instance) If this option is set, if `-global_root_context` is not set, and if a root operator needs a C state, two functions are generated to deal with the state vector: • The first one, **`kcg_save`**, copies the state variables of the context structure into a state vector. Its prototype is : `extern void kcg_save_SV_<root_name>(SV_<root_name> *SV, outC_<root_name> *outC);` • The second one, **`kcg_load`**, restores the state variables of a state vector into a context. Its prototype is: `extern void kcg_load_SV_<root_name>(outC_<root_name> *outC, SV_<root_name> *SV);` Notice that the corresponding cycle functions have to be called at least once after loading to ensure that all observable variables hold valid values. ... continued next page

Description (continued)	If this option is set and the root operator needs a C state, the header file associated with a root operator contains the declaration of the state vector structure made of: • initial tick variables • root operator memories • state vectors of non expanded sub-operators which needs a C state (one per instance, one per iterator step) When state vector structure is added in the header file, the root operator source file and each source file of non-expanded operators contain the definition of the restore and save functions.

OPTION -USE_TYPES

Option Format	**`-use_types`** `typeslist`
Code Generation	Ada
Argument Format	Comma-separated list of built-in Scade types among char, size, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, and float64.
Description	This option generates each specified type with a “use type” clause in the generated Ada code. By default, no use type is generated for predefined types.
Errors	An error is raised if one of the listed types is not a valid keyword among char, size, int8, int16, int32, int64, uint8, uint16, uint32, uint64, float32, or float64.

OPTION -USER_CONFIG

Option Format	`-user_config filename`
Code Generation	C
Argument Format	string ending with ".h"
Description	If this option is set, the header file <filename> is included (#include directive) at the beginning of the generated file `kcg_types.h`. The header file `<filename>` can contain a path.
Errors	An error is raised if: • no argument is given to the option • <filename> is not an .h file

OPTION -WRAP_C_OPS

Option Format	`-wrap_c_ops`
Code Generation	C

Description	This option encapsulates arithmetics and relational operators into C protected macros. Macro definitions are included into the generated `kcg_types.h` file.
	Macros definitions for *non-Boolean unary operators* match this pattern:
	`#ifndef kcg_<Name>`
	`#define kcg_<Name>(kcg_C1) body`
	`#endif /* kcg_<Name> */`
	Macro definitions for all *non-Boolean binary operators without type cast* match this pattern except for `kcg_index`:
	`#ifndef kcg_<Name>`
	`#define kcg_<Name>(kcg_C1, kcg_C2) ((kcg_C1) op (kcg_C2))`
	`#endif /* kcg_<Name> */`
	The macro definition of `kcg_index` for array indexation operator is:
	`#ifndef kcg_index`
	`#define kcg_index(kcg_C1, kcg_C2) ((kcg_C1) [(kcg_C2)])`
	`#endif /* kcg_index */`
	Macro definitions for *binary operators with one or more type cast* match this pattern:
	`#ifndef kcg_<Name>`
	`#define kcg_<Name>(kcg_C1, kcg_C2) body`
	`#endif /* kcg_<Name> */`
	All available C macro definitions are listed in Table 11.8 below.
	If `-short_circuit` option is set, the default macro definitions for Boolean operators use the translations described in Table 11.6.
	As for arithmetics and relational operators, this option wraps any C conditional expression into a call to the user-defined `kcg_cond` macro.
	With this option, any increment of a loop variable is wrapped into a call to the predefined `kcg_incr` (or `kcg_decr`) macro.
	With this option, any projection of an array in the generated C code is wrapped into a call to the predefined `kcg_index` macro.

Table 11.8: List of C protected macros by categories of operator

Categories	SCADE expression	C macro names	C expression
Arithmetic operators:			
Unary operator	`- A`	`kcg_uminus_t(A)`[1]	`- A`
Binary operator	`A + B`	`kcg_plus_t(A,B)`	`A + B`
	`A - B`	`kcg_minus_t(A,B)`	`A - B`
	`A * B`	`kcg_mult_t(A,B)`	`A * B`
	`A / B`	`kcg_div_t(A,B)`	`A / B`
	`A div B`	`kcg_div_t(A,B)`	`A / B`
	`A mod B`	`kcg_mod_t(A,B)`	`A % B`
Predefined or imported scalar type relations:			
Binary inequality	`A != B`	`kcg_diff_t(A,B)`	`A != B`
Binary equality	`A == B`	`kcg_eq_t(A,B)`	`A == B`
Enumerated type relations:			
Binary inequality	`A != B`	`kcg_diff(A,B)`	`A != B`
Binary equality	`A == B`	`kcg_eq(A,B)`	`A == B`
Structured or imported type relations:			
Binary inequality	`A != B`	`kcg_not(cmp(&A, &B))`	`! cmp(&A, &B)`
Binary equality	`A == B`	`cmp(&A, &B)`	`cmp(&A, &B)`
Boolean conditions:			
Unary operator		`kcg_cond(C)`	`C`
Iterator loop variables:			
Unary operator		`kcg_incr(A)`	`A++`
Binary operator		`kcg_index(A,B)`	`A[B]`

