Twin Builder® VHDL-AMS Tutorial

ANSYS, Inc.
Southpointe
2600 Ansys Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
https://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2025 R2
July 2025

Table of Contents

1 - Introduction

Twin Builder is a software package used to design and analyze complex technical systems. Simulation models created with Twin Builder can contain circuit components from different physical domains, block elements, and state machine structures modeled in SML as well as VHDL-AMS.

Twin Builder's simple graphical interface makes even complex models easy to design. Fast and stable simulation algorithms reduce simulation time and provide reliable results.

The various tools used for modeling, simulating, and analyzing are integrated within the Twin Builder application which manages the project files, sets options for both simulation and program environment, runs analyses, and generates reports.

Twin Builder performs calculations for simulation models described in VHDL-AMS. VHDL-AMS stands for **V**ery high-speed integrated circuit **H**ardware **D**escription **L**anguage – **A**nalog **M**ixed **S**ignal. The SML compiler automatically starts the VHDL-AMS simulator if VHDL-AMS components are used in the simulation model. Standard components for VHDL-AMS simulation are available in the **Component Libraries** window: in the **Basic Elements VHDLAMS** folder, the **Digital Elements** folder, and the **Tools** > **Transformations** > **OmniCasters** folder. In addition, components created specially for use with this are located under **"C:\Program Files\ANSYS Inc\v252\AnsysEM\syslib\"**. To use the libraries, rename the **ExampleLibraries._aclb** and **vhdlams_tutorial._aclb** files to **ExampleLibraries.aclb** and **vhdlams_tutorial.aclb**, respectively. The tutorial libraries will be contained in the Systems Libraries in the **Component Libraries** window when you use Twin Builder later.

Tutorial Content

This tutorial describes the special functionality for modeling VHDL-AMS elements within the Twin Builder environment. This includes an introduction to the use of VHDL-AMS components as well as general model development in Twin Builder. The simulation models used in the examples in this tutorial are included on the CD that accompanies the Twin Builder student version. The examples can be loaded into Twin Builder from the included files or can be created by following the step-by-step instructions in this tutorial.

The tutorial does not teach engineering design or cover the complete VHDL-AMS language syntax. These topics are large enough by themselves to warrant several books.

General Description of Contents

Introduction

Describes the general procedure for solving a simulation problem, lists general hints for using this manual, and explains how to install Twin Builder.

VHDL-AMS and VHDL Models in Twin Builder

Describes the modeling of VHDL-AMS models using Schematics, and introduces the use of subsheets and macros.

Automotive Powernet System Example

Describes the modeling and use of VHDL-AMS models in a system-level automotive powernet example. This example is developed in six steps using simplified models for powertrain, battery, alternator, DC-DC converter, and load models in VHDL-AMS. It also illustrates the use of configurations.

Case Studies

Uses six case study examples to provide an in-depth understanding of how to use the VHDL-AMS language for different modeling tasks. These examples describe the use of multiple model architectures for different levels of complexity, different modeling styles, and different behavior. Examples that illustrate the multidomain and digital modeling flexibility of VHDL-AMS are also included.

Model Development

Describes how to create model libraries, models, and packages in Twin Builder, and gives several examples of model development in Twin Builder projects.

VHDL-AMS Language Fundamentals

Describes the VHDL-AMS language syntax.

Appendix

Provides a glossary of frequently used terms and a literature reference list for Twin Builder.

VHDL-AMS Tutorial Conventions

Twin Builder is fully Windows compliant. The basic operation of the windows and user environment is consistent with the Windows standard, and therefore is not described here. For questions about Microsoft Windows, please consult your Windows documentation. The following formatting conventions are used in this documentation:

[Apply]	Button to confirm selection activity
File>Open	Menu sequence to start an action
«Properties»	Text on menus and option fields
QUANTITY	VHDL-AMS keyword
s == v * t;	VHDL-AMS modeling description

☑	Selected check box in a dialog
☐	Cleared check box in a dialog
⦿	Selected option button in a dialog
○	Deactivated option button in a dialog
Note	**Indicates important information – Please note!**

Twin Builder Documentation

Use the following guides and manuals to quickly find help while working with Twin Builder.

Table 1: PDF Manuals

Getting Started	Program functionality at a glance, new functions, step-by-step simulation examples, program conventions. Click **Help** > **Twin Builder Getting Started Guides** or **Help** > **Twin Builder PDFs** > **Twin Builder Getting Started Guides** > **Twin Builder Getting Started Guide**.
Modelica Tutorial	Detailed descriptions and examples of Twin Builder's Modelica functionality (this guide). Listed under **Help** > **Twin Builder PDFs** > **Twin Builder Getting Started Guides** > **Modelica Tutorial**
Installation Guide	Refer to the Ansys EM Suite Installation topic. This topic provides links to the Windows *Installation Guide* and the *Platform Support* website.

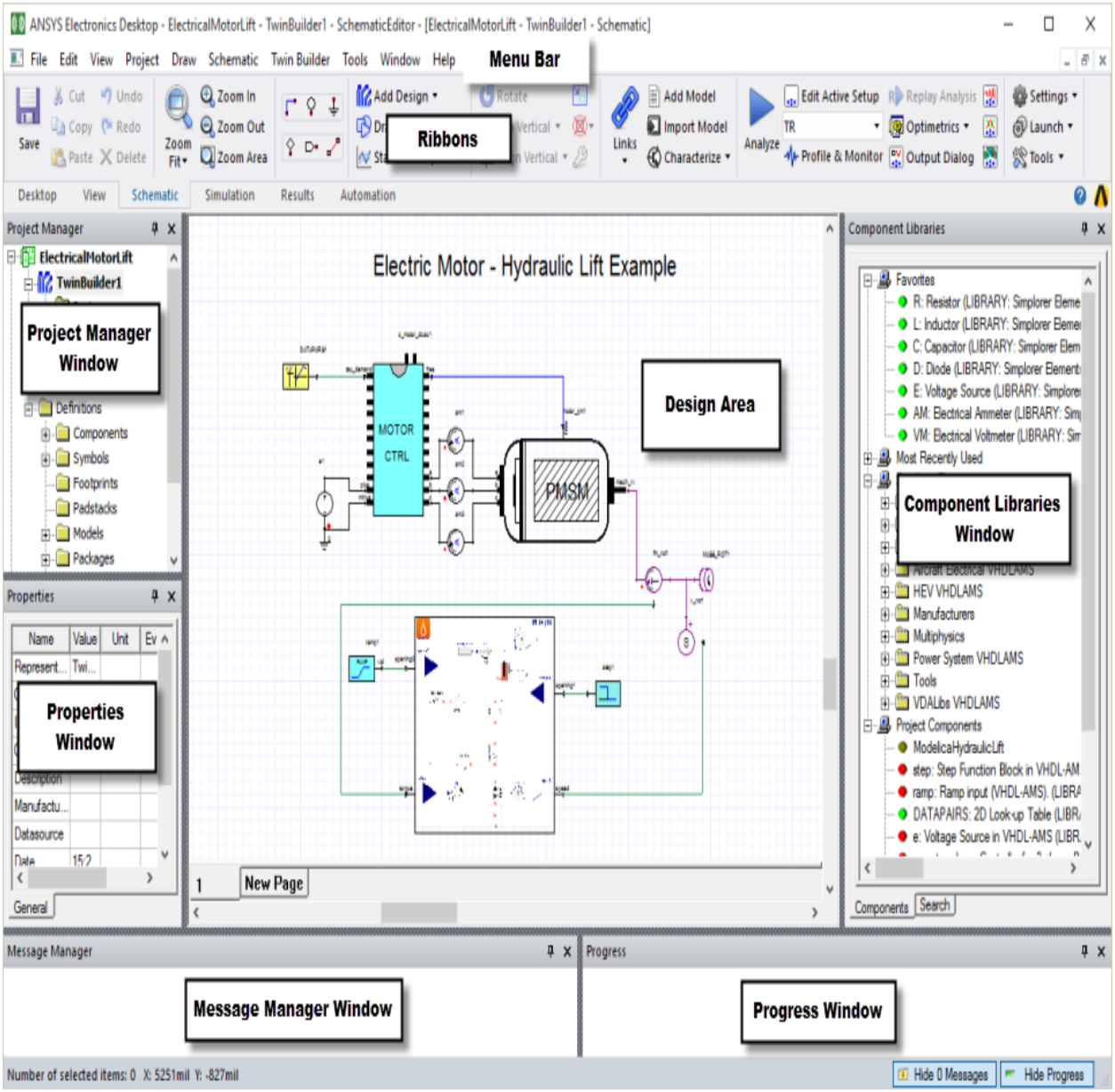
Table 2: Help

Help	Click **Help** > **Twin Builder Help** to access the Help with index and search capabilities available for the Twin Builder programs and installed models. Help also has a description of additional Twin Builder modules such as Interface and Coupling elements, C-interface, and optimization algorithms. Click **Help** > **Twin Builder PDFs** > **Twin Builder Help** to access the PDF of the Twin Builder Help.
Examples	Examples are available for installed Twin Builder models. Open the example by right-clicking on the model in the Project Manager and selecting **Load Example**. If Twin Builder was installed in the default location, Twin Builder application examples are available in **"C:\Program Files\ANSYS Inc\v252\AnsysEM\Examples\Twin Builder\Applications"**

	and the Modelica Tutorial examples are available in **"C:\Program Files\ANSYS Inc\v252\AnsysEM\Examples\Twin Builder\Modelica Tutorial\Tutorial Examples"**

Overview of the Twin Builder Interface

The following figure and the table following describe the major Twin Builder interface elements for an existing project with its associated schematics.



Project	The **Project Manager** pane shows all the components, models, and other

Manager pane and Project tree	elements of each design in the project. Each project has its own expandable Project tree. You can perform many operations on the design elements directly from the **Project Manager** pane.
Message Manager pane	Displays error, informational, and warning messages for the active project.
Progress window	Displays solution progress information.
Properties window	Displays the attributes of a selected object in the active model, such as the object's name, electrical or other associated physical quantities, orientation, and color. Also displays information about a selected command that has been carried out. For example, if a circle was drawn, its command information would include the command's name, the circle's center position coordinates, and the size of its radius.
Design area window	Displays one or more editor windows such as the Schematic Editor, model editors, and symbol editor. It also displays various report windows.
Component Libraries window	Displays, on the **Components** tab, the component categories, including Favorites, Most Recently Used, Simplorer Elements, and Project Components. You can pin the window to make it remain visible or make it visible only when it is being used. These elements are defined as favorites by default: • R (resistor) • L (inductor) • C (capacitor) • D (diode) • E (voltage source) • AM (ammeter) • VM (voltmeter) The Project Components section lists the elements that are active in your projects. If you have created any personal libraries, a Personal Libraries section displays them. The **Component Manager** window also provides a search feature on the **Search** tab.
Menu bar	Provides various menus that enable you to perform Twin Builder tasks, such as managing project files, designs, and libraries; customizing desktop components;

	drawing objects; and setting and modifying project parameters and options.
Ribbons	Provides tabs containing icons that act as shortcuts for executing various commands.
Status bar	Shows current actions and provides instructions.

> **Note:**
>
> The screen layout may be different from what is shown in the picture above, depending on the **View** menu settings.

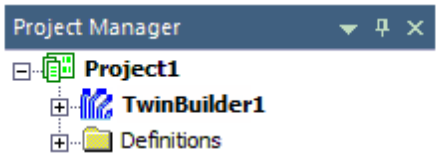
For additional information, please refer to the help and the *Twin Builder Getting Started Guide*.

Starting Twin Builder and Creating a New Project

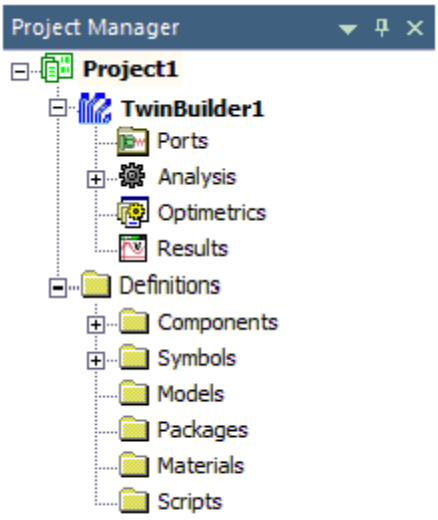
Twin Builder can be started like all Windows applications - by using the **Start** menu on the Windows task bar.

1. Click **Start**, and select **Programs** > **Ansys EM Suite** *version* > **Ansys Twin Builder** *version*.

   By default, opening Twin Builder creates a new project named **Project***n* and inserts a new design named **TwinBuilder***n*, where *n* is the order in which each was added to the current session. A project is a collection of one or more designs saved in a single ***.aedt** file.



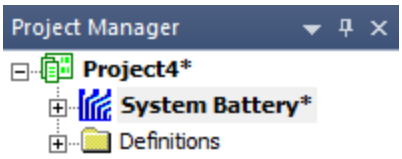
   The new project contains a file structure that organizes design elements such as Ports, Analysis, Optimetrics, and Results. Project Definitions such as Components, Symbols, Models, Packages, Materials, and Scripts are also listed.



2. To rename the design, right-click **Twin Builder***n* in the project tree, and choose **Rename** on the shortcut menu.

   This enables the text cursor for the design name.

3. Type a name of your choosing, and then press **Enter** to complete the change.



Note:

To use the example files provided for this Tutorial, choose **File** > **Open Examples** from the main menu and navigate to the appropriate system example (for example, *system_battery.aedt*) or case study example (for example, *case_study_automotive_alarm_system.aedt*) files in the **Tutorial Examples** subfolder under **"C:\Program Files\ANSYS Inc\v252\AnsysEM\Examples\Twin Builder\VHDLAMS Tutorial\"**.

Save the New Project

To save the new project:

1. Click **File** > **Save**.

   The **Save As** dialog box appears.

2. Use the file browser to find the directory where you want to save the file.
3. Type the desired file name in the **File name** text box.
4. In the **Save as type** list, ensure that **Ansys Electronics Desktop Project File (*.aedt)** is chosen. Project files are given an **.aedt** extension by default.
5. Click **Save** to save the project to the specified location.

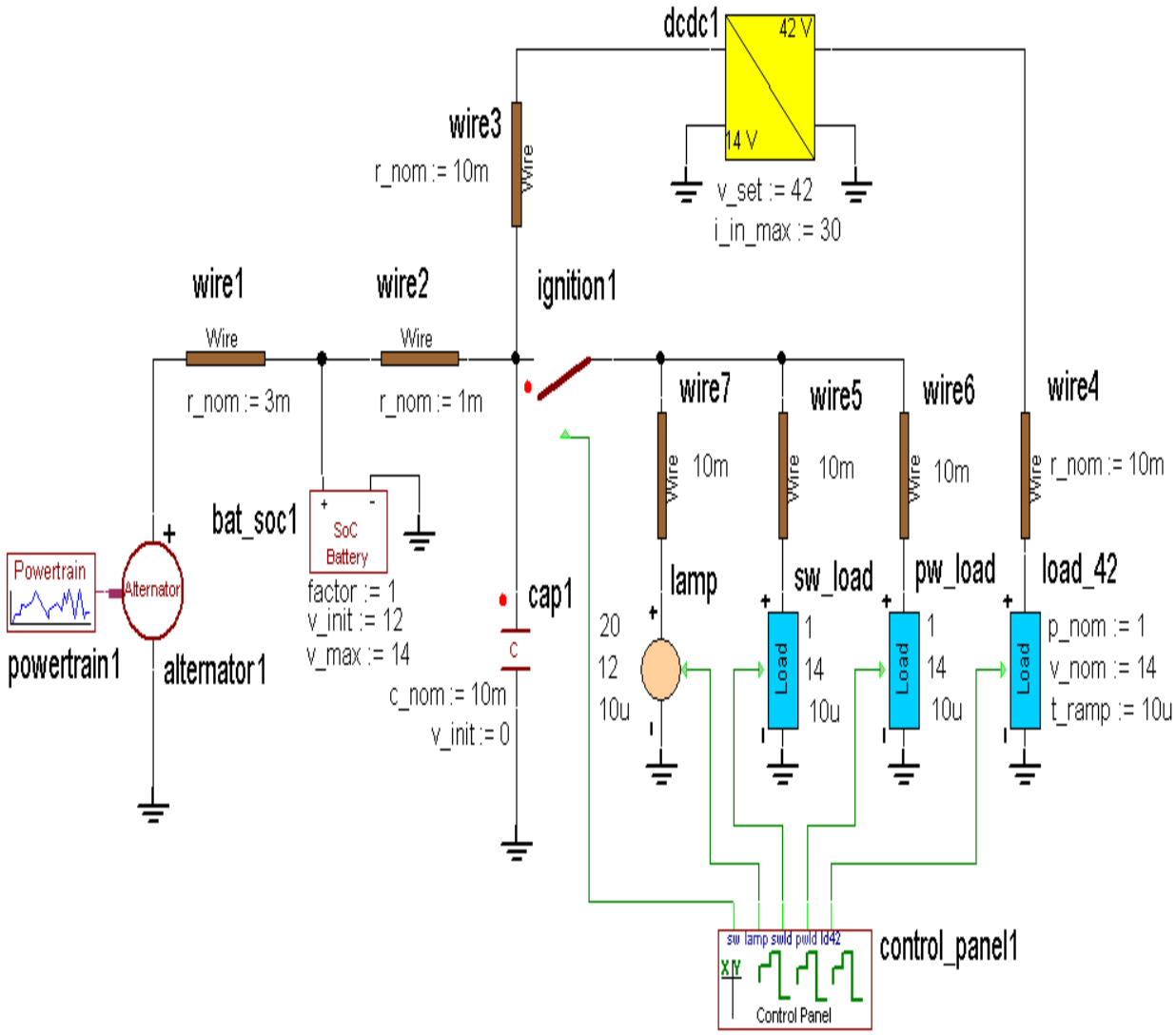
> **Note:**
>
> For further information on any Twin Builder topic, schematic editor commands or windows, you can view Twin Builder‘s context-sensitive help in one of the following ways:
>
> - Click the **Help** button in a pop-up window.
> - Press F1 to open the **Help** window. If you have a dialog box open, the **Help** window opens to a page that describes that dialog box.
> - Use the commands in the **Help** menu.