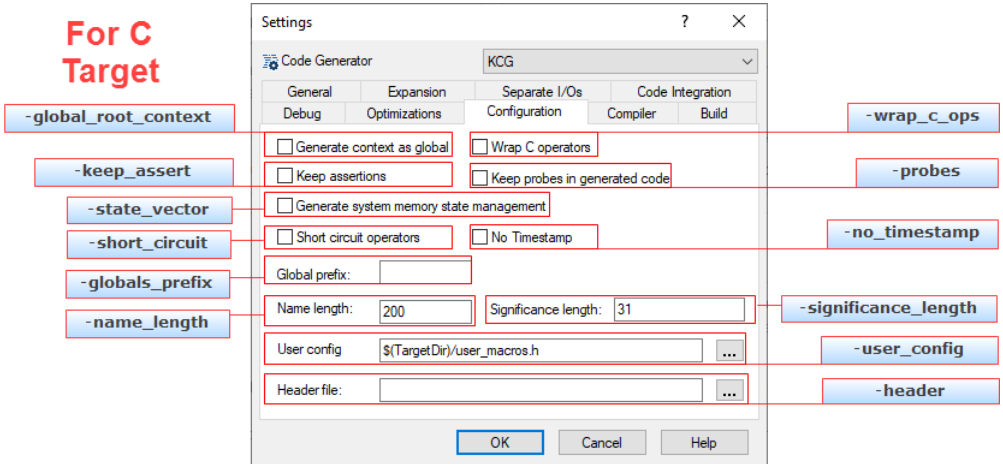
Table 11.8: List of C protected macros by categories of operator (Continued)

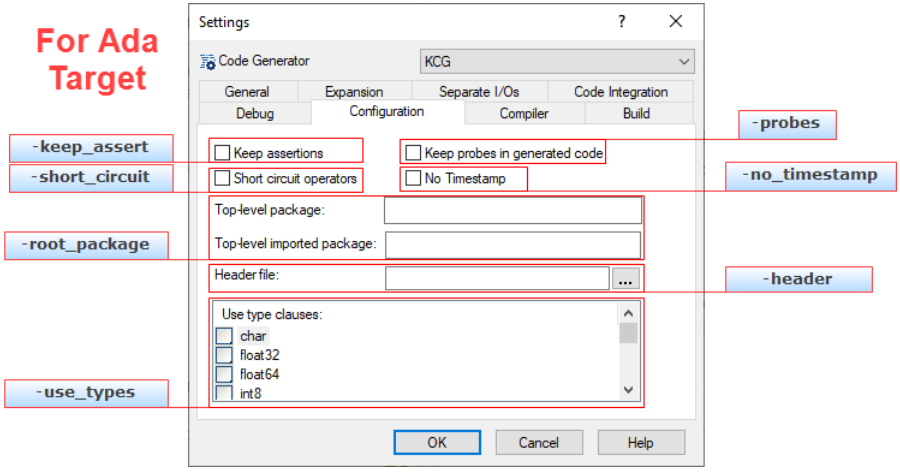
Categories	SCADE expression	C macro names	C expression
Comparison operators:			
Binary operator	`A < B`	`kcg_lt_t(A,B)`	`A < B`
	`A > B`	`kcg_gt_t(A,B)`	`A > B`
	`A >= B`	`kcg_ge_t(A,B)`	`A >= B`
	`A <= B`	`kcg_le_t(A,B)`	`A <= B`
Logical operators:			
Binary operator	`A land B`	`kcg_land_t(A,B)`	`A & B`
	`A lor B`	`kcg_lor_t(A,B)`	`A | B`
	`A lxor B`	`kcg_lxor_t(A,B)`	`A ^ B`
	`A lsl B`	`kcg_lsl_t(A,B)`	`A << B`
	`Alsr B`	`kcg_lsr_t(A,B)`	`A >> B`
Unary operator	`lnot A`	`kcg_lnot_t(A)`	`kcg_lnot_t(A)`
Literals:			
Literal of type size	`l`	`kcg_lit_size(l)`	`l`
Literal of other numeric type (predefined or imported)	`l`	`kcg_lit_t(l)`	`kcg_lit_t(l)`
Negative literal of type size	`-l`	`kcg_uminus_size(kcg_lit_size(l))`	`-l`
Negative literal of other numeric type (predefined or imported)	`-l`	`kcg_uminus_size(kcg_lit_t(l))`	`kcg_lit_t(-l)`
Numeric cast:			
	`e:t`	`kcg_t1_to_t(e)` `where t1 is type of e`	

1. In macro names, t in bold stands for numeric type name.

KCG CONFIGURATION OPTIONS IN UI SETTINGS

The configuration options available to run KCG/MCG in command line have an equivalent in the **Configuration** tab on the Code Generator Settings window. The content depends on the selected Code Generator target.





Description of Additional Options

The following KCG/MCG command line option is additional.

OPTION -XML_BASENAME

Option Format	**`-xml_basename`** `<name>`
Code Generation	C and Ada
Argument Format	string ending with ".scade"
Description	This option renames the Scade textual file generated during XML translation of input files into `<name>.scade`. By default, the file is named `kcg_xml_filter_out.scade` and it is always generated when the model contains `.xscade` files (see "Schema of Input/Output Files" on page 183). If there is no such file as input, only `kcg.log` is generated.
Errors	An error is raised if no argument is given to the option

12 /Code Generation Messages

This appendix provides the list of messages returned by SCADE Suite KCG and Multicore Code Generator (MCG). The execution of SCADE Suite Code Generators may raise the following messages:

- Warnings (WAR): notification of potential issues having no impact on the correctness of the generated code
- Errors (ERR): incorrect input from model or command line causing tool interruption
- Failures (FAIL): unexpected internal or external event causing tool interruption

- "Messages from Model Checks"
- "Messages from XML Checks"
- "Failure Messages"
- "Command Line Messages"

Messages from Model Checks

Model checks performed by SCADE Suite KCG Code Generator and Multicore Code Generator can report the following:

- "Syntactic/Post-Synctactic Check Errors/Warnings"
- "Namespace Check Errors"
- "Type Check Errors/Warnings"
- "Clock Check Errors"
- "Causality Check Errors/Warnings"
- "Initialization Check Errors"
- "Monomorphisation Check Errors/Warnings"
- "Messages from Additional Checks"

Syntactic/Post-Synctactic Check Errors/Warnings

Table 12.1: Syntactic check errors

ID	Message	Meaning
Syntax Error (ERR_1)	*Parse error*	An input file in textual format does not comply with BNF syntax or a pragma file does not comply with syntax of external pragmas
Syntax Error (ERR_17)	*An array cannot have size zero*	

Table 12.2: Post-syntactic check errors

ID	Message	Meaning
Post-syntactic Error (ERR_10)	*An imported type declaration cannot have a type definition part*	
Post-syntactic Error (ERR_11)	*A non imported type declaration must have a type definition part*	
Post-syntactic Error (ERR_12)	*A non imported constant declaration must have an expression definition part*	
Post-syntactic Error (ERR_13)	*An imported constant declaration cannot have a definition part*	
Post-syntactic Error (ERR_15)	*A non imported user operator declaration must have a body*	
Post-syntactic Error (ERR_16)	*An imported user operator declaration cannot have a body*	
Post-syntactic Error (ERR_17)	*A non imported type declaration cannot have a numeric kind*	

Table 12.3: Post-syntactic check warnings

ID	Message	Meaning
Post-syntactic Warning (WAR_1)	*Parameter with default or last in imported operator*	An imported user operator declaration cannot have parameters with default or last declarations.
Post-syntactic Warning (WAR_2)	*A default declaration in an input parameter has no use*	An input of a user defined operator cannot have a default declaration

Namespace Check Errors

Table 12.4: Semantic namespace errors

ID	Message	Meaning
Namespace Error (ERR_1)	*Multiple package declaration*	A package identifier is used several times in the same scope
Namespace Error (ERR_2)	*Multiple declaration*	A declaration identifier (type, constant, sensor, user operator, local flow, signal, enum value) is used several times in the same scope
Namespace Error (ERR_3)	*Multiple declaration via open*	A package cannot be opened in a scope because it redefines an identifier belonging to this scope or to another opened package
Namespace Error (ERR_4)	*Multiple field declaration*	A field identifier is used several times in the same record declaration

Table 12.4: Semantic namespace errors (Continued)

ID	Message	Meaning
Namespace Error (ERR_5)	*Multiple block declaration*	A block identifier (control block or automaton state) is used several times in the same scope
Namespace Error (ERR_6)	*Multiple open*	A package is opened several times in the same scope
Namespace Error (ERR_7)	*Unknown package*	A package identifier is undefined in the current scope
Namespace Error (ERR_8)	*Unknown identifier*	An identifier is undefined in the current scope
Namespace Error (ERR_9)	*Bad identifier kind*	An valid identifier is used in a wrong context (for instance: a constant for a type)

Type Check Errors/Warnings

Table 12.5: Semantic type errors

ID	Message	Meaning
Type Error (ERR_100)	*Type mismatch*	An expression has a type different from the expected one
Type Error (ERR_101)	*Polymorphic type too general*	A polymorphic type variable should be replaced by a more restricted type
Type Error (ERR_102)	*Size constraint violated*	Several possible origins: a size must be a strictly positive integer, some operators require two arrays of the same size, an index is out of bounds
Type Error (ERR_103)	*Numeric constraint violated*	An integer literal or size parameter is not in the range of its inferred type
Type Error (ERR_104)	*Invalid instantiation of size parameters*	Missing or extra size parameter
Type Error (ERR_105)	*Not a size expression*	Valid size expressions are described in "Type Expressions" on page 19 of the *Scade Language Reference Manual*.
Type Error (ERR_107)	*Not a structure*	An expression of a structure type is required
Type Error (ERR_108)	*Not an enumeration*	An expression of an enumerated type is required
Type Error (ERR_109)	*Not an array*	An expression of an array type is required
Type Error (ERR_110)	*Neither an array nor a structure*	An expression of a structure or array type is required
Type Error (ERR_111)	*Unexpected internal state*	An operator declared as function should probably be declared as node

Table 12.5: Semantic type errors (Continued)

ID	Message	Meaning
Type Error (ERR_112)	*Pattern-matching not exhaustive*	A reachable pattern is not caught by the current enumeration
Type Error (ERR_113)	*Redundancy in pattern-matching*	Two clauses can be satisfied by the same expression
Type Error (ERR_114)	*Malformed pattern-matching*	An unexpected clause occurs according to the condition expression
Type Error (ERR_115)	*Unknown structure label*	Use of a label that has not been declared in the structure's type
Type Error (ERR_116)	*Invalid dimension in transpose*	Second and third argument of transpose should be between 1 and the maximum number of nested arrays
Type Error (ERR_117)	*Invalid path projection*	A label projection is applied to anything but a structure; or an index projection to anything but an array
Type Error (ERR_120)	*Invalid use of iterator*	Iterators require as argument operators with a specific type signature.
Type Error (ERR_121)	*Unexpected empty list*	Some operators require an non empty list as argument (#, array constructor)
Type Error (ERR_122)	*Unexpected group of flows*	Group expressions can only be used by specificoperators see "Group Expressions" on page 24 in the *Scade Language Reference Manual*.
Type Error (ERR_125)	*Not a static expression*	Valid size expressions are described in "Constants" on page 27 of the *Scade Language Reference Manual*.
Type Error (ERR_126)	*Polymorphic specializing operator*	A specializing operator must not contain type variables