2 - Automotive Powernet System Example

This basic automotive powernet example describes the development of a simulation model for an automotive powernet in five incremental steps.

1. In the first step, a current source controlled by speed is used to model an alternator. Multi-domain modeling concepts are also introduced.
2. The second step introduces a simple behavioral 14V battery model with state-of-charge calculations that can be used in parallel with the alternator model in the powernet.
3. The third step adds a 42V subnet to the existing 14V powernet with a DC-DC converter model that supplies a 42V admittance load.
4. The fourth step introduces an ignition switch to supply different types of loads such as a switched load, a power load, and a lamp load.
5. The fifth step explains how several speed-profile characteristics, contained in text files, can be used as different architectures for the powertrain model.

The examples used in this chapter were derived from a powernet design developed by the *MSR Consortium*[1]. The models used in the examples are highly simplified to be better suited for the introduction of VHDL-AMS concepts.



This chapter contains information on:

- *Model Overview* about the automotive powernet system example
- Using example sheets on CD
- VHDL-AMS modeling features described in these steps
  - Step 1: Alternator that transforms speed input to electrical output
  - Step 2: Battery with state-of-charge calculation
  - Step 3: DC-DC Converter that supplies a 42V admittance load
  - Step 4: Ignition switch to supply different types of loads

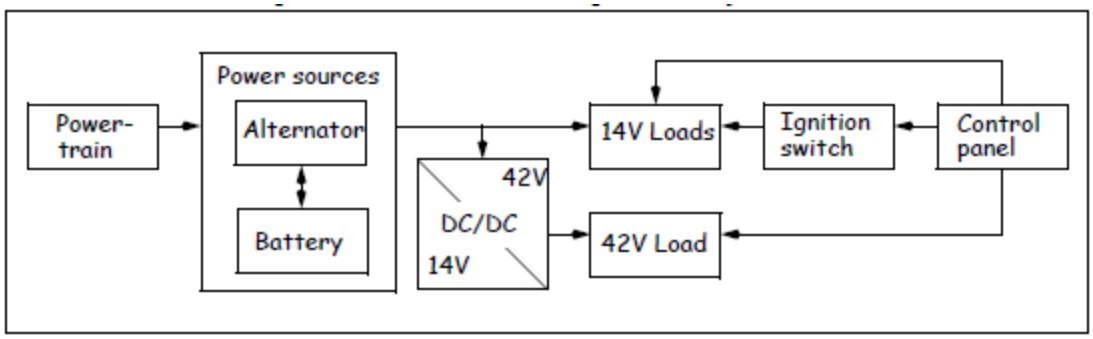
- Step 5: Configuration files
- Step 6: Powertrain with different speed profiles from text files

---

1. Visit the *MSR Consortium* on the Internet: http://www.msr-wg.de/msr.html.

Model Overview

The two main sources of electrical power in an automotive vehicle are the *alternator* and the *battery*. The alternator transforms the mechanical power of the engine into electrical power, which is then distributed to the vehicle's *powernet*. When a car is started, the engine takes some time to power up with the initial power being provided by the battery. Once the engine is up and running, the alternator supplies all the power required by the powernet. When the electric power supplied by the alternator is less than that required by the vehicle, the battery is used as a supplemental source. At other times, when the power generated by the alternator is greater than the vehicle's power requirements, the extra power can be used to charge the battery. This simplified block diagram shows the main components of an automotive powernet system.



- The *powertrain* block is used to describe the speed profile of the vehicle. In this example, the powertrain describes the speed profile of the engine over a period of time.
- The *alternator* and the lead-acid *battery* supply the electrical power to loads that typically operate off of a 14V bus. In this example the loads operate from a 12V battery, and there is a dual bus system with both 14V and 42V.
- The *DC-DC* converter transforms 14V to 42V and supports the 42V bus.
- Three loads are present on the 14V system voltage bus and one load on the 42 V system voltage bus.
- The *ignition switch* is present for the 14V system bus, while the 42V bus is always powered (keyless bus).
- The *control panel* is used to switch on/off or trigger the loads in the car. The control panel can be compared to a user switching on and off the AC/seat warmer/GPS module/Wiper system, etc., controls on the dashboard of the car.

- The non-ideal wires of the powernet are modeled as wires with given resistances. Typically, wire harnesses with specific characteristics are used for the powernet connections.

Using Example Sheets

The project files for the examples described in this chapter and the next, and the required library vhdlams_tutorial.asmd, are provided on the student version CD. Twin Builder is needed to create and simulate the examples. The Student Version has some limitations in the number and types of simulation models available. If an example cannot be executed with the student version, a note is included on the sheet in the project file. In order to run these examples with the student version, some models need to be excluded from the simulation. To do this, select the component instances that need to be excluded and choose **Edit** > **Deactivate (Open)**. The selected components will have a large red "X" superimposed on them, indicating that they are not included in the simulation.

The project files for the examples can be opened in Twin Builder, or created by following the step-bystep instructions in this chapter. All sheets for the powernet example can be accessed from the *system_ examples.asmp* project file, which can be found in the Tutorial Examples folder. The steps of the example can be done sequentially, extending the example sheets step-by-step, or starting with any step since each step explains how to create the complete example. If there are no instructions to change parameter values in the examples, the default values are used.

Creating Examples from Scratch

To create the examples in this chapter from scratch, start Twin Builder and create a new project as explained in "Starting Twin Builder and Creating a New Project " on page 1-6. Once the project is created and Schematic is open, go to the **Component Libraries** window. Navigate to the **Basic Elements VHDLAMS** folder and open it to display the tutorial models.

Each of the examples in this chapter includes a schematic that shows how the models are connected, and step-by-step instructions for:

- Selecting and placing models on the sheet.
- Setting the parameter values for the models.
- Making necessary architecture changes.
- Adding reports to the sheet.

Each example also includes an in-depth discussion of the entity and architecture descriptions for critical models in the system, and an explanation of the VHDL-AMS code contained in those models.

Each example is an extension of the previous example, adding models to the existing schematic.

VHDL-AMS Modeling Features

The following table lists VHDL-AMS features and models used in the case studies (including the example file name), and indicates if the examples can be run in the Student Version (SV).

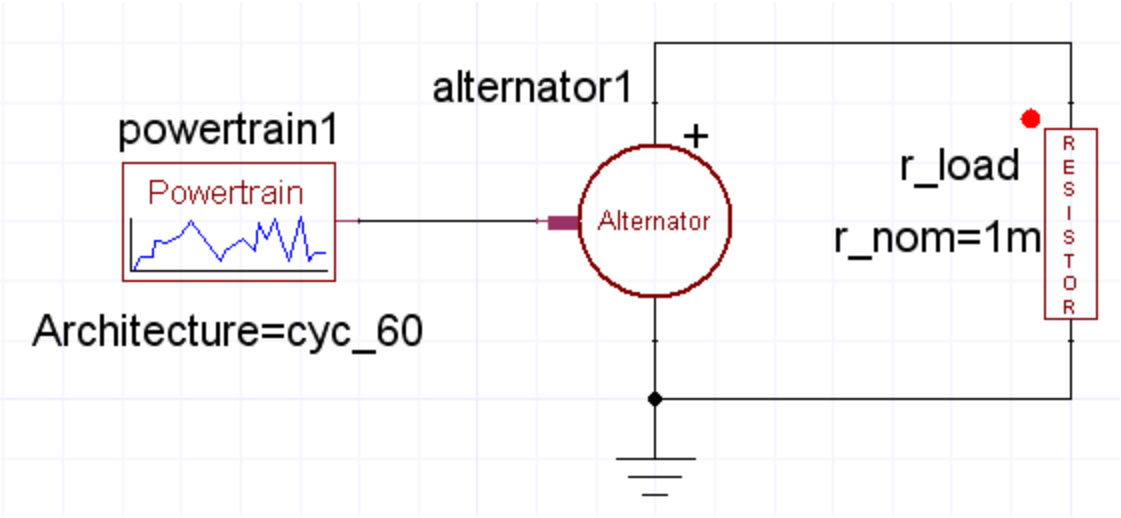
Case	VHDL-AMS Features	Model/Sheet Name	SV
1	Alternator	• Multidomain modeling • **CONSTANT** and **FUNCTION** statements	yes
2	Battery	• **BREAK** and **IF-USE** statements • *'DOT* attribute	yes
3	DC-DC Converter	• Modeling with OmniCasters • Solvability Criteria • *'INTEG* attribute	partly
4	Ignition switch	• Mixed-signal modeling • **SIGNAL** statement	partly
5	Control Panel	• Modeling with several configurations and architectures • **PROCESS** statement	no
6	Powertrain	• **FILE** I/O statements • Multidomain modeling using Domain-to-Domain model	yes

Step 1: Modeling the Alternator

The first step in the powernet example, found in the file **system_alternator.aedt**, introduces concepts in multi-domain modeling with a simple alternator model. The alternator is modeled as a current source that transforms engine speed input to electrical output that is then used to supply a resistive load.

The **Powertrain** block provides the speed profile of the engine through a mechanical pin output. This mechanical pin is connected as an input to the **Alternator** model. The **Alternator** model provides the electrical power to the entire system (represented by the load resistor).

The following figure shows the *system_alternator* schematic. The *cyc_60* architecture is used for the powertrain model to provide a drive cycle for a period of 60 seconds.

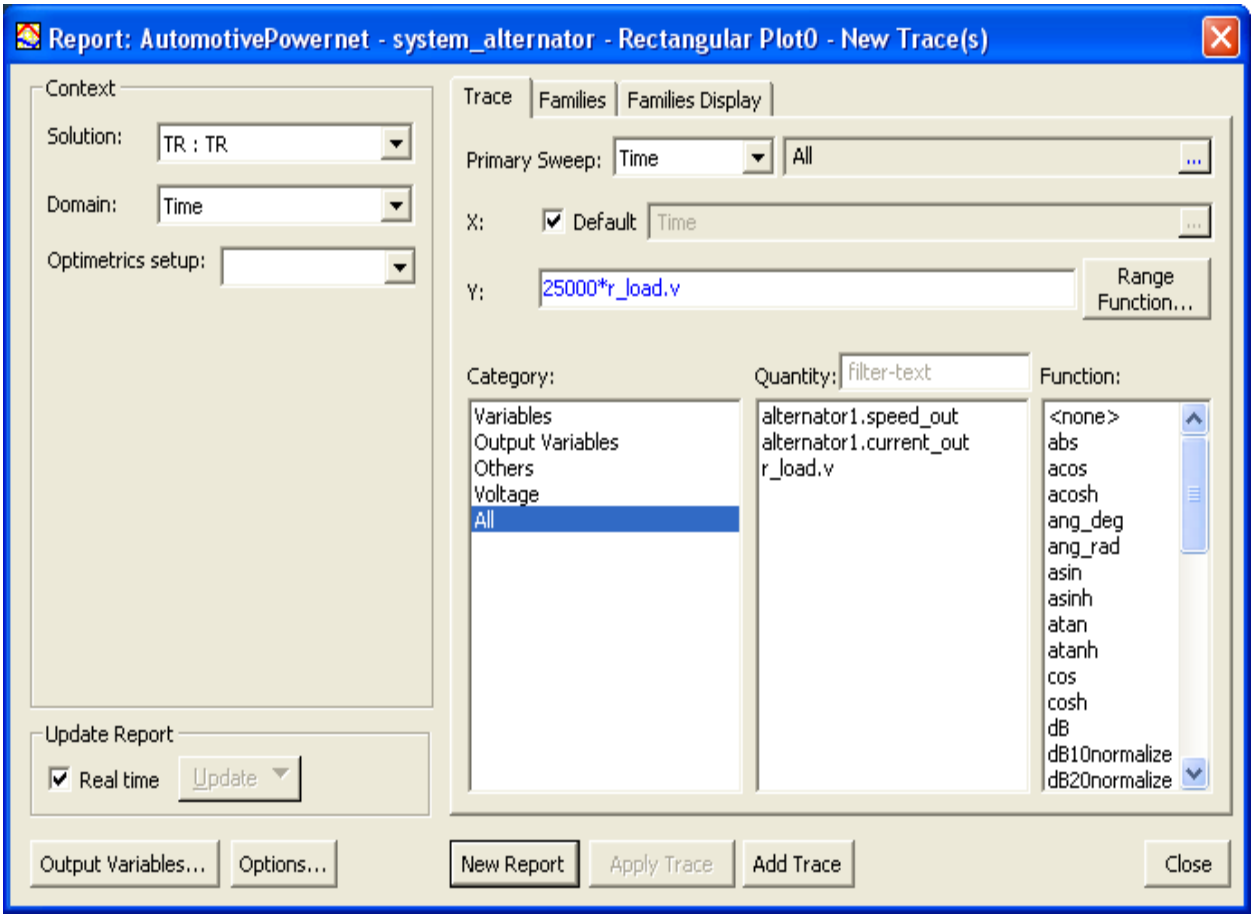


> **Note:**
>
> This example does not provide comprehensive results for all possible speed profiles. It is used to introduce the general model design that will be completed in the six modeling steps of the Powernet System Example.

Creating the Simulation Model

1. **Place and arrange all components shown in the figure above.** *Alternator*, *Powertrain*, and *Resistor* are VHDL-AMS models from the **vhdlams_tutorial** folder. (Before opening Twin Builder, go to **"C:\Program Files\ANSYS Inc\v252\AnsysEM\syslib\"** and rename the **ExampleLibraries._aclb** and **vhdlams_tutorial._aclb** files to **ExampleLibraries.aclb** and **vhdlams_tutorial.aclb**, respectively. The tutorial libraries will be contained in the Systems Libraries in the **Component Libraries** window when you open Twin Builder later.) Choose **Draw** > **Ground** or type Ctrl+G to place the ground node.
2. **Connect the models.** Place the cursor on one of the model pins to activate the wire cursor and click to enter wire mode. Connect the components as indicated in the figure, setting the beginning, the corners, and the end of a wire with the mouse.
3. **Define the resistor parameters.** Double-click the resistor symbol to open the **Properties** dialog box. On the Parameter Values tab, change the InstanceName from *res1* to *r_load*. (The name also can be changed by clicking on it, entering the new name, and then pressing **Enter**.) Click in the **«Value»** field, and enter *1.0m* for the resistance value. On the Property Displays tab, click **Add** to add a new property to the list of displayed properties. In the **«Name»** field choose *r_nom*, and in the **«Visibility»** field choose *Both*. Click **OK** to apply the changes.