Table 12.5: Semantic type errors (Continued)

ID	Message	Meaning
Type Error (ERR_127)	*Invalid specializing operator*	An operator can only be specialized by a compatible operator (including temporality and subtyping)
Type Error (ERR_128)	*Identical specializing operator*	Several specializations of the same operators have identical type signatures
Type Error (ERR_130)	*Cannot specialize by a parametrized operator*	A specializing operator cannot have size parametrized operator parameters.
Type Error (ERR_131)	*Cannot infer literal type*	The type of a literal cannot be inferred from its context

Table 12.6: Semantic type warnings

ID	Message	Meaning
Type Warning (WAR_151)	*Internal state expected*	An operator is declared with the keyword **node** but it has no internal state
Type Warning (WAR_152)	*Useless application of transpose*	Both dimensional arguments of the **`transpose`** operator are equal

Clock Check Errors

Table 12.7: Semantic clock errors

ID	Message	Meaning
Clock Error (ERR_200)	*Incompatible clocks*	Several possible origins: two expressions are required to have the same clock, a group of flows is not on the same clock, an operator cannot be applied on a list of parameters
Clock Error (ERR_201)	*Not declared as a clock*	An identifier used as clock has not been declared as a clock identifier
Clock Error (ERR_202)	*Invalid flow clock in control block*	An identifier cannot be used inside a control block because its clock is incompatible with the block's clock
Clock Error (ERR_204)	*First argument of "merge" is not a clock*	Only a clock identifier can be used as its first argument
Clock Error (ERR_205)	*An argument of "merge" is not on a sub-clock*	All the remaining arguments of **`merge`** operator must be on a subclock of the first one
Clock Error (ERR_206)	*"merge" is not exhaustive*	The clocks of all the remaining arguments of **`merge`** operator must cover all the enumerators of the first argument clock's type
Clock Error (ERR_207)	*"merge" is redundant*	Several of the remaining arguments are based on the same subclock as the first argument
Clock Error (ERR_208)	*Undefined carrier name in clock*	An input declared as a clock must be instantiated by another clock identifier

Table 12.7: Semantic clock errors (Continued)

ID	Message	Meaning
Clock Error (ERR_209)	*Clock declarations are mutually recursive*	For instance: `clock h1: bool when h2` and `clock h2: bool when h1`
Clock Error (ERR_210)	*"merge" has branches with different clock arities*	There are two branches with clock types of different length.

Causality Check Errors/Warnings

Table 12.8: Semantic causality errors

ID	Message	Meaning
Causality Error (ERR_400)	*Causality error*	A flow is defined on itself

Initialization Check Errors

Table 12.9: Semantic initialization errors

ID	Message	Meaning
Initialization Error (ERR_300)	*Initialization error*	An expression is not well defined on its first instant of activation

Monomorphisation Check Errors/Warnings

Table 12.10: Monomorphisation errors

ID	Message	Meaning
Monomorphisation Error (ERR_550)	*root operator has a polymorphic type*	One of the arguments of the -root option is a path to a polymorphic operator. Only applicable to SCADE Suite KCG Ada.
Monomorphisation Error (ERR_551)	*root operator has size parameters*	One of the arguments of the -root option is a path to an operator having size parameters. Only applicable to SCADE Suite KCG Ada.
Monomorphisation Error (ERR_552)	*illegal root operator*	A root operator (as specified by the -root option) appears in the call graph of another root operator. Only applicable to SCADE Suite KCG Ada.

Table 12.11: Monomorphisation warnings

ID	Message	Meaning
Monomorphisation Warning (WAR_153)	*No specialized operator found*	A polymorphic imported operator has no specializing operator

Messages from Additional Checks

You can encounter messages from the following checks:

- "Post-Namespace Checks"
- "Code Generation Checks"

POST-NAMESPACE CHECKS

Table 12.12: Post-namespace warnings

ID	Message	Meaning
Post-namespace Warning (WAR_502)	*Unreachable state*	No path exists between the initial state and another state in an automaton
Post-namespace Warning (WAR_503)	*Impossible synchro*	When at least one sub-automaton has no final state, the synchro transition is never fired.
Post-namespace Warning (WAR_505)	*Synchro always fired*	When no sub-automaton exists, or when all the states in all the sub-automata are final, the synchro transition is always fired.
Post-namespace Warning (WAR_506)	*Useless final declaration*	A state is tagged as final while there is no synchro transition in its father state
Post-namespace Warning (WAR_507)	*Uncaught signal*	A signal was emitted but never received
Post-namespace Warning (WAR_508)	*Unused flow*	An input or local flow is never used in its operator
Post-namespace Warning (WAR_510)	*Unemitted signal*	A signal is never emitted in its operator
Post-namespace Warning (WAR_511)	*Useless pragma*	A pragma scalar is used on an imported numeric type

Table 12.12: Post-namespace warnings (Continued)

ID	Message	Meaning
Post-namespace Warning (WAR_513)	*Ignored external pragma*	A pragma in a pragma file was ignored because the model path is not supported or does not match the input model. Only applicable to MCG.

Table 12.13: Post-namespace errors

ID	Message	Meaning
Post-namespace Error (ERR_511)	*Reused pragma*	Occurrence pragma , default pragma, initializer pragma, enum_val pragma, or name pragma used several times on the same construction
Post-namespace Error (ERR_512)	*Multiple flow definition*	A flow identifier has several definitions, either direct or within control blocks. A flow cannot be defined in a state and in a transition. A flow identifier cannot appear twice in a return block.
Post-namespace Error (ERR_513)	*Incorrect control block "returns"*	Several possible origins: a flow is declared as returned but has not been defined, a flow was defined but was not declared as returned
Post-namespace Error (ERR_514)	*Input definition*	An input flow identifier cannot appear on the left hand side of a definition
Post-namespace Error (ERR_515)	*Illegal default or last declaration*	A local group cannot have a default or a last declaration
Post-namespace Error (ERR_516)	*Undefined variable*	A local or output variable is not defined in its operator
Post-namespace Error (ERR_517)	*Illegal clock declaration*	No clock declaration is allowed in a group of flows

Table 12.13: Post-namespace errors (Continued)

ID	Message	Meaning
Post-namespace Error (ERR_518)	*Cyclic declarations*	A declaration depends on another declaration which also depends on the first one
Post-namespace Error (ERR_519)	*Missing initial state*	An automaton must have an initial state
Post-namespace Error (ERR_520)	*More than one initial state*	An automaton must have only one initial state
Post-namespace Error (ERR_521)	*Unused size*	A size parameter is never used in its operator
Post-namespace Error (ERR_522)	*Unexpected operator with clocked I/O*	An operator with clocked input or output cannot be a root operator and must be expanded when it is not a root operator.
Post-namespace Error (ERR_523)	*Incompatible pragmas*	Pragma separate_io (resp. expand) is incompatible with its opposite no_separate_io (resp. no_expand)
Post-namespace Error (ERR_524)	*Cannot extract task*	An operator with an attached pragma `C:task` cannot be extracted because: `-static` option is used in the command line; the caller is multi-instantiated after expansion; the caller is iterated but not expanded; the caller is under a fold- like or partial iterator; or the caller is under an unrolled iterator. Only applicable to MCG.

Table 12.14: Code generation errors

ID	Message	Meaning
Code Generation Error (ERR_600)	*Cannot use pragma name*	Name pragma conflicts with other interface identifiers, is too long with respect to `-name_length`, conflicts with option `-significance_length`, or it is not a valid C/Ada identifier.
Code Generation Error (ERR_606)	*Cannot assign a name*	The global prefix is commonly too long with respect to the significance length which must be increased.
Code Generation Error (ERR_607)	*Multiple manifest pragmas*	More than one type in the type equivalence class has a manifest pragma.
Code Generation Error (ERR_608)	*Invalid initializer pragma*	Initializer pragma conflicts with other interface identifiers, is too long with respect to `-name_length`, conflicts with option `-significance_length`, or it is not a valid C/Ada identifier.
Code Generation Error (ERR_609)	*Invalid task pragma*	A C:task pragma is not a valid C identifier. Only applicable to MCG.

Table 12.15: Code generation warnings

ID	Message	Meaning
Code Generation Warning (WAR_604)	*Renaming interface identifier*	The name of an interface identifier (sensor, constant, type, root operator, any imported declaration) is not generated with a predictable name and gets renamed.
Code Generation Warning (WAR_620)	*Introducing additional package*	Choose either to ignore the warning, or re-design the SCADE Suite model to directly comply with the SPARK package system. Only applicable to SCADE Suite KCG (Ada).

Messages from XML Checks

Table 12.16: XML errors

ID	Message	Meaning
XML Format Error (ERR_1)	*Parse error*	The input file in XML format is invalid according to XML standard
XML Format Error (ERR_2)	*Bad xscade format*	The input file in XML format does not comply with the xscade format according to the RelaxNG description of the Scade XML format
XML Format Error (ERR_3)	*Incomplete model*	An .xscade XML file contains the XML element `<NullRef/>`.

Failure Messages

Table 12.17: Internal and system failure messages

ID	Message	Meaning
Internal Failure (FAIL_1)	*Unexpected failure*	Internal failure occurred while running SCADE Suite KCG. Not applicable to MCG.
Internal Failure (FAIL_2)	*Unsupported feature*	The input model contains a specialization of an internal (*i.e.,* non-imported) user operator, combinations of activate and partial iterators, or a literal pattern that cannot be evaluated in the range of int64 or uint64
System Failure (FAIL_1)	*Input/Output error*	An input or output file is not accessible
System Failure (FAIL_3)	*Out of memory*	Not enough memory available to run the program (heap exhaustion).

Table 12.17: Internal and system failure messages (Continued)

ID	Message	Meaning
System Failure (FAIL_4)	*Stack overflow*	Not enough memory available to run the program (stack exhaustion).