4. **Define the powertrain parameters.** Double-click the powertrain symbol to open the **Properties** dialog box. On the Parameter Values tab, change the SimulatorModel from *cyc_manhattan* to *cyc_60*. Click in the **«Value»** field, and select *cyc_60* from the list. On the Property Displays tab, click **Add** to add a new property to the list of displayed properties. In the **«Name»** field choose an option, and in the **«Visibility»** field choose *Both*. Click **OK** to apply the changes.
5. **Define the simulation parameters.** Choose **Twin Builder** > **Add Solution Setup** > **Transient** to open the Transient Analysis Setup dialog box, and change the default value for simulation **End Time** to *60* seconds, **Min Time Step** to *1ms*, and **Max Time Step** to *10 ms*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Trapezoid**. Click **OK** to apply the changes.
6. **Place and arrange a 2D Rectangular Plot to display simulation results.** Select **Draw** > **Report** > **Rectangular Plot**, and then click and drag to place and size the plot on the sheet.
7. **Open the simulation Output list.** Choose **Twin Builder** > **Output Dialog** to open the **Output** dialog box.
8. **Define the simulation output quantities.** Click the **+** next to the components to reveal the list of available quantities. Check the output boxes of *r_load.v, alternator1.current_out,* and *alternator1.speed_out* to add these to the list of displayed quantities. Click **OK** to apply the changes.
9. **Add traces to the plot.** Double-click the on-sheet plot to open the **Report** dialog box. Select each of the output quantities defined above and change the scaling used for the traces as shown in “"Results - Alternator Model" on page 2-13 . For example in the Y field, multiply *r_load.v* by 25000. Click **Add Trace** to add it to the plot. Click **Close** to apply the changes.



10. **Save the project**. Choose **File** > **Save As**, enter a file name and directory, and click **OK**.

Overview of Parameter Values - Alternator Model

Model Name	Parameter Values	Tab/Library	Architecture
powertrain1	no parameters	Projects/vhdlams_tutorial.smd	cyc_60
alternator1	no parameters	Projects/vhdlams_tutorial.smd	behav
r_load	r_nom [Ω]=1m	Projects/vhdlams_tutorial.smd	behav

Alternator Model

The alternator model is a multi-domain model that uses the electrical (*ELECTRICAL*) and mechanical (*ROTATIONAL_V*) domains. Consequently, the *electrical_systems* and *mechanical_systems* packages that are defined in the *IEEE* library must be used. (To view the entity and architecture descriptions for a model, in the **Project Manager Definitions** > **Models**

folder, double-click the desired model to open it in the VHDL-AMS Editor.) The following statements in the model definition include the two packages:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
USE IEEE.MECHANICAL_SYSTEMS.ALL;
```

Entity Description - Alternator Model

The model accepts the speed input from a speed source through a *ROTATIONAL_V* terminal and outputs the corresponding electrical current and voltage through two *ELECTRICAL* terminals. It also provides the speed, torque, and current values as outputs from the model. The following table summarizes the port definitions in the model interface:

Interface	Name	Property
TERMINAL	rot_in	*ROTATIONAL_V*
	p_out	*ELECTRICAL*
	m_out	*ELECTRICAL*
QUANTITY	speed_out	**OUT***VELOCITY*
	current_out	**OUT***CURRENT*
	torque_out	**OUT***TORQUE*

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY Alternator IS
 PORT(TERMINAL rot_in : ROTATIONAL_V;
 TERMINAL p_out, m_out : ELECTRICAL;
 QUANTITY speed_out : OUT VELOCITY;
 QUANTITY current_out : OUT CURRENT;
 QUANTITY torque_out : OUT TORQUE);
 END ENTITY Alternator;
```

Architecture Description - Alternator Model

The model's architecture describes the method in which the alternator transforms the mechanical energy to electrical energy. This alternator is a simple model assumed to have no inertia; therefore, the torque depends on the input speed and the electrical power output. The alternator uses a characteristic function to obtain an output current value based on the input speed as shown in the following table:

Speed (rpm)	0	1000	1500	2000	3000	4000	6000	10000	14000	¥
Current (A)	0	0	30	60	70	85	90	100	105	105

The model defines a constant value of *OM2N* that is equal to 60/2π for transforming the value of rotational speed from rad/s to rpm. The VHDL-AMS statement for defining the constant value is as follows:

```
CONSTANT OM2N : REAL := REAL := 60.0/MATH_2_PI;
```

The model calculates a torque output based on the following equations:

```
Torque * Omega = Voltage * Current
Torque = Voltage * Current/Omega
```

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE behav OF Alternator IS
 CONSTANT OM2N : REAL := 60.0/MATH_2_PI; -- This is equal to 60/2*pi
 QUANTITY v ACROSS i THROUGH p_out TO m_out;
 QUANTITY omega ACROSS torque THROUGH rot_in TO ROTATIONAL_V_REF;

 FUNCTION CH (x: REAL) RETURN real IS
 CONSTANT X1 : REAL := 1000.0 ;
 CONSTANT Y1 : REAL := 0.0 ;
 CONSTANT X2 : REAL := 1500.0 ;
```

```
CONSTANT Y2 : REAL := 30.0 ;
CONSTANT X3 : REAL := 2000.0 ;
CONSTANT Y3 : REAL := 60.0 ;
CONSTANT X4 : REAL := 3000.0 ;
CONSTANT Y4 : REAL := 70.0 ;
CONSTANT X5 : REAL := 4000.0 ;
CONSTANT Y5 : REAL := 85.0 ;
CONSTANT X6 : REAL := 6000.0 ;
CONSTANT Y6 : REAL := 90.0 ;
CONSTANT X7 : REAL := 10000.0 ;
CONSTANT Y7 : REAL := 100.0 ;
CONSTANT X8 : REAL := 14000.0 ;
CONSTANT Y8 : REAL := 105.0 ;
VARIABLE result: REAL := 0.0 ;
```

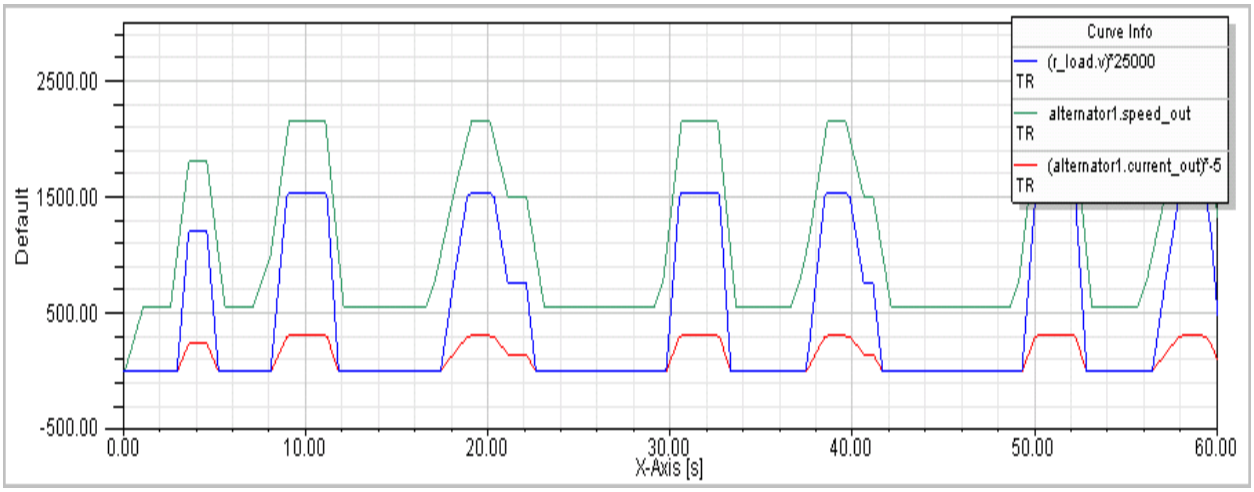
```
 BEGIN
IF X < X1 THEN result:= Y1 ;
ELSE
IF X < X2 THEN result:= Y1+(x - X1) / (X2-X1) * (Y2-Y1);
ELSE
IF X < X3 THEN result:= Y2+(x - X2) / (X3-X2) * (Y3-Y2);
ELSE
IF X < X4 THEN result:= Y3+(x - X3) / (X4-X3) * (Y4-Y3);
ELSE
IF X < X4 THEN result:= Y3+(x - X3) / (X4-X3) * (Y4-Y3);
ELSE
IF X < X5 THEN result:= Y4+(x - X4) / (X5-X4) * (Y5-Y4);
ELSE
IF X < X6 THEN result:= Y5+(x - X5) / (X6-X5) * (Y6-Y5);
```

```
ELSE
IF X < X7 THEN result:= Y6+(x - X6) / (X7-X6) * (Y7-Y6);
ELSE
IF X < X8 THEN result:= Y7+(x - X7) / (X8-X7) * (Y8-Y7);
ELSE result:=Y8;
END IF;
END IF;
END IF;
END IF;
END IF;
END IF;
END IF;
END IF;
END IF;
RETURN result;
END FUNCTION CH;

BEGIN
i == -CH(omega * OM2N);
torque == -i*v / (omega + 1.0E-12);
speed_out == omega*OM2N;
torque_out == torque;
current_out == i;
END ARCHITECTURE behav;
```

> **Note:**
>
> Usually, subprogram modules like **FUNCTION** declarations and reusable declarations such as **TYPE/CONSTANT** declarations are defined in a package. In this model, the function and constant definitions are within the model architecture since they need not have a scope outside of the model.

Results - Alternator Model

Start the simulation using Select **Twin Builder** > **Analyze** menu command. The simulation model is compiled and evaluated. The on-sheet plot shows the simulation results. The alternator outputs an electrical current based on the input speed from the powertrain block.

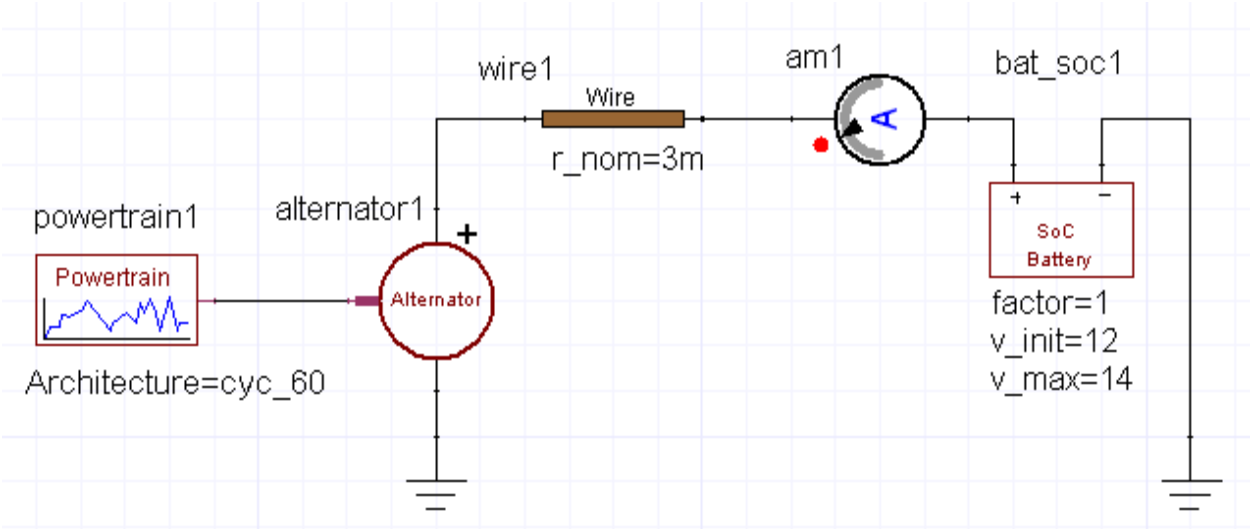


Step 2: Battery Model

The second step of the example, found in the file **system_battery.aedt**, adds a simple behavioral 14V battery model that can be used in parallel with the alternator model in the powernet.

The alternator and battery models provide the electrical power to the entire system. The battery characteristics of interest are the current, voltage, and the State-of-Charge (SoC). The SoC of the battery is expressed as a value between '0' and '1' and suggests how much charge from the battery is used up during a drive cycle. When the alternator is able to support the loads and drive current into the battery, the battery SoC increases. On the other hand, if the battery is powering the electrical system of the vehicle, its SoC decreases. The graph of the SoC indicates how the SoC varies continuously as different loads are switched on and off.

The following figure shows the circuit constructed by adding the battery model, the wire component, and an ammeter to the *system_alternator.aedt* schematic sheet.



Creating the Battery Simulation Model

1. **Place and arrange all components shown in the figure above.** (The schematic from the previous step can be used as a basis, if desired.) **Alternator**, **Powertrain**, **Battery**, and **Wire** are VHDL-AMS models from the **vhdlams_tutorial folder under System Libraries** in the **Component Libraries window**. The **ammeter** is a model from the **Basic Elements VHDLAMS>Measurement>Electrical** library folder. Type Ctrl+G to place the ground nodes.
2. **Connect the models.** Place the mouse cursor on a model pin to activate the wire cursor and click to enter wire mode. Connect the components as indicated in the figure.
3. **Define the Wire component parameters.** Double-click the **Wire** component to open its **Properties** dialog box. Click in the **r_nom** property's **Value** field, and enter *3m* for the resistance value. Click **OK** to apply the changes. Use the Property Displays tab to display r_nom with its value as shown above.
4. **Define the simulation parameters.** Choose **Twin Builder** > **Add Solution Setup** > **Transient** to open the Transient Analysis Setup dialog box, and set the default value for simulation **End Time** to *60* seconds, **Min Time Step** to *1ms*, and **Max Time Step** to *10 ms*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Trapezoid**. Click **OK** to apply the changes.
5. **Place and arrange two 2D Rectangular Plots to display simulation results.** Select **Draw** > **Report** > Rectangular Plot, and then click and drag to place and size the plot on the sheet.
6. **Open the simulation Output list.** Choose **Twin Builder** > **Output Dialog** to open the **Output** dialog box.

7. **Define the simulation output quantities.** Click the **"+"** next to the components to reveal the list of available quantities. Check the output boxes of *alternator1.speed_out*, *bat_soc.v_out*, and *am1.i* to add these to the list of displayed quantities. Click **OK** to apply the changes.
8. **Add traces to the plot.** Double-click the on-sheet plot to open the **Report** dialog box. Select each of the output quantities defined above and change the scaling used for the traces as shown in "Results - Battery" on page 2-19 . Click **Add Trace** to add it to the plot. Click **Close** to apply the changes.
9. Repeat steps 6, 7, and 8 for the second plot, but check the box of *bat_soc.soc_out* as output.
10. **Save the sheet**. Choose **File** > **Save As**, enter a file name and directory, and click **OK**.

Overview of Parameter Values - Battery Model

Model Name	Parameter Values	Tab/Library	Architecture
powertrain1	no parameters	Projects/vhdlams_tutorial.smd	cyc_60
alternator1	no parameters	Projects/vhdlams_tutorial.smd	behav
bat_soc1	factor=1; v_init[V]=12; v_max [V]=14	Projects/vhdlams_tutorial.smd	behav
wire1	r_nom [Ω]=3m	Projects/vhdlams_tutorial.smd	behav
am1	no parameters	AMS/basic_vhdlams.smd	behav

Battery Model

The battery model uses the electrical (*ELECTRICAL*) domain; consequently, the *electrical_systems* package that is defined in the *IEEE* library must be included. The following statements in the model definition include the package:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
```

Entity Description - Battery

The entity description of the battery model uses static **GENERIC** parameters (constant value inputs evaluated only at the beginning of the simulation) and terminals in the interface. The

battery model has three parameters: factor, initial voltage, and maximum voltage, as defined in the **GENERIC** statement. The parameter *factor* refers to a scaling factor that can scale the capacitance values. The battery provides its electrical output through a pair of electrical terminals, the voltage across the battery as an output through *v_out*, and the state of charge of the battery through *soc_out* as defined in the **QUANTITY** statement.