Table 12.18: License failure messages

ID	Message	Meaning
License Failure (FAIL_22)	*Missing License*	No license token is present
License Failure (FAIL_23)	*Invalid authentication*	

Command Line Messages

The following messages can occur while using code generation in batch commands:

- "Command Line Errors"
- "Command Line Warnings"

COMMAND LINE ERRORS

Table 12.19: Command line error messages

ID	Message	Meaning
Command Line Error (ERR_1)	*No root operator specified*	The option -root is mandatory to build the model
Command Line Error (ERR_2)	*SCADE 6 files expected*	An input file has another extension than .scade, .xscade
Command Line Error (ERR_3)	*File not found*	One of input files has no valid path for operating system or is not an existing file

Table 12.19: Command line error messages (Continued)

ID	Message	Meaning
Command Line Error (ERR_5)	*Option already set*	An option occurs several times in the option list
Command Line Error (ERR_6)	*Missing argument*	An argument needed by an option is missing
Command Line Error (ERR_7)	*Missing integer*	An integer value needed by an option is missing
Command Line Error (ERR_8)	*Incompatible options*	One of the following pairs of options are present in the command line: -exp and -expall; -noexp and -expall; -exp and -noexp; -observe and -no_observe; -observe and -debug; -debug and -no_observe; -separate_io and -no_separate_io; -separate_io and -separate_io_all; -separate_io_all and -no_separate_io; -target Ada and C specific options; -target C and Ada specific options.
Command Line Error (ERR_9)	*Unknown optimization level*	Valid optimization levels are 0,1,2,3. Only applicable to SCADE Suite KCG.
Command Line Error (ERR_10)	*Invalid comment in the header*	The header file given to the -header option contains invalid C or Ada comment. Only applicable to SCADE Suite KCG.
Command Line Error (ERR_12)	*Unknown target*	The only valid argument for -target option is C or Ada depending on used KCG
Command Line Error (ERR_13)	*Unknown option*	An option occurring in the option list is not a recognized option
Command Line Error (ERR_14)	*Bad command line format*	No file is given to the -config option of kcg command line or the list of command line arguments is empty

Table 12.19: Command line error messages (Continued)

ID	Message	Meaning
Command Line Error (ERR_15)	*-exp/-noexp operator doesn't exist*	An operator's name given in the list does not belong to the input model
Command Line Error (ERR_16)	*Root operator doesn't exist*	The operator's name given to the -root option does not belong to the input model
Command Line Error (ERR_17)	*Root operator has a polymorphic type*	The root operator given to the -root option is a polymorphic operator.
Command Line Error (ERR_18)	*Root operator is imported*	The root operator given to the -root option is an imported operator
Command Line Error (ERR_19)	*Root operator has size parameters*	The root operator given to the -root option is an operator with size parameters.
Command Line Error (ERR_20)	*cannot expand an imported operator*	The path of an imported operator is in the list of operators given to the -exp option
Command Line Error (ERR_21)	*-significance_length too long with respect to name_length*	The value given to -significance_length option must be lower than that given to -name_length option
Command Line Error (ERR_23)	*-globals_prefix argument is not a valid C identifier*	The argument given to -globals_prefix option must be a valid C identifier
Command Line Error (ERR_25)	*-observe/ no_observe operator doesn't exist*	A path provided as argument of `-observe/ -no_observe` option is not the fully qualified path of an existing user-defined operator. A fully qualified path explicitly indicates which packages are traversed to reach an operator starting from the top-level namespace of the model and ignoring any open directive. Only applicable to SCADE Suite KCG.

Table 12.19: Command line error messages (Continued)

ID	Message	Meaning
Command Line Error (ERR_26)	*cannot observe an imported operator*	The path of an imported operator is in the list of operators provided by `-observe` option. Only applicable to SCADE Suite KCG.
Command Line Error (ERR_27)	*incompatible root operators*	A root operator is in the cone of influence of another root operator.
Command Line Error (ERR_28)	*invalid root package name*	The `-root_package` argument is not a valid Ada path. Only applicable to SCADE Suite KCG (Ada).
Command Line Error (ERR_29)	*bad predefined or internal type name*	An argument of the `-use_types` option is not a valid name for Ada. Only applicable to SCADE Suite KCG (Ada).
Command Line Error (ERR_30)	*-separate_io/-no_separate_io operator doesn't exist*	A path provided as argument of the `-separate_io/-no_separate_io` option is not a fully qualified path of an existing operator. Only applicable to SCADE Suite KCG.
Command Line Error (ERR_31)	*cannot use -separate_io/-no_separate_io on imported operator*	The fully qualified path of an imported operator was provided as argument of `-separate_io/-no_separate_io` option. Only applicable to SCADE Suite KCG.
Command Line Error (ERR_32)	*invalid imported root package name*	The `-imported_root_package` argument is not a valid Ada path. Only applicable to SCADE Suite KCG (Ada).

COMMAND LINE WARNINGS

Table 12.20: Command line warning messages

ID	Message	Meaning
Command Line Warning (WAR_1)	*root operator not expanded for code generation*	The root operator specified in the `-root` option is in the operator list set for `-exp` option or an `expand` pragma is attached to the specified operator. Only applicable to SCADE Suite KCG.
Command Line Warning (WAR_3)	*-name_length parameter must not be lower than 16. It has been set to 16.*	Valid integer values are greater than or equal to 16.
Command Line Warning (WAR_4)	*-significance_length must not be lower than 16. It has been set to 16.*	Valid integer values are greater than or equal to 16.

Index

Symbols

.ann files
  technical description 25
.aty files
  technical description 20

A

Abnormal configuration 175
Access denied 175
Additional observable variables 152
Annotable objects 6
Annotation type file
  annotability relationship definition module 22
  note type definition module 20
  structure 20
Annotations
  adding note type 15
  annotable environment objects 6
  annotations functions 14
  attributes of inlayable annotation types 29
  checking functions 17
  creating note on annotable object 14
  data identification 12
  definition 5
  deleting function 14
  deleting note type 15
  editing function 14
  inlaid 11
  list of annotable objects 6
  loading and saving functions 17
  modifying note type 16
  note handling functions 14
  note type compatibility 16
  note type handling functions 15
  objects
    Attribute definition 7
    Attribute group 7
    Attribute instance 9
    Category 8
    Note 9
    Note type 7
    Property 8
  objects and associations 7
  predefined properties
    kernel recognized 27
  printable annotation 31
  propagated in generated code 12
  storage formalism 18
    annotation file 25
    annotation type file 20
    annotation values definition module 26
    file structure 25
  storage principles 13
API
  public API gateway functions 138
API-related Tcl commands 73
Application states
  definition 163
Argument missing 175
Assertion violated 174
Attribute definition 7
Attribute group 7
Attribute instance
  annotation objects 9

B

Bad input list 176

C

C files
  monomorphic instance 190
Cardinality 17
Category
  annotation objects 8
Checking function
  cardinality limits 17
Code generation
  operator context 195
Cone of influence 189, 202, 209, 215, 217, 218, 249, 301
Conversion functions
  Ada 126
  C 123
  external types 107
  vtable structure 107
CSN1 format 18
Custom Registration 77
Custom Tcl scripts
  file registration 78
  registering 77
Customization scripts 74
  registration and evaluation 77
Cycle function 196

Index

D

deep 104
Default values
  basic types 90
  Boolean variables 106
  characters 106
  integers 106
  real variables 106

E

Error codes 173
  abnormal configuration 175
  access denied 175
  argument missing 175
  assertion violated 174
  incompatible options 175
  incorrect option 175
  inexistent file 175
  invalid number of inputs 174
  invalid option combination 174
  numeric error 174
  TCL_LIBRARY environment 174
  type error 174
  unknown option 174
Extended Backus-Naur Forms (EBNF) 18

F

File format
  OBS files 168
Fully qualified path 285
Functions
  cycle 195
  initialization 195
  reset 195

G

Generated random values
  intervals 97
Graphical UI extensions 75
GUI display-related commands 74

I

Imported operators
  vtable structure (for Ada code) 131
  vtable structure (for C code) 130
Imported types
  vtable structure (for Ada code) 111
  vtable structure (for C code) 110
Incompatible options 175
Incorrect option 175
Inexistent file 175
Initialization function 196
Inlaid annotations
  characteristics 11
  storing information display 11
Input file 103, 164
Input scenarios 150
Input vector scenarios
  definition 164
Instance name
  pragma 242
Interactive simulation
  scenario-driven 150
Invalid number of inputs 174
Invalid option combination 174

K

Kcg_Types
  Ada package 225

L

Loading/saving functions
  loading annotation types 17
  loading annotations 17
  saving annotations 17

M

Macros
  KCG_NO_EXTERN_CALL_TO_RESET 196
  KCG_USER_DEFINED_INIT 196
Monomorphic instance
  C files 190

N

Note
  annotation objects 9
  handling functions 14
Note type 7
  handling functions 15
Numeric error 174

O

Observable variables 93, 94, 95, 96, 105
Operator context
  functions 195

Index

Output file 166

P

Pragmas
- Ada-specific 255
- by KCG version 260
- const 257
- C-specific 257
- doc 246
- enum_val 247
- expand 248
- for instance names 242
- hide 255
- initializer 249
- keep 250
- manifest 252
- name 253
- noexpand 249
- scalar 257
- separate_io 254
- syntax 241
- task 258
- unroll 259

Property
- annotation objects 8

R

Random input vector 96, 97, 98
Random Tcl generator 97
Reset function 196
Root 105

S

SCADE user interface
- customization 74

Script Wizard 76
Scripting commands
- GUI display 74

Scripts
- SCADE user interface customization 74

SimConvType
- definition 112

Simulation
- Full batch mode 150
- in command line 149

Simulation batch options
- -obs 152
- -out 152
- -scenario 152

Simulation errors
- bad input list 176
- error codes 173
- type check 176

Simulation scenarios
- example 170

Simulation snapshots
- definition 163

Simulation variables 105
Simulator API function
- SsmCloseScadeSimulator 139
- SsmDbgStep 141
- SsmGetLastError 142
- SsmGetOutput 141
- SsmGetOutputVector 142
- SsmGuiActivate 139
- SsmGuiRefresh 139
- SsmOpenScadeSimulator 139
- SsmSetConnectRetry 140
- SsmSetHostName 140
- SsmSetInput 140
- SsmSetInputVector 141
- SsmSetScadeInstallPath 140
- SsmStep 141