Interface	Name	Property	Default Value
GENERIC	factor	REAL	1.0
	v_init	VOLTAGE	12.0
	v_max	VOLTAGE	14.0
TERMINAL	p	ELECTRICAL	
	m	ELECTRICAL	
QUANTITY	v_out	VOLTAGE	
	soc_out	REAL	

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY bat_soc IS
 GENERIC(
 factor: REAL := 1.0;
 v_init : VOLTAGE := 12.0;
 v_max : VOLTAGE := 14.0);
 PORT(
 TERMINAL p,m : ELECTRICAL;
 QUANTITY soc_out : OUT REAL := 0.0;
 QUANTITY v_out : OUT VOLTAGE := 0.0);
END ENTITY bat_soc;
```

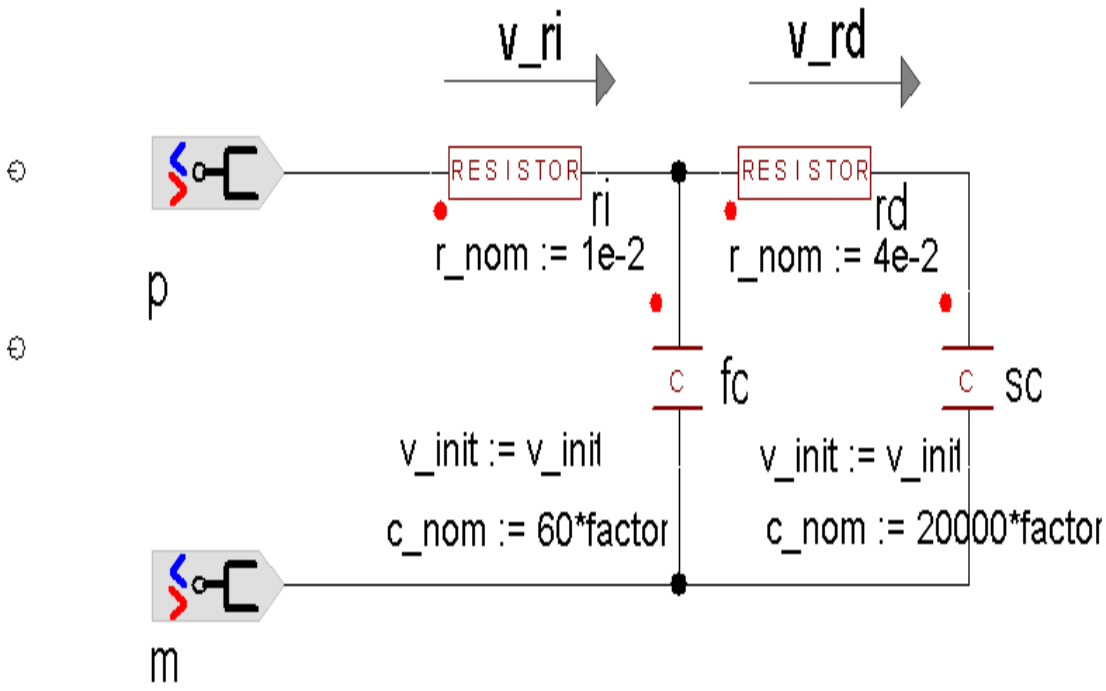
Architecture Description - Battery

In a lead-acid battery, chemical energy is transformed to electrical energy at the battery electrodes through chemical reaction of the acid molecules with the electrode materials. The

chemical reaction at the battery electrodes results in a lower concentration of the lead acid near the electrodes during discharge.

The diffusion of the acid molecules between regions of high and low concentration is modeled with three components: the diffusion resistor, a 'slow' capacitor, and a 'fast' capacitor. Since the acid near the battery electrodes is consumed faster, its concentration is lower. This is modeled as a fast capacitor that discharges quickly. The acid concentration farther from the electrodes is not consumed as fast, so it is modeled as a 'slow' capacitor that discharges slowly. The actual diffusion of the acid molecules between the regions of low and high acid concentrations is modeled with a diffusion resistor. The electrical resistance of the electrodes is modeled as the internal resistance of the battery.

The *behav* architecture of the battery model describes the behavioral implementation of the following circuit:



The internal resistance (ri=10m), diffusion resistance (rd=40m), fast capacitor (fc=60*factor), and slow capacitor (sc=20000*factor) are constant values. The model evaluates the circuit based on the following four equations:

v_ri = i_ri * ri, d(v_fc)/dt = 1/(fc*factor) * i_fc

```
v_rd = i_rd * rd, d(v_sc)/dt = 1/(sc*factor) * i_sc
```

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE behav OF bat_soc IS
 TERMINAL t1, t2: ELECTRICAL;
 CONSTANT ri: RESISTANCE := 1.0e-2;
 CONSTANT fc: CAPACITANCE := 60.0;
 CONSTANT rd: RESISTANCE := 4.0e-2;
 CONSTANT sc: CAPACITANCE := 2.0e4;
 CONSTANT init_charge : CHARGE := ((fc + sc)*factor)*v_init;
 CONSTANT max_charge : CHARGE := ((fc + sc)*factor)*v_max;
 QUANTITY v_ri ACROSS i_ri THROUGH p TO t1;
 QUANTITY v_fc ACROSS i_fc THROUGH t1 TO m;
 QUANTITY v_rd ACROSS i_rd THROUGH t1 TO t2;
 QUANTITY v_sc ACROSS i_sc THROUGH t2 TO m;
 QUANTITY v ACROSS p TO m;
 QUANTITY ri_val : RESISTANCE := ri;
 QUANTITY total_charge : CHARGE := init_charge;
BEGIN
 BREAK v_fc => v_init, v_sc => v_init;
 BREAK i_ri => v_init/ri;
 BREAK soc_out => init_charge/max_charge;
 IF (i_ri < 0.0) OR (soc_out < 0.1) USE
 ri_val == (2.0*ri)**2/(1.0-soc_out);
 ELSE
 ri_val == (2.0*ri)**2/soc_out;
 END USE;
```

```
v_ri == i_ri * ri_val;
total_charge == ((fc*factor)*v_fc) + ((sc*factor)*v_sc);
soc_out == total_charge/max_charge;
v_fc'dot == 1.0/(fc*factor) * i_fc;
v_rd == i_rd * rd;
v_sc'dot == 1.0/(sc*factor) * i_sc;
v_out == v;
END ARCHITECTURE behav;
```

The state of charge of the battery is based on the initial and maximum voltage values of the battery.

The **IF-USE** statement is used to limit the charging of the battery when its SoC value approaches one (1), and the internal resistor is used to limit the discharging of the battery when its SoC value approaches zero (0). Unrealistic values of SoC can occur by forcing/drawing large currents to/from the battery.

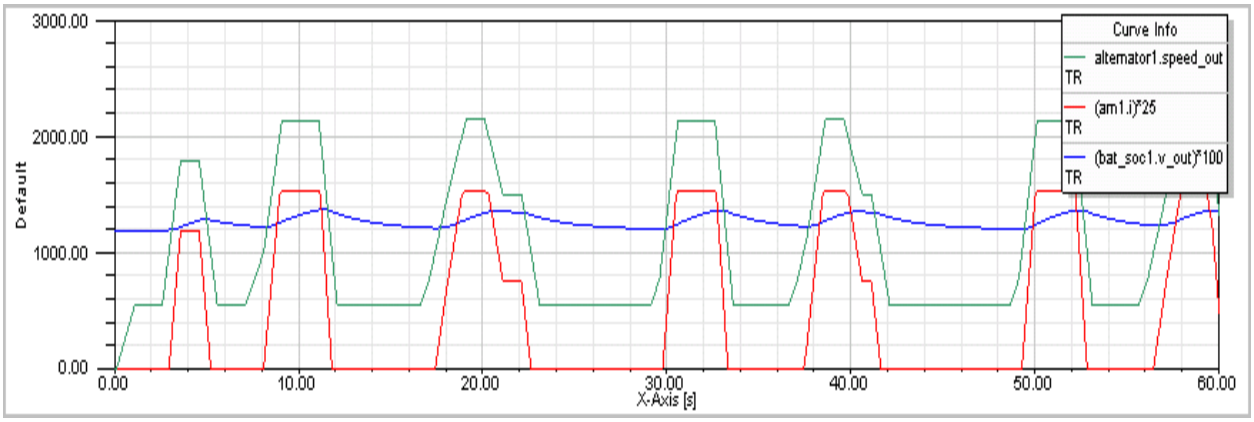
The *'DOT* attribute is used to evaluate the derivative of an analog quantity in a simultaneous differential algebraic equation.

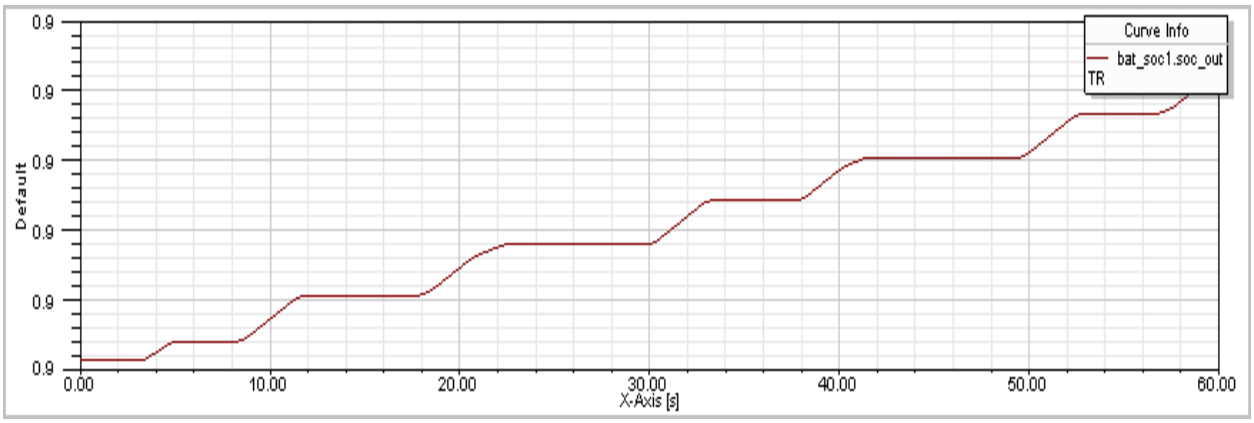
To model the discontinuous behavior of the battery model, **BREAK** statements are used to indicate the occurrence of a discontinuity to the analog solver. Here, since the first derivative of the *v_fc* and *v_sc* quantities may be discontinuous the **BREAK** statement uses initial voltage values that are specified by *v_init*.

Results - Battery

Select **Simulation** > **Start** to start the simulation. The simulation model is compiled and evaluated. The plots show the simulation results on the sheet.

The alternator outputs an electrical current based on the input speed from the *Powertrain* model. The State of Charge (SoC) of the battery is expressed as a value between “0” and “1”. When the alternator drives current into the battery, the SoC of the battery increases. If the battery is powering the electrical system of the vehicle, its SoC decreases.



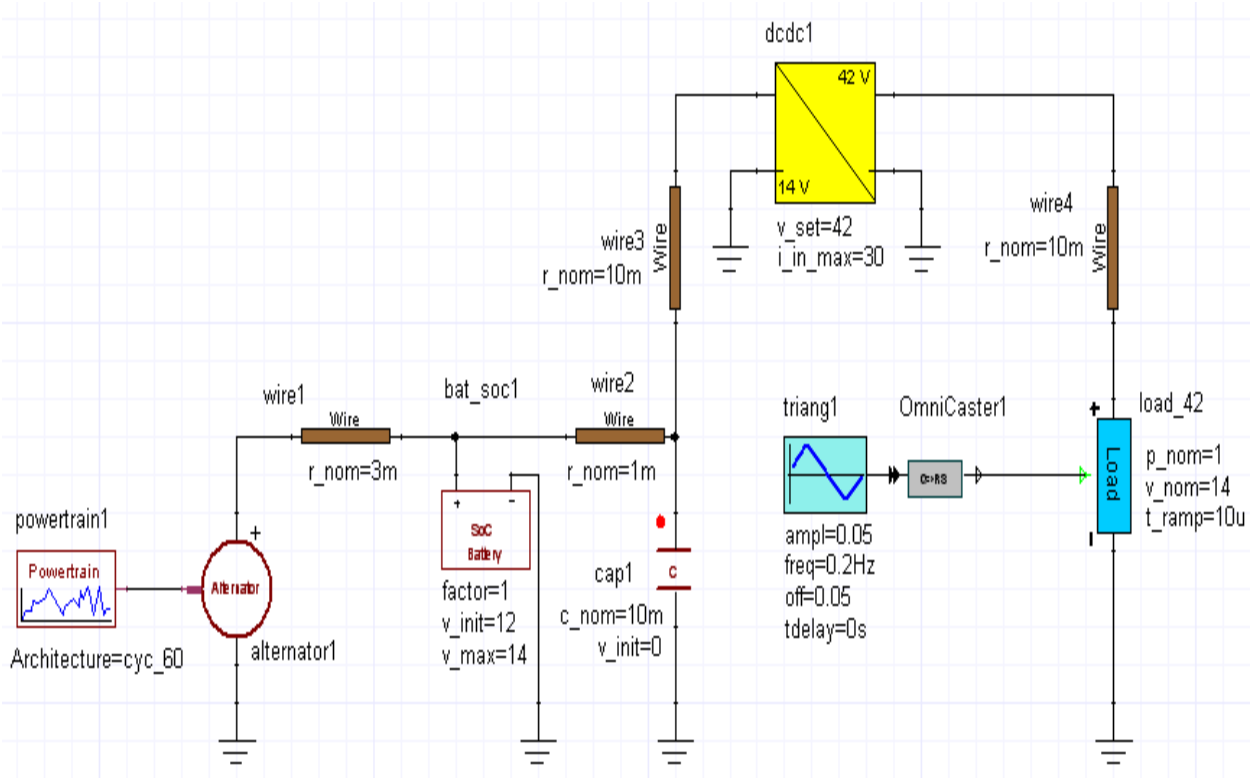


Step 3: DC-DC Converter

The third step of the example, found in the file **system_dcdc.aedt**, adds a 42V subnet to the existing 14V powernet with a DC-DC converter model that supplies a 42V admittance load.

The system voltage of 14V is stepped-up to 42V by a DC-DC converter to provide the dual bus voltage. The 42V bus supplies an admittance load that is switched on and off by a control signal with a triangular waveform characteristic. The use of *OmniCaster* models is illustrated by connecting a REAL QUANTITY output of a triangular time function model with the REAL SIGNAL input required by the admittance load. If the pins of these models are connected directly, a flexible OmniCaster will automatically be inserted between them. The flexible OmniCaster model will choose the correct transformation based on the data types of the pins connected to it. Or the Real-Real OmniCaster model (from the **Tools>OmniCasters>Quantity-Signal** folder) can be placed on the sheet and connected as shown in the figure below.

The following figure shows the *system_battery.aedt* design extended with the DC-DC converter model and the 42V subnet:



Creating the DC-DC Converter Simulation Model

1. Place and arrange all components shown in the figure above. (The schematic from the previous step can be extended.) *Alternator*, *Powertrain*, *Battery*, *Capacitor*, *DC-DC Converter*, *Load*, and *Wire* are VHDL-AMS models from the **vhdlams_tutorial folder under System Libraries** in the **Component Libraries window**. The OmniCaster model is inserted automatically by drawing a connection from the *Triangular Wave* component to the *Load*. The *Triangular Wave* is a model from the *Time Functions* folder under the *Tools* folder of the *basic_vhdlams* library. Type Ctrl+G to place a ground node.
2. **Connect the models.** Connect the components as indicated in the figure, setting the beginning, the corners, and the end of a wire with the mouse.
3. **Define wire parameters.** For each wire model, double-click the wire symbol to open its Properties dialog box. Click in the **r_nom Value** field, and enter the corresponding resistance value. Click **OK** to apply the changes.
4. **Define parameters of the capacitor.** Double-click the capacitor symbol to open its **Properties** dialog box. Click in the **c_nomValue** field, and enter *10m* as capacitance. Click **OK** to apply the changes.
5. **Define parameters of the load.** Double-click the load symbol to open its **Properties** dialog box and change the model name to *load_42.* The model uses the *admittance* architecture (default setting). Click **OK** to apply the changes.

6. **Define parameters of the triangular wave.** Double-click the triangular wave model to open its **Properties** dialog box. Click in the appropriate **Value** fields and enter *0.05* as **off**set, *0.05* as **ampl**itude, and *0.2* as **freq**uency value. Click **OK** to apply the changes.
7. **Define the simulation parameters.** Choose **Twin Builder>Add Solution Setup>Transient** to open the Transient Analysis Setup dialog box, and set the default value for simulation **End Time** to *5* seconds, **Min Time Step** to *1ms*, and **Max Time Step** to *1 s*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Trapezoid**. Click **OK** to apply the changes.
8. **Place and arrange two 2D Rectangular Plots to display simulation results.** Select **Draw** > **Report** > **Rectangular Plot**, and then click and drag to place and size the plot on the sheet.
9. **Open the simulation Output list.** Choose **Twin Builder>Output Dialog** to open the **Output** dialog box.
10. **Define the simulation output quantities.** Click the **"+"** next to the components to reveal the list of available quantities. Check the output boxes of *load_42.i*, *load_42.v*, and *load_42.ctrl_ramp* to add these to the list of displayed quantities. Click **OK** to apply the changes.
11. **Add traces to the plot.** Double-click the on-sheet plot to open the **Report** dialog box. Select each of the output quantities defined above and change the scaling used for the traces as shown in ***Results - DC-DC Converter"***. Click **Add Trace** to add it to the plot. Click **Close** to apply the changes.
12. Repeat the steps 9, 10 and 11 for the second plot, but check the box of *bat_soc.soc_out* as output.
13. **Define the simulation parameters.** Choose **Twin Builder>Add Solution Setup>Transient** to open the Transient Analysis Setup dialog box, and set the value for simulation **End Time** to *5* seconds, **Min Time Step** to *1ms*, and **Max Time Step** to *1 s*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Trapezoid** and **Maximum number of iterations** to *45*. Click **OK** to apply the changes.
14. **Save the sheet**. Choose **File>Save As**, enter a file name and directory, and click **OK**.

Overview of Parameter Values - DC-DC Converter Model

Model Name	Parameter Values	Library	Architecture
powertrain1	no parameters	vhdlams_tutorial	cyc_60
alternator1	no parameters	vhdlams_tutorial	behav
bat_soc1	factor=1; v_init[V]=12; v_max[V]=14	vhdlams_tutorial	behav
wire1	r_nom [Ω]=3m	vhdlams_tutorial	behav
wire2	r_nom [Ω]=1m	vhdlams_tutorial	behav

Model Name	Parameter Values	Library	Architecture
wire3	r_nom [Ω]=10m	vhdlams_tutorial	behav
wire4	r_nom [Ω]=10m	vhdlams_tutorial	behav
cap1	c_nom[F]=10m; v_init[V]=0	vhdlams_tutorial	behav
dcdc1	v_set[V]=42; i_in_max[A]=30	vhdlams_tutorial	behav
triang1	ampl=0.05; freq[Hz]=0.2; off=0.05; tdelay[s]=0	basic_vhdlams	behav
OmniCaster1	ts[s]=1m; *inp=trian1.val*	Tools/transformations	behav
load_42	p_nom[W]=1; v_nom[V]=14; t_ramp [s]=10u; *on_ctrl=OmniCaster1.val*	vhdlams_tutorial	admittance

DC-DC Converter Model

The DC-DC converter uses the electrical (ELECTRICAL) domain; consequently, the *electrical_ systems* package that is defined in the IEEE library must be included. The following statements in the model definition include the package:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
```

Entity Description - DC-DC Converter

The entity description of the DC-DC converter uses static parameters (constant value inputs evaluated only at the beginning of the simulation) and terminals in the interface. The DC-DC converter has two parameters for the set voltage and maximum input current as indicated in the **GENERIC** statement. These parameters are of type REAL with specific default values.

The model also has four terminals, a pair of plus (+) and minus (–) terminals at the input (*p_in*, *m_in*) and at the output (*p_out*, *m_out*). The following table summarizes the port definitions in the model interface:

Interface	Name	Property	Default Value
GENERIC	v_set	VOLTAGE	42.0
	i_in_max	CURRENT	30.0

Interface	Name	Property	Default Value
TERMINAL	p_in	ELECTRICAL	
	m_in	ELECTRICAL	
	p_out	ELECTRICAL	
	m_out	ELECTRICAL	

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY dcdc IS
 GENERIC(
 v_set: VOLTAGE := 42.0;
 i_in_max: CURRENT := 30.0);
 PORT(
 TERMINAL p_in, m_in, p_out, m_out : ELECTRICAL);
END ENTITY dcdc;
```

Architecture Description - DC-DC Converter

The *behav* architecture of the DC-DC converter uses a function to limit the current and voltage based on specified maximum and minimum values.

Condition	**Value**
$xmin + \varepsilon \leq x \leq xmax - \varepsilon$	x
$xmin + \varepsilon \geq x$	$xmin + \varepsilon^2/(xmin-x+2*\varepsilon)$
else	$xmax - \varepsilon^2/(x - xmax+2*\varepsilon)$

The model limits the voltage and current based on the following equations:

```
IF x_min + eps <= x AND x <= x_max - eps THEN RETURN x
ELSEIF x_min + eps >= x THEN RETURN x_min + eps * eps / (-x + x_min + 2.0 * eps)
ELSE x_max - eps * eps / (x - x_max + 2.0 * eps)
```

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE behav OF dcdc IS
 CONSTANT load_resistor: RESISTANCE := 0.1;
 CONSTANT tank_latency: REAL := 1.0;
 CONSTANT efficiency: REAL := 0.9;
 CONSTANT i_in_min: CURRENT := 0.0;
 CONSTANT v_tank_min : VOLTAGE := 10.0;
 CONSTANT p_ctrl : REAL := 1.0;
 CONSTANT i_ctrl : REAL := 1.0;
 CONSTANT v_tank_init: VOLTAGE := 12.0;
 QUANTITY v_in ACROSS i_in THROUGH p_in TO m_in;
 QUANTITY v_out ACROSS i_out THROUGH p_out TO m_out;
 QUANTITY v_tank, v_act_set, v_diff: VOLTAGE := 0.0;
 FUNCTION saturation( x, x_min, x_max: real) RETURN real IS
 CONSTANT eps: REAL := 1.0e-3;
 BEGIN
 IF x_min + eps <= x AND x <= x_max - eps THEN
 RETURN x;
 ELSIF x_min + eps >= x THEN
 RETURN x_min + eps * eps / (-x + x_min + 2.0 * eps);
 ELSE
 RETURN x_max - eps * eps / (x - x_max + 2.0 * eps);
 END IF;
 END FUNCTION;
BEGIN
 BREAK v_tank => v_tank_init;
 i_in == saturation((v_in - v_tank)/load_resistor, i_in_min, i_in_max);
```

```
v_tank'dot == (i_in + (i_out * v_out/v_tank)/efficiency)/tank_latency;
v_act_set == saturation(v_set, 0.0, v_tank * (v_set/v_tank_min));
v_diff == v_act_set - v_out;
i_out == -(p_ctrl * v_diff + i_ctrl * v_diff'integ);
END ARCHITECTURE behav;
```

This model architecture illustrates the use of branch and free quantities in a model description. The statement

```
QUANTITY v_in ACROSS i_in THROUGH p_in TO m_in;
```

associates two branch quantities, an across quantity *v_in* and a through quantity *i_in*, between the *p_in* and m_in terminals of the model. These branch quantities follow the energy conservation laws defined by nature of the terminals they are associated with. The statement

```
QUANTITY v_tank, v_act_set, v_diff: VOLTAGE := 0.0;
```

defines three free quantities not associated with any laws of conservation, of type *VOLTAGE*.

The solvability of a set of equations in a mixed signal model description should satisfy the following two rules:

- The number of equations specified in the model should be equal to the number of free quantities, through quantities, and port quantities of mode **OUT**.

  In this model description, the five equations specified account for three free quantities (*v_tank*, *v_act_set*, *v_diff*) and two branch quantities (*i_in*, *i_out*). There are no port quantities specified in the entity description.

- Any free quantities and any port quantities of mode **OUT** must appear in a simultaneous equation within the architecture of the model.

  In this model, *v_tank*, *v_act_set*, and *v_diff* all appear in simultaneous statements.

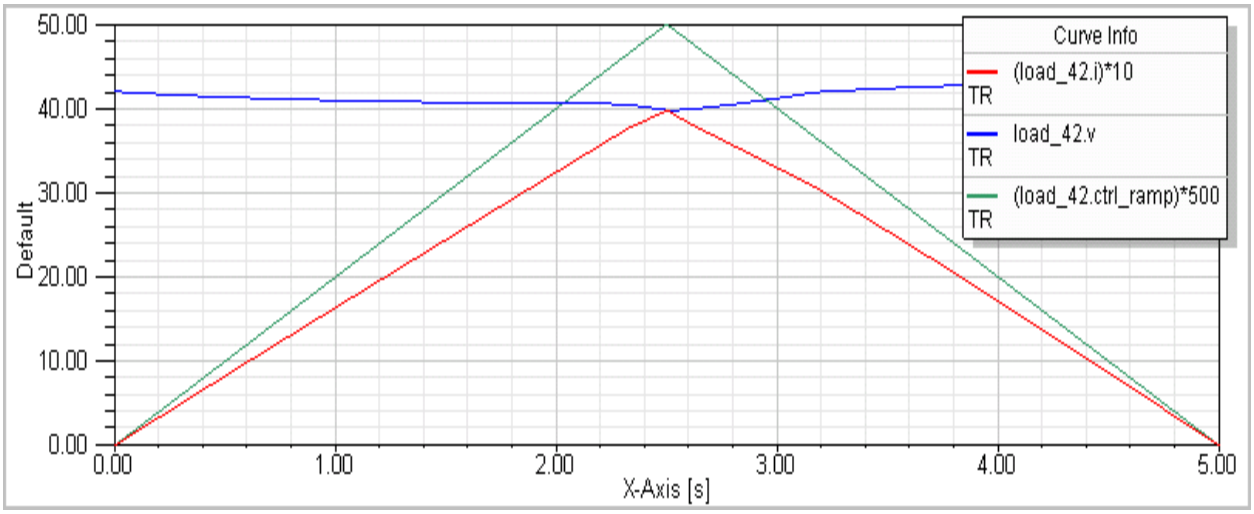
The equation for the output current models a PI controller and uses the '*INTEG* attribute to obtain the implicit quantity value of ∫v_diff.dt.

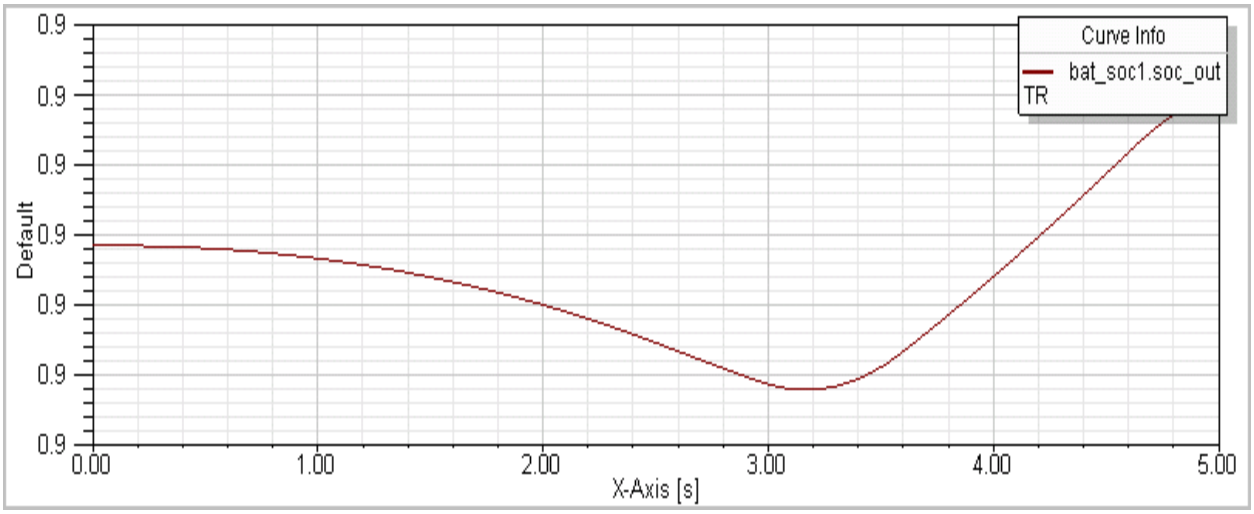
The Admittance load model is discussed with other load models in"Multilevel Modeling Techniques: Loads" on page 4-40 .

Results - DC-DC Converter

Select **Twin Builder** > **Analyze** to start the simulation. The simulation model is compiled and evaluated. The on-sheet plots show the simulation results.

Current and voltage of the Load model varies depending on the control signal and the resulting admittance (as the rate of change of the control signal).





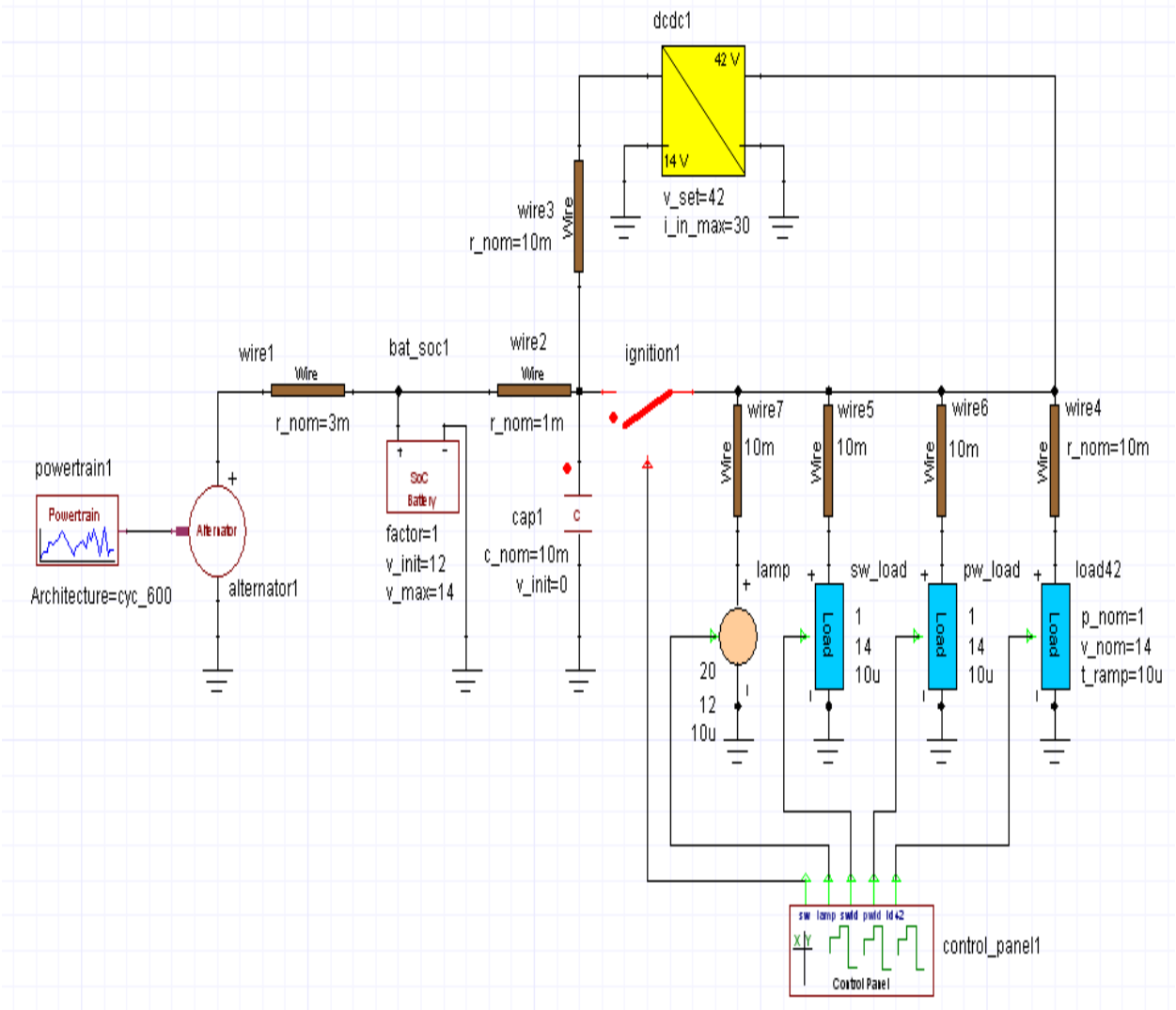
Step 4: Ignition Switch and Loads

This step, found in the file **system_ignition_loads.asmp**, extends the powernet system example with three 14V load models and a digital control panel. The control panel provides the

control inputs to each of the loads and controls the operation of the ignition switch that acts as a main control for the loads.

The following figure shows the extended *system_dcdc.asmp* sheet with the lamp and load models in the 14V subnet and the control panel.

The sheet can be simulated as shown with the full version of Twin Builder. Since the student version limits the size of a design, some models need to be excluded from the simulation.

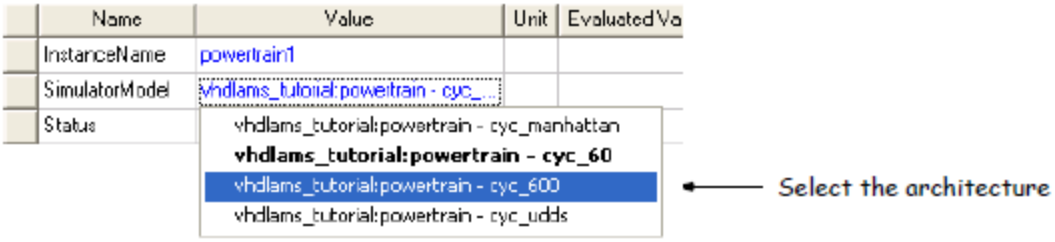


Defining the Ignition Switch and Load Simulation Model

1. Place and arrange all components shown in the figure above. (The schematic from the previous step can be extended.) *Alternator*, *Powertrain*, *Battery*, *Capacitor*, *DC-DC Converter*, *Ignition Switch*, *Load*, *Lamp, Control Panel*, and the *Wire model* are VHDL-

AMS models from the library on the **«Components»** tab. Type CTRL+G to place a ground node.

2. **Connect the models.** Place the cursor on a model pin to get the wire cursor and click to enter wire mode. Connect the components as indicated in the figure, setting the beginning, the corners, and the end of a wire with the mouse.
3. **Select the powertrain architecture.** Double-click the powertrain symbol to open the **Properties** dialog box and click the **«Parameter Values»** tab. Choose the **«cyc_600»** architecture from the list. This architecture provides a drive cycle waveform of a 600 second duration. Click **OK** to apply the changes.



4. **Define wire parameters.** Double-click each wire symbol to open its Properties dialog box. Click in the **«Value»** field, and enter the corresponding resistance value. Use the table in "Overview of Parameter Values - Ignition Switch and Loads Models" on the next page to set the correct value. Click **OK** to apply the changes.
5. **Define parameters of the capacitor.** Double-click the capacitor symbol to open the **Properties** dialog box. Click in the **«Value»** field, and enter *10m* as capacitance. Click **OK** to apply the changes.
6. **Define parameters of the loads.** Double-click each load symbol to open its Properties dialog box, and change the model name to the values shown in the diagram. Use the table in "Overview of Parameter Values - Ignition Switch and Loads Models" on the next page to set the correct architecture for each load. Click **OK** to apply the changes.
7. Place and arrange two Plots to display simulation results. Select **Draw** > **Report** > **Rectangular Plot**, and then click and drag to place and size the plot on the sheet. Similarly, select **Draw** > **Report** > **Digital Plot**, and then click and drag to place and size the plot on the sheet.
8. **Open the simulation Output list.** Choose **Twin Builder** > **Output Dialog** to open the **Output** dialog box.
9. **Select the simulation quantities**. Click the **"+"** next to the components to reveal the list of available quantities. Check the output boxes of *control_panel1.switch_out*, *control_panel1.lamp_out*, *control_panel1.power_out*, *control_panel1.switched_out*, and *control_panel1.admittance_out*. Click **OK** to apply the changes.
10. **Add traces to the plot.** Double-click the on-sheet plot to open the **Report** dialog box. Select each of the output quantities defined above and change the scaling used for the

traces as shown in "Results - Ignition Switch Model" on page 2-33 . Click **Add Trace** to add it to the plot. Click **Close** to apply the changes.

11. Repeat the steps 8, 9,and 10 for the second plot but check the box of *bat_soc1.soc_out* as the output.
12. **Define the simulation parameters.** Choose **Twin Builder>Add Solution Setup>Transient** to open the Transient Analysis Setup dialog box, and set the value for simulation **End Time** to *400* seconds, **Min Time Step** to *10 us*, and **Max Time Step** to *5 s*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Adaptive Trapezoid-Euler**, **Maximum current error** to *0.01* and **Maximum voltage error** to *0.01*. Click **OK** to apply the changes.
13. **Save the sheet**. Choose **File>Save As**, enter file name and directory, and click **OK**.

> **Note:**
>
> The Digital Graph is especially designed to display digital signals of VHDL-AMS models. In contrast to the Rectangular Plot, each signal has its own coordinate system (Y-axis).

Overview of Parameter Values - Ignition Switch and Loads Models

Model Name	Parameter Values	Library	Architecture
powertrain1	no parameters	vhdlams_tutorial	cyc_600
alternator1	no parameters	vhdlams_tutorial	behav
bat_soc1	factor=1; v_init[V]=12; v_max[V]=14	vhdlams_tutorial	behav
wire1	r_nom [Ω]=3m	vhdlams_tutorial	behav
wire2	r_nom [Ω]=1m	vhdlams_tutorial	behav
wire3	r_nom [Ω]=10m	vhdlams_tutorial	behav
wire4/5/6/7	r_nom [Ω]=10m	vhdlams_tutorial	behav
cap1	c_nom[F]=10m; v_init[V]=0	vhdlams_tutorial	behav

Model Name	Parameter Values	Library	Architecture
dcdc1	v_set[V]=42; i_in_max[A]=30	vhdlams_tutorial	behav
ignition1	*on_ctrl=control_panel.switch_out*	vhdlams_tutorial	behav
load_42	p_nom[W]=1; v_nom[V]=14; t_ramp[s]=10u; *on_ctrl=OmniCaster1.val*	vhdlams_tutorial	admittance
lamp	p_nom[W]=20; v_nom[V]=12; t_ramp[s]=10u; *on_ctrl=control_panel1.lamp_out*	vhdlams_tutorial	behav
sw_load	p_nom[W]=1; v_nom[V]=14; t_ramp[s]=10u; *on_ctrl=control_panel.switched_out*	vhdlams_tutorial	switch
nom_load	p_nom[W]=1; v_nom[V]=14; t_ramp[s]=10u; *on_ctrl=control_panel.power_out*	vhdlams_tutorial	nominal
control_panel1	no parameters	vhdlams_tutorial	all_loads

Ignition Switch Model

The ignition switch model is a simple non-ideal switch that accepts the on-resistance and off-conductance of the switch as parameters. A digital Boolean signal with *TRUE/FALSE* values is used as a control signal to turn the switch on and off.

The Load model uses the electrical (ELECTRICAL) domain; consequently, the *electrical_systems* package that is defined in the IEEE library must be included. The following statements in the model definition include the package:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
```