Simulator scenarios 150
- definition (Tcl) 163

SSM::alias
- Simulator Tcl command 86, 87

SSM::back
- Simulator Tcl command 87

SSM::check
- Simulator Tcl command 88

SSM::cycle
- Simulator Tcl command 90

SSM::default_bool
- Simulator Tcl variable 106

SSM::default_char
- Simulator Tcl variable 106

SSM::default_inputs
- Simulator Tcl command 90

SSM::default_int
- Simulator Tcl variable 106

SSM::default_real
- Simulator Tcl variable 106

SSM::default_value
- Simulator Tcl command 91

SSM::default_value_matching
- Simulator Tcl command 91

SSM::get
- Simulator Tcl command 91

SSM::get_active_branches

Index

Simulator Tcl command 92
SSM::get_active_matches
Simulator Tcl command 92
SSM::get_active_states
Simulator Tcl command 92
SSM::get_cycle
Simulator Tcl command 92
SSM::get_fired_strong_transitions
Simulator Tcl command 93
SSM::get_fired_weak_transitions
Simulator Tcl command 93
SSM::get_obs_list
Simulator Tcl command 93
SSM::get_one_obs_list
Simulator Tcl command 94
SSM::get_outputs
Simulator Tcl command 94
SSM::go
Simulator Tcl command 95
SSM::print
Simulator Tcl command 95
SSM::print_all_obs
Simulator Tcl command 95
SSM::print_obs
Simulator Tcl command 96
SSM::print_watch_values
Simulator Tcl command 96
SSM::print_watches
Simulator Tcl command 96
SSM::random_bool
Simulator Tcl command 96
SSM::random_char
Simulator Tcl command 96
SSM::random_cycle
Simulator Tcl command 97
SSM::random_inputs
Simulator Tcl command 97
SSM::random_int
Simulator Tcl command 97
SSM::random_real
Simulator Tcl command 97
SSM::random_sign
Simulator Tcl command 98
SSM::random_value
Simulator Tcl command 98
SSM::random_value_matching
Simulator Tcl command 98
SSM::re_alias
Simulator Tcl command 98
SSM::reset
Simulator Tcl command 99
SSM::set
Simulator Tcl command 100
SSM::set_breakpoint
Simulator Tcl command 99
SSM::set_inputs
Simulator Tcl command 101
SSM::set_inputs_file
Simulator Tcl command 101
SSM::set_tolerance
Simulator Tcl command 102
SSM::type
Simulator Tcl command 102
SSM::uncheck
Simulator Tcl command 102
SSM::unset_breakpoint
Simulator Tcl command 103
SSM::unset_inputs_file
Simulator Tcl command 103
SSM::unwatch
Simulator Tcl command 103
SSM::unwatch_node
Simulator Tcl command 104
SSM::watch
Simulator Tcl command 104
SSM::watch_node
Simulator Tcl command 104
SSM::watches
Simulator Tcl variable 106
SSM::what
Simulator Tcl command 105
SsmCloseScadeSimulator 139
SsmDbgStep 141
SsmGetLastError 142
SsmGetOutput 141
SsmGetOutputVector 142
SsmGuiActivate 139
SsmGuiRefresh 139
SsmOpenScadeSimulator 139
SsmSetConnectRetry 140
SsmSetHostName 140
SsmSetInput 140
SsmSetInputVector 141
SsmSetScadeInstallPath 140
SsmStep 141
State scenarios 150

Index

STEP (Tcl keyword) 166
Stop conditions 90

T

Tcl commands
- API-related 73
- SSM::alias (Simulator) 86, 87
- SSM::back (Simulator) 87
- SSM::check (Simulator) 88
- SSM::cycle (Simulator) 90
- SSM::default_inputs (Simulator) 90
- SSM::default_value (Simulator) 91
- SSM::default_value_matching (Simulator) 91
- SSM::get (Simulator) 91
- SSM::get_active_branches (Simulator) 92
- SSM::get_active_matches (Simulator) 92
- SSM::get_active_states (Simulator) 92
- SSM::get_cycle (Simulator) 92
- SSM::get_fired_strong_transitions (Simulator) 93
- SSM::get_fired_weak_transitions (Simulator) 93
- SSM::get_obs_list (Simulator) 93
- SSM::get_one_obs_list (Simulator) 94
- SSM::get_outputs (Simulator) 94
- SSM::go (Simulator) 95
- SSM::print (Simulator) 95
- SSM::print_all_obs (Simulator) 95
- SSM::print_obs (Simulator) 96
- SSM::print_watch_values (Simulator) 96
- SSM::print_watches (Simulator) 96
- SSM::random_bool (Simulator) 96
- SSM::random_char (Simulator) 96
- SSM::random_cycle (Simulator) 97
- SSM::random_inputs (Simulator) 97
- SSM::random_int (Simulator) 97
- SSM::random_real (Simulator) 97
- SSM::random_sign (Simulator) 98
- SSM::random_value (Simulator) 98
- SSM::random_value_matching (Simulator) 98
- SSM::re_alias (Simulator) 98
- SSM::reset (Simulator) 99
- SSM::set (Simulator) 100
- SSM::set_breakpoint (Simulator) 99
- SSM::set_inputs (Simulator) 101
- SSM::set_inputs_file (Simulator) 101
- SSM::set_tolerance (Simulator) 102
- SSM::type (Simulator) 102
- SSM::uncheck (Simulator) 102
- SSM::unset_breakpoint (Simulator) 103
- SSM::unset_inputs_file (Simulator) 103
- SSM::unwatch (Simulator) 103
- SSM::unwatch_node (Simulator) 104
- SSM::watch (Simulator) 104
- SSM::watch_node (Simulator) 104
- SSM::what (Simulator) 105

Tcl scripts
- Custom Registration 77

Tcl variables
- SSM::default_bool (Simulator) 106
- SSM::default_char (Simulator) 106
- SSM::default_int (Simulator) 106
- SSM::default_real (Simulator) 106
- SSM::watches (Simulator) 106

TCL_LIBRARY 174
Traced variables 103, 104
- values of 90
- watch Tcl variable 106

Type check error 176
Type error 174

U

Unknown option 174

Index

W

Watched variables 96