Entity Description - Ignition Switch

This model defines two parameter values for *r_on* and *g_off*, terminals *p* and *m*, and a digital control signal *on_ctrl* of type *BOOLEAN*. The following table summarizes the port definitions in the model interface:

Interface	Name	Property	Default Value
GENERIC	r_on	*RESISTANCE*	1.0e-3
	g_off	*REAL*	1.0e-9
TERMINAL	p	*ELECTRICAL*	
	m	*ELECTRICAL*	
SIGNAL	on_ctrl	*BOOLEAN*	*FALSE*

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY ignition IS
 GENERIC (
 r_on: RESISTANCE := 1.0e-3;
 g_off: REAL := 1.0e-9);
 PORT (
 TERMINAL p,m: ELECTRICAL;
 SIGNAL on_ctrl : IN BOOLEAN);
END ENTITY ignition;
```

Architecture Description - Ignition Switch

The ignition switch model uses a simultaneous **IF** condition to turn the switch on/off based on the input Boolean control signal, *on_ctrl*. The **BREAK** statement is used to ensure that the analog solver is able to resolve discontinuities that may arise because of the sudden transition of the digital control signal.

The equivalent VHDL-AMS description for defining the model architecture is as follows:

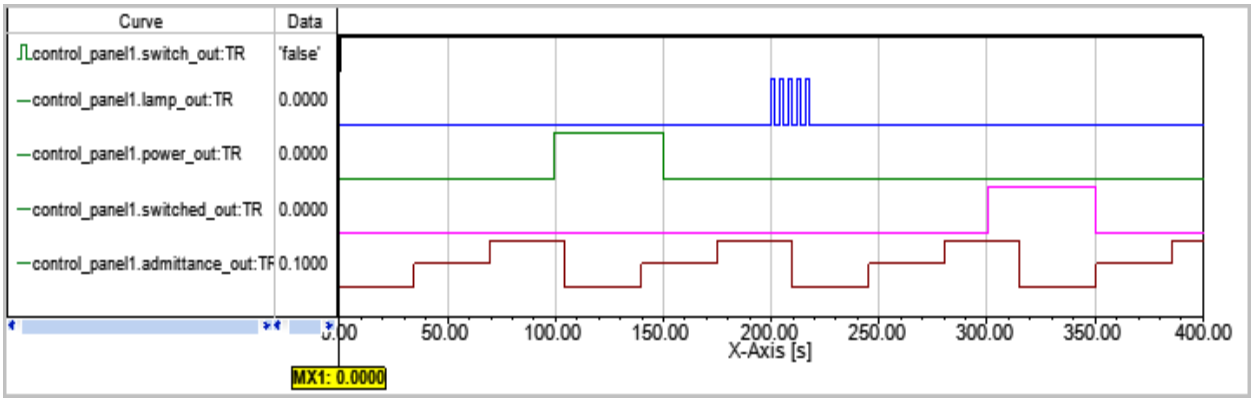
```
ARCHITECTURE behav OF ignition IS
 QUANTITY v ACROSS i THROUGH p TO m;
BEGIN
 IF on_ctrl USE
 v == i * r_on;
```

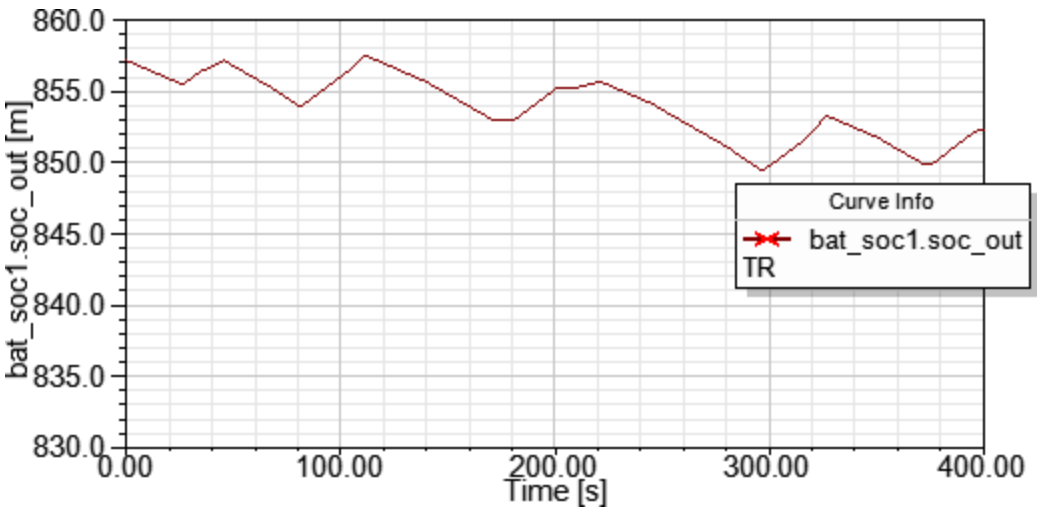
```
ELSE
i == v * g_off;
END USE;
BREAK ON on_ctrl;
END ARCHITECTURE behav;
```

Results - Ignition Switch Model

Select **Twin Builder** > **Analyze** to start the simulation. The simulation model is compiled and evaluated. The on-sheet plots show the simulation results on the sheet.

Depending on the switching characteristics defined in the control panel the loads are switched on and off, causing the battery State of Charge to vary.





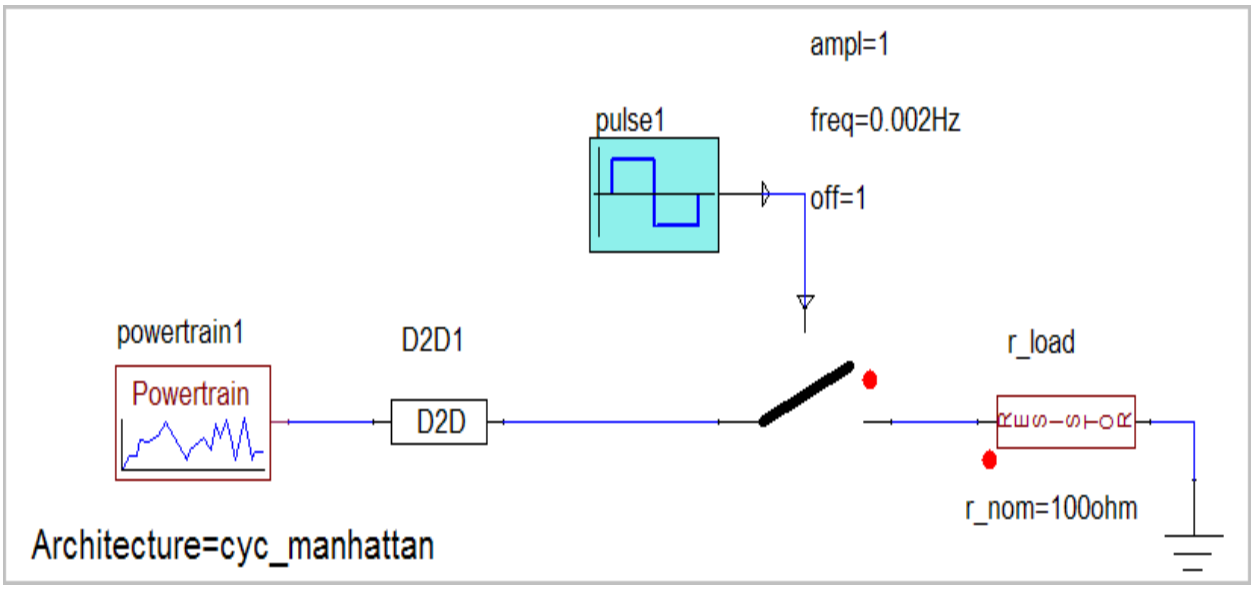
Step 5: Powertrain

This example, found in the file **system_powertrain.aedt**, uses an architecture that models component behavior using a lookup table. The powertrain model reads the lookup table from a file and uses the values for simulation.

The powertrain model provides the speed profile of an engine through a mechanical pin output. This mechanical pin is connected as an input to a Domain-to-Domain (D2D) model that transfers the value of the *ROTATIONAL_V* pin to the electrical domain. The *cyc_udds* architecture is used for the powertrain model to provide a drive cycle.

Some non-electrical parts of a multidomain simulation model can be modeled in the electrical domain. To interface the mechanical domain powertrain model with the electrical switch model, use the Domain-to-Domain model (*D2D*) from the *Nature Transformations* folder on the **«Tools»** tab.

The following figure shows the *system_powertrain.aedt* schematic sheet:



Creating the Powertrain Simulation Model

1. **Place and arrange all components shown in the figure above.***Powertrain* and *Resistor* are VHDL-AMS models from the **Basic Elements VHDLAMS** library in the **Component Libraries** window. The D2D model is a model from the *Nature Transformations* folder in the **Tools** > **Transformations** library folder. (This model does not need to be placed manually: it will be automatically inserted when the powertrain model is connected to the switch model.) The *Ideal Switch in VHDL-AMS* is a model from the *Switches* folder under the *Circuit* folder, and the *Pulse Wave Time Function in VHDL-AMS* a model from the *Time Functions* folder under the *Tools* folder, in the *Basic Elements*

VHDLAMS library in the **Component Libraries** window. Type Ctrl+G to place a ground node.

2. **Connect the models.** Place the cursor on a model pin to get the wire cursor and click to enter wire mode. Connect the components as indicated in the figure, setting the beginning, the corners, and the end of a wire with the mouse. Press Esc to end wire mode.
3. **Define the powertrain parameters.** Double-click the powertrain symbol to open the **Properties** dialog box, and click the **Parameter Values** tab. Choose the **cyc_manhattan** architecture from the **SimulatorModel** Value list. This architecture provides a drive cycle representation of a 1000 second duration. Click **OK** to apply the changes.
4. **Define the pulse parameters.** Double-click the pulse symbol to open the Properties dialog box. Click in the **Value** field, and enter *1* for the amplitude value, *2m* for the frequency value, and *1* for the off value. Click **OK** to apply the changes.
5. **Define the resistor parameters.** Double-click the resistor symbol to open the **Properties** dialog box. Change the model name from *res1* to *r_load*. Click in the **Value** field, and enter *100* for the resistance value. Click **OK** to apply the changes.
6. **Place and arrange two Rectangular Plots to display simulation results.** Select **Draw** > **Report** > **Rectangular Plot**, and then click and drag to place and size the plots on the sheet.
7. **Open the simulation Output list.** Choose **Twin Builder** > **Output Dialog** to open the **Output** dialog box.
8. **Select the simulation quantities**. Click the **"+"** next to the components to reveal the list of available quantities. Check the boxes of output *r_load.v* and input *switch.ctrl*. Click **OK** to apply the changes.
9. **Add traces to the plot.** Double-click the on-sheet plot to open the **Report** dialog box. Select each of the output quantities defined above and change the scaling used for the traces as shown in "Results - Powertrain Model" on page 2-39 . Click **Add Trace** to add it to the plot. Click **Close** to apply the changes.
10. Repeat steps 7, 8, and 9 for the second plot but check the box of *powertrain1.vel* as the output.
11. **Define the simulation parameters.** Choose **Twin Builder** > **Add Solution Setup** > **Transient** to open the **Transient Analysis Setup** dialog box, and set the value for simulation **End Time** to *1000* seconds, **Min Time Step** to *1 ms*, and **Max Time Step** to *10 ms*. Click **OK** to apply the changes. Right-click the Analysis icon in the Project Manager and select **Add Solution Options**. On the **TR** tab set **Integration formula** to **Adaptive Trapezoid-Euler**. Click **OK** to apply the changes.
12. **Save the sheet**. Choose **File** > **Save As**, enter file name and directory, and click **OK**.

Overview of Powertrain Model Parameter Values

The following table summarizes the parameter values for the models on the sheet:

Model Name	Parameter Values	Library	Architecture
powertrain1	no parameters	vhdlams_tutorial	cyc_ manhattan
D2D1	no parameters	Tools/Transformations/Nature Transformations	SML model
pulse1	ampl=1 ; freq[Hz]=2m ; off=1; tdelay]s]=0	Basic Elements VHDMLAMS/Tools/Time Functions	behav
switch1	*ctrl=pulse.val*	AMS/basic_vhdlams.smd	behav
r_load	r_nom [Ω]=100	Projects/vhdlams_tutorial.smd	behav

Powertrain Model

The powertrain model uses the mechanical (ROTATIONAL_V) domain and file I/O functions; consequently, the *mechanical_systems* and *textio* package that are defined in the IEEE and STD libraries must be used. The following statements in the model definition include the packages:

```
LIBRARY IEEE, STD;
USE IEEE.MECHANICAL_SYSTEMS.ALL;
USE STD.TEXTIO.ALL;
```

Entity Description - Powertrain Model

The model provides the characteristic values through a ROTATIONAL_V terminal as indicated in the following table:

Interface	Name	Property
TERMINAL	n_out	*ROTATIONAL_V*

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY powertrain IS
 PORT (
 TERMINAL n_out : ROTATIONAL_V);
```

```
END ENTITY powertrain;
```

Architecture Description - Powertrain Model

The cyc_*manhattan* architecture of the powertrain model obtains the speed profile information from the *cyc_mph_manhattan_tab.txt* file. This file is stored in the project folder.

The following table lists sample values stored in this file:

Time [s] *1k	0.013	0.014	0.015	0.016	0.017	0.018
Speed [mph] *1k	0.0028	0.0049	0.0049	0.0117	0.0134	0.0149

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE cyc_manhattan OF powertrain IS
 CONSTANT unit_time : TIME := 1000 sec;
 FILE cyc : TEXT OPEN READ_MODE IS "cyc_mph_manhattan_tab.txt";
 SIGNAL val : REAL := 0.0;
 SIGNAL ped : TIME := 1 ms;
 QUANTITY vel ACROSS tor THROUGH n_out TO ROTATIONAL_V_REF;
BEGIN
 PROCESS
 VARIABLE buf : LINE;
 VARIABLE t_val_old, t_val_new, t_val, s_val : REAL := 0.0;
 BEGIN
 WHILE NOT ENDFILE(cyc) LOOP
 READLINE(cyc,buf);
 READ (buf,t_val_new);
 READ (buf,s_val);
 t_val := t_val_new - t_val_old;
 t_val_old := t_val_new;
```

```
ped <= unit_time * t_val;
val <= s_val*1.0e3;
WAIT FOR ped;
END LOOP;
WAIT;
END PROCESS;
vel == 8.3776*val'RAMP(1.0,1.0); --vel in rad/sec assuming 2000rpm=>25mph
END ARCHITECTURE cyc_manhattan;
```

A file object *cyc* of predefined file type *TEXT* is declared using the following statement:

```
FILE cyc : TEXT OPEN READ_MODE IS "cyc_mph_manhattan_tab.txt";
```

The **WHILE** loop reads the file from beginning to end, using the **ENDFILE** function to test if the file has been read completely. It uses the *READLINE* procedure to read one line at a time into the *buf* variable of predefined type LINE. The two values of time and speed from the *buf* line variable are read into the variables *t_val* and *s_val* using the *READ* procedures. The speed value *vel* is applied as the across quantity to the output conservative pin, *n_out*, after converting the value to rotational domain (vel=*2π/60). The value *vel* is calculated assuming that 2000 rpm is equivalent to 25 mph.

The digital **SIGNAL** value *val* is transformed to the analog **QUANTITY***vel* with the '*RAMP* attribute. The rise time and fall times for the '*RAMP* attribute are 1 second each.

To specify data from a different file, for example *cyc_mph_udds_tab.txt*, a new architecture may be created, as follows:

```
ARCHITECTURE udds OF powertrain IS
 CONSTANT unit_time : TIME := 1000 sec;
 FILE cyc : TEXT OPEN READ_MODE IS "cyc_mph_udds_tab.txt";
 SIGNAL val : REAL := 0.0;
 SIGNAL ped : TIME := 1 ms;
 QUANTITY vel ACROSS tor THROUGH n_out TO ROTATIONAL_V_REF;
BEGIN
```

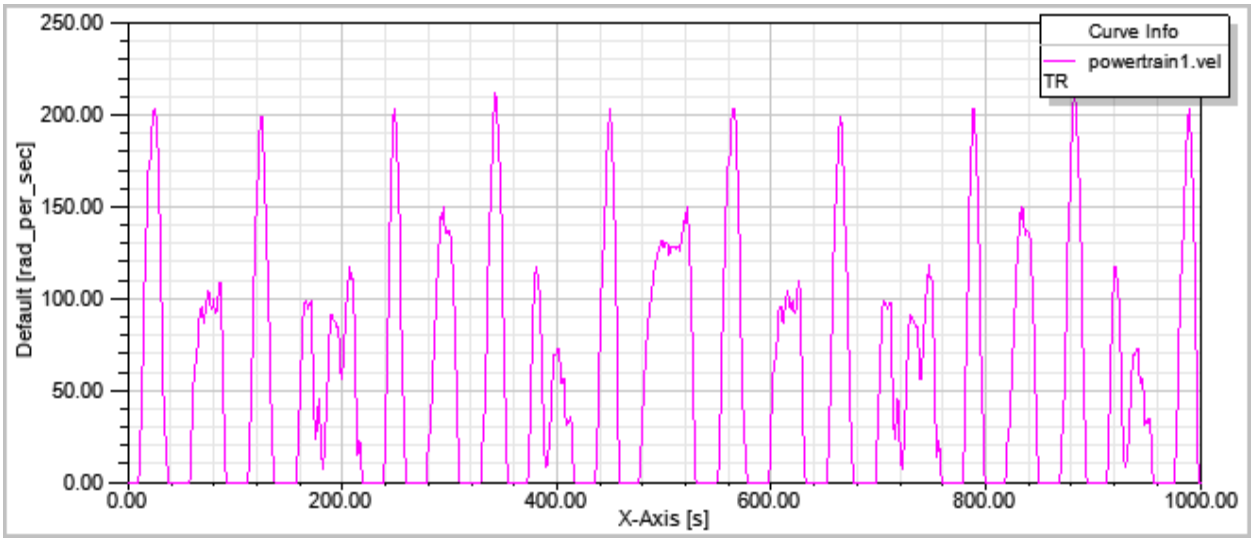
```
PROCESS
VARIABLE buf : LINE;
VARIABLE t_val_old, t_val_new, t_val, s_val : REAL := 0.0;
BEGIN
WHILE NOT ENDFILE(cyc) LOOP
READLINE(cyc,buf);
READ (buf,t_val_new);
READ (buf,s_val);
t_val := t_val_new - t_val_old;
t_val_old := t_val_new;
ped <= unit_time * t_val;
val <= s_val*1.0e3;
WAIT FOR ped;
END LOOP;
WAIT;
END PROCESS;
vel == 8.3776*val'RAMP(1.0,1.0); --vel in rad/sec assuming 2000rpm=>25mph
END ARCHITECTURE udds;
```

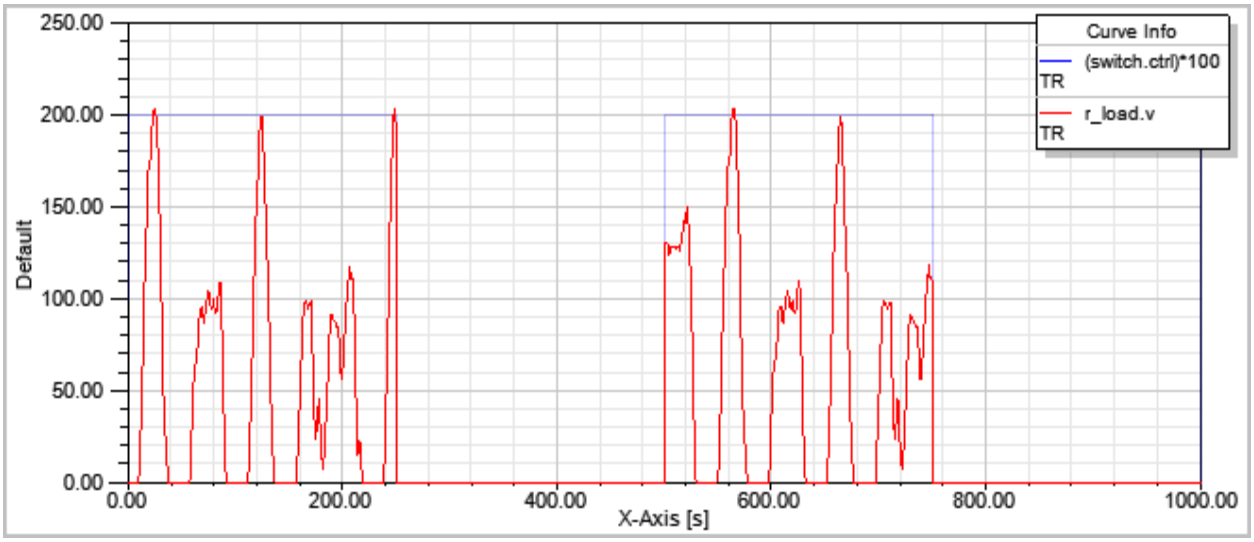
The only difference between the two architectures is the file name specified. However, the multiple architectures allow the sheet to be simulated using different speed profiles by simply changing the architecture, enabling multiple tests to be quickly performed on the powernet system.

Results - Powertrain Model

Select **Twin Builder** > **Analyze** to start the simulation. The simulation model is compiled and evaluated. The on-sheet plots show the simulation results on the sheet.

The powertrain model provides the values for simulation based on the values in the characteristic file. Whenever the electrical switch is turned on, the speed output from the powertrain model is transferred to the load resistor.



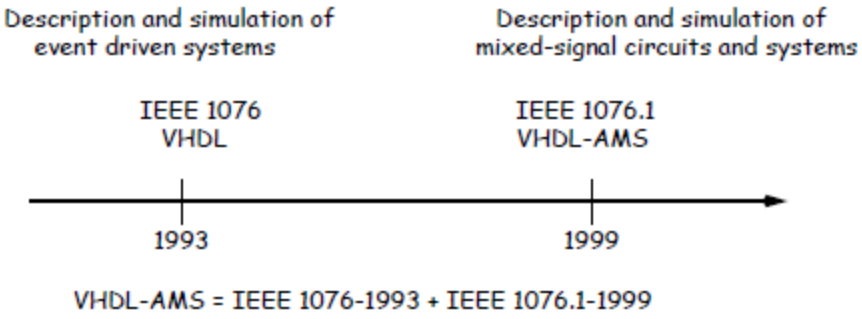


3 - VHDL-AMS Models in Twin Builder

VHDL-AMS (**V**ery high-speed integrated circuit **H**ardware **D**escription **L**anguage – **A**nalog **M**ixed **S**ignal) is a standardized language used for describing digital, analog, and mixed-signal systems.

The Institute of Electrical and Electronics Engineers (IEEE) standardized the VHDL-1076 language as a Hardware Description Language (HDL) for digital models. The VHDL standard from 1993 was extended in 1999 for the description of analog and mixed-signal models in the form of the IEEE 1076.1 standard for VHDL-AMS.

See ***VHDL-AMS Language Fundamentals*** and information on the Web site of IEEE 1076.1 Working Group at http://www.eda.org/vhdl-ams.



Twin Builder supports the development and simulation of models that have been developed in VHDL-AMS:

- VHDL-AMS Wizard can easily and quickly create simple and complex models without the hassle of typing code and correcting syntax.
- VHDL-AMS models in ASCII text can easily be imported into a library or a subsheet on the Schematic.
- VHDL-AMS models in libraries as well as text or graphical subsheets can be easily exported to ASCII text files.
- Schematics containing VHDL-AMS models can be exported as netlists to VHDL-AMS ASCII files.
- Stimulus generator can be used to create digital stimulus in VHDL and supports a variety of patterns and data types.
- VHDL-AMS models can instantiate Twin Builder models as foreign models - this helpful feature allows users to take advantage of the large number and variety of highly optimized and fast Twin Builder models.

This chapter contains information on:

- VHDL-AMS models
- Load reference arrow system of physical domains
- Packages and libraries
- Entities and architectures
- Schematic environment
- Model output definitions
- 2D Digital Graph

Using VHDL-AMS Models

Twin Builder provides several libraries containing VHDL-AMS components developed according to IEEE 1076.1 (VHDL Analog and Mixed Signal Extensions Standard) and IEEE 1076 (VHDL standard):

- **Basic Elements VHDLAMS**: contains common basic circuit components and blocks.
- **Digital Elements**: contains common basic components used for simple digital circuits.
- **Tools** > **Transformations** > **OmniCasters**: contains auxiliary components (called OmniCasters) that facilitate the connection of different data types and natures.

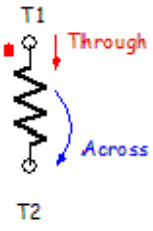
The models provided are open and can be used to derive more advanced models by copying the description to a user-defined library and editing the text to modify the model. The files can be used and distributed provided the copyright statement, included in each model description, is not removed. To access the model description, right-click the component name in the **Component Libraries** window and select **Edit Component** on the context menu to open the **Edit Component** dialog box. On the **«Simulation Models»** tab, click **Edit** in the **Edit Model** field to view (and edit) the text.

The functionality of all VHDL-AMS models is a subset of that of the equivalent Twin Builder models available in the **Basic Elements** library. VHDL-AMS models can be used in parallel with Twin Builder models.

The digital models operate with digital signals and can be characterized with rise time/fall time/propagation delays. They do not have any conservative nodes but can be connected with analog quantities using *OmniCasters* (in the **Tools** library).

Across and Through Quantities of Natures

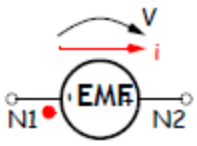
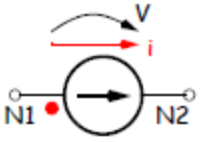
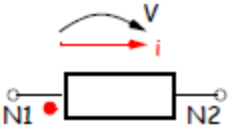
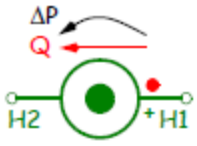
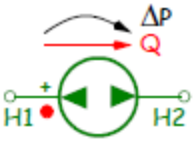
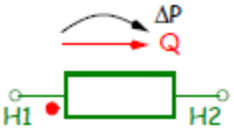
VHDL-AMS models can support *nature types* for several physical domains. *Nature types* are properties of *conservative nodes* (also referred to as *ports* or *terminals*) of models. At least one specific nature exists for each domain. An *across* and a *through* quantity is associated with each nature. The following table links the *across* and *through* quantities for each nature type:

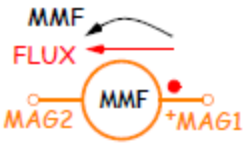
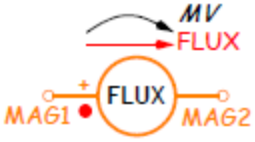
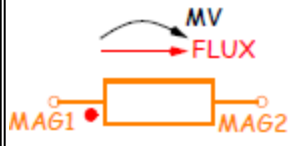
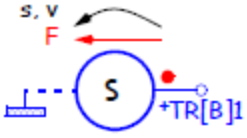
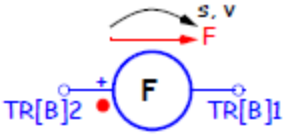
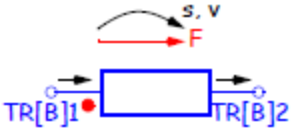
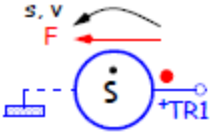
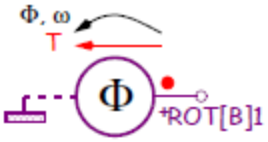
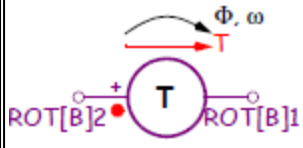
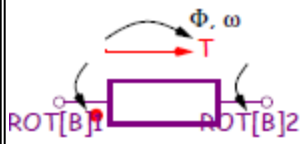
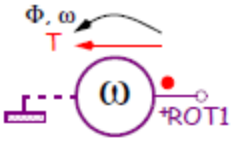
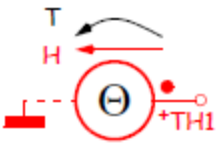
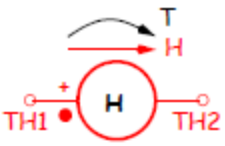
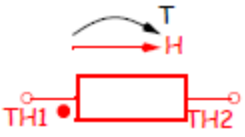
Nature	Across	Through	Circuit
Electrical	Voltage [V]	Current [A]	
Fluidic	Pressure [Pa]	Flow Rate [m³/s]	
Magnetic	Magneto Motive Force [A]	Magnetic Flux [Vs]	
Translational	Displacement [m]	Force [N]	
Translational_v	Velocity [m/s]	Force [N]	
Rotational	Angle [rad]	Torque [Nm]	
Rotational_v	Angular Velocity [rad/s]	Torque [Nm]	
Thermal	Temperature [K]	Heat Flow [J/s]	

Load Reference Arrow System

The following table shows the Twin Builder load reference arrow system for all available domains. This system defines how *across* and *through* quantities are measured for models in each domain. The measuring direction is marked by the red dot on the model symbol. The red dot is always at pin 1 of a model.

- *Across* quantities: Value is calculated by subtracting the value at Pin2 from the value at Pin1
- *Through* quantities: Value is positive if the quantity flows into the model at the pin marked with the red dot

	Difference Sources	Flow Sources	Passive Components
Electrical			
Fluidic			

	Difference Sources	Flow Sources	Passive Components
Magnetic			
Mechanic Translational			
			
Mechanic Rotational			
			
Thermal			

Packages and Models in Libraries

Twin Builder model libraries (**.asmd** files) can also serve as VHDL-AMS model libraries. VHDL-AMS packages are provided as **.apkg** files. Additionally, component definitions are contained in **.aclb** files, and symbols for components and models reside in **.aslb** files.

The following symbols are used to distinguish between the different types of Twin Builder library elements:

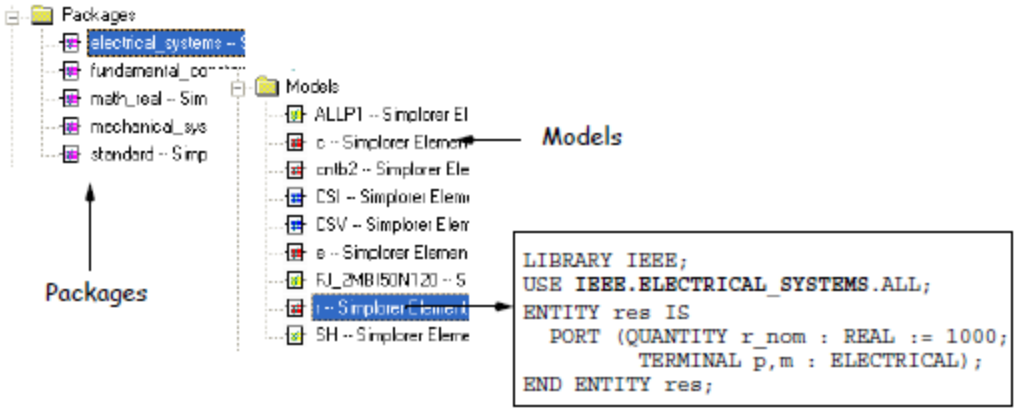
	Twin Builder library
	SML model
	VHDL-AMS model
	C model
	VHDL-AMS package
	Symbol library element
	C-model defined in a separate DLL
	Internally implemented model

Packages are collections of re-usable declarations and definitions such as types, constants, functions, procedures, and natures. Standardized packages from IEEE (such as *math_real* and *textio*) and proposed packages from IEEE (such as *electrical_systems* and *thermal_systems*) are available in the *ieee* and *std* libraries on the **«AMS»** tab.

> **Note:**
>
> Model and package symbols that have a blue padlock on them indicate that the models or packages are locked and cannot be edited.

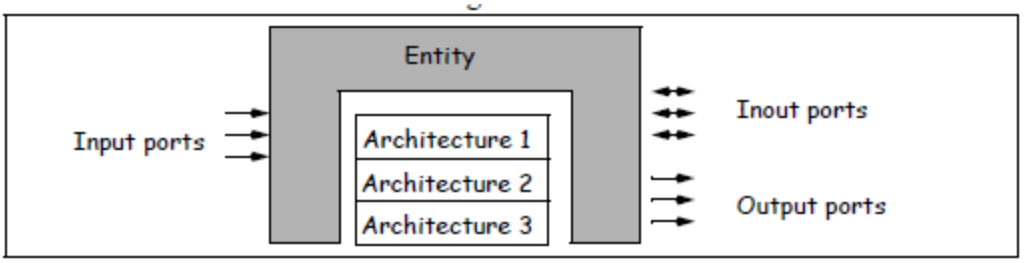
To use declarations from a package, the corresponding package must be included in the model description, and the package must be available in an installed model library within the current Twin Builder configuration. In the following example the Resistor model uses the ELECTRICAL nature for its conservative pins. Consequently, the ELECTRICAL_SYSTEMS package from the

IEEE library needs to be included in the model description. User-defined packages can be added to a model library and used in model descriptions.



Entities and Architectures of VHDL-AMS Models

VHDL-AMS models consist of two parts: an *entity* declaration and one or more *architecture* descriptions. The entity describes the interface of the model and declares inputs, outputs, constant value parameters, conservative pins, and so on. The architecture defines the behavior of the model, and several modeling styles may be used for this description such as behavioral, dataflow, structural. It is possible to associate multiple architectures with an entity declaration, and only the selected architecture will be used during simulation.



The model description for a passive resistor model used in the tutorial examples follows. Explanations of the statements used in the model description are included. Open the **system_dcdc.aedt** project file as described in "Entities and Architectures" on page 6-2 to view the model.

VHDL-AMS Resistor Model Description

To view the model description, select **Tools>Edit Libraries>Models**. Choose the **[sys] Simplorer Elements\Basic Elements VHDLAMS\Basic Elements VHDLAMS** library

from the **Libraries** list, and click to select the "**r**" model from the list in the **Edit Libraries** dialog box. The model description appears in the **Model text** pane to the right of the list. The example resistor model has an interface with two electrical terminals and one non-conservative input for the static resistance value.

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;

ENTITY R IS
 PORT (QUANTITY R : RESISTANCE := 1.0e+3;
 TERMINAL p,m : ELECTRICAL);
END ENTITY R;

ARCHITECTURE behav OF R IS
 QUANTITY v ACROSS i THROUGH p TO m;
BEGIN
 v == i*R;
END ARCHITECTURE behav;
```

ENTITY	Interface description of the model *r*.
PORT	Conservative and non-conservative pins of a circuit. Here, there are two electrical **TERMINAL**s that represent the plus (+) and minus (–) pins of the model, and one non-conservative **QUANTITY** input for the resistance value of the model.
TERMINAL	Conservative pin associated with a domain. Here, there are two conservative pins *p* and *m* declared for the **ELECTRICAL** domain.
ARCHITECTURE	Defines the behavior *behav* of the model *r*. **QUANTITY** voltage **ACROSS** current **THROUGH** p **TO** m; This statement defines voltage as an **ACROSS** quantity and current as a **THROUGH** quantity between the pins *p* and *m* for the model. v == i * R; This statement specifies the model behavior v = i*R in VHDL-AMS equation form.

Capacitor

The capacitor model has an interface with two electrical terminals and two non-conservative inputs for the static capacitance value and initial voltage value.

To view the model description, select **Tools** > **Edit Libraries** > **Models**. Choose the **[sys] Simplorer Elements\Basic Elements VHDLAMS\Basic Elements VHDLAMS** library from the **Libraries** list, and click to select the "**c**" model from the list in the **Edit Libraries** dialog box. The model description appears in the **Model text** pane to the right of the list.

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;


ENTITY cap IS
GENERIC (
c_nom: CAPACITANCE := 1.0;
v_init: VOLTAGE := 0.0);
PORT (TERMINAL p,m : ELECTRICAL);
END ENTITY cap;


ARCHITECTURE behav OF cap IS
QUANTITY v ACROSS i THROUGH p TO m;
BEGIN
BREAK v => v_init;
i == c_nom*v'DOT;
END ARCHITECTURE behav;
```

ENTITY	Interface description of the model *cap*.
PORT	Conservative and non-conservative pins of a circuit. Here, there are two electrical terminals that represent the plus (+) and minus (–) pins of the model and two non-conservative inputs for the capacitance and initial voltage value of the model.
GENERIC	Static value represented as a non-conservative pin, defined at simulation time t=0 that remains unchanged during simulation. Here, *c_nom* and *v_*

	init are static values. The data type associated with the capacitance and the initial value is of type *REAL* and the default values are '1' farad and '0' volts.
TERMINAL	Conservative pin associated with a domain. Here, there are two conservative pins *p* and *m* declared for the electrical domain. Other common domains that are used include *TRANSLATIONAL*, *ROTATIONAL*, *THERMAL*, *FLUIDIC*, and so on.
ARCHITECTURE	Defines the behavior *behav* of the model *cap*. **QUANTITY** voltage **ACROSS** current **THROUGH** p **TO** m; This statement defines voltage as an across quantity and current as a through quantity between the pins *p* and *m* for the model. **BREAK** v => v_init; Here, the **BREAK** statement is used to initialize the voltage of the capacitor if a discontinuity is detected by the simulator. i == c_nom * v'DOT; This statement specifies the model behavior i = C*(dv/dt) in VHDL-AMS equation form. '*DOT* is a pre-defined attribute used to obtain the differential of a quantity (in this case, *voltage*) with respect to time.

Placing and Connecting Models

The **Component Libraries** dialog box provides VHDL-AMS models in the **Basic Elements VHDLAMS**, **Digital Elements**, and **Tools>Transformations>OmniCasters** libraries. To place a component, select a library in the **Component Libraries** dialog box and then a component in the tree. Drag the component onto the sheet. See also ***Overview of the Twin Builder Interface***.

> **Note:**
>
> The **Search** tab allows quick access to all models.

Wire mode allows connections to be made between a number of components. To enter wire mode, choose **Draw** > **Wire**, or type Ctrl+W. Wire mode can also be activated by placing the cursor over a connection point (it changes to the wire cursor) and clicking. Place the cursor over a second point and click to make a connection to that point. Continue to click on the beginning and end points to make connections. Clicking while not on a connection point allows corners to be set to change the direction of the wire. To exit wire mode, press ESC.

Components can also be connected by overlapping their pins, or can be connected to a wire by placing a component's pin over the wire. Unconnected wires are shown as broken red lines.

Using Transformation Models

Complex simulation models usually need simple auxiliary models to connect different data types and natures simply and quickly, so that the real subject of simulation can be investigated easily. VHDL-AMS models need transformation models that perform data type conversions; VHDL-AMS models from different domains need transformation models that connect conservative nodes of different natures.

OmniCasters are interface models that are used to interface analog quantities with digital signals or connect digital signals of different data types.

The flexible OmniCaster model, which performs conversions depending on the connected data types, and the fixed OmniCaster models with predefined data types are available for use. The flexible OmniCaster model uses a fixed OmniCaster model based on the data types of the pins connected to it. The fixed OmniCaster models have an **ENTITY** description according to the nature of the conversion and an **ARCHITECTURE** description that may use function calls to perform the conversion.

The different data types considered for signals are **REAL**, **INTEGER**, **BIT**, **BOOLEAN**, **BIT_VECTOR**, **STD_LOGIC**, and **STD_LOGIC_VECTOR**. The only data type considered for analog quantities is **REAL**. Some transformations can be specified with propagation delay, rise time, fall time, threshold, and output value parameters. The functions used for the type conversions are available in the **omnicaster_package**.

Connect conservative nodes of different domains using a Domain-to-Domain (**D2D**) model available in the **Nature Transformations** folder of the **Tools** > **Transformations** library. It is also possible to connect a conservative node to a non-conservative node using a **C2NC** connection model. In this case, the across value from the conservative node is transferred to the non-conservative node.

Unlike the OmniCasters described in VHDL-AMS, the D2D and C2NC are not VHDL-AMS models. Consequently, if schematics that use D2D or C2NC models are exported to an ASCII netlist, the exported description may not simulate in a third party VHDL-AMS simulator.

See "Step 5: Powertrain" on page 2-34 .

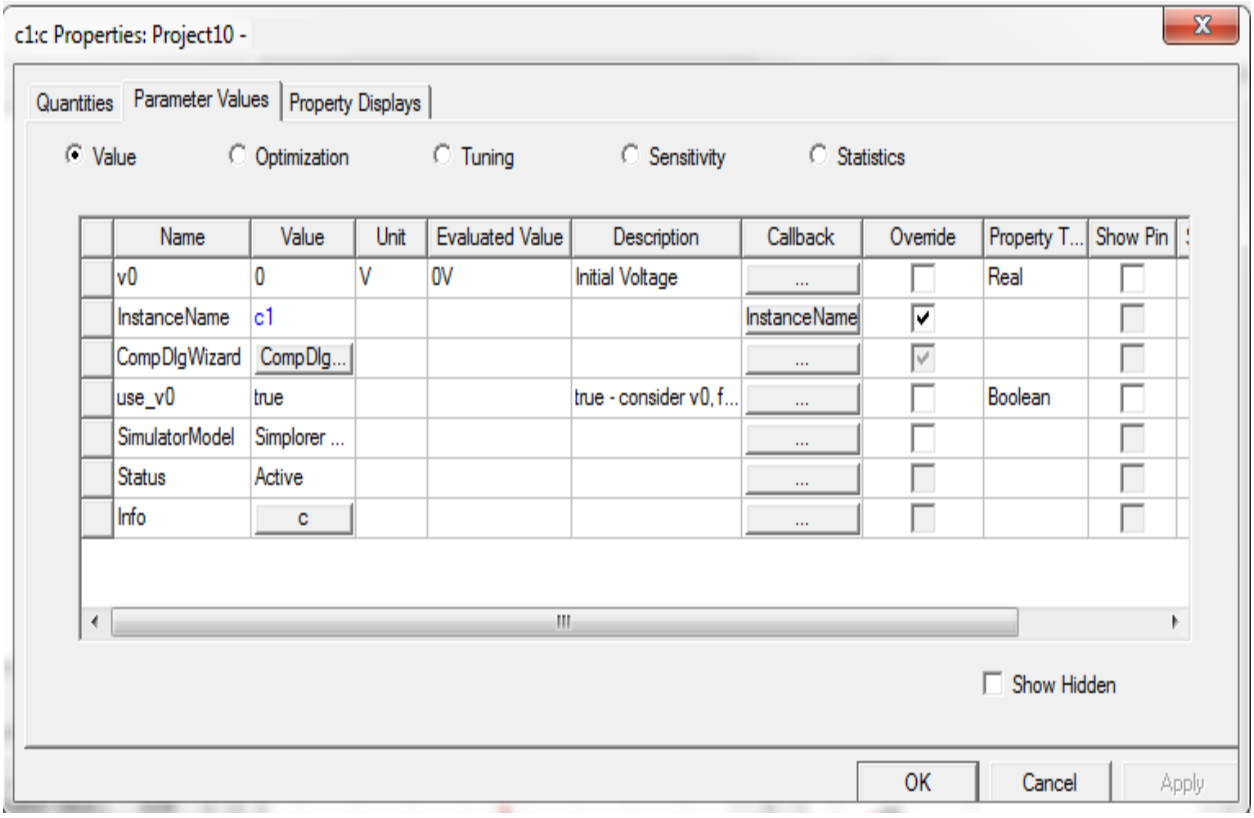
Defining Model Properties

Every VHDL-AMS model placed on the sheet has a **Properties** dialog box where its parameters can be modified and where its symbol and display properties can be selected. Unlike Twin Builder models, VHDL-AMS models do not use component **Parameter** dialog boxes for parameterization of the components. All parameters and display properties are simply listed and changed in the **Properties** dialog box tabs.

To open the **Properties** dialog box, do one of the following:

- Double-click the component.
- Right-click on the component, and choose **Properties**.
- Choose **Edit>Properties** for the selected component.

The number of parameters and quantities in a **Properties** dialog box varies depending on the model selected. The following example shows the dialog box displayed when you right-click on this component and choose **Properties**. To enter a parameter value, click in the **Value** input field of the corresponding parameter in the **Parameter Values** or **Quantities** tab and enter a numerical value (with or without suffix), a variable, or an expression. You can also choose the desired unit of measure in the **Unit** field. Default values are used if no other value is defined for the parameter. The **Properties Display** tab allows you to control the display of a components properties..



Note:

If you have selected the **Show advanced property data** check box on the **General** tab at **Tools** > **Options** > **Schematic Editor Options**, more tabs appear. For this component, these additional tabs are:

- The **General** tab, which shows information about the model's source library and the model type identification. These settings cannot be changed.
- The **Symbol** tab, which allows you to control various symbol properties such as color and position.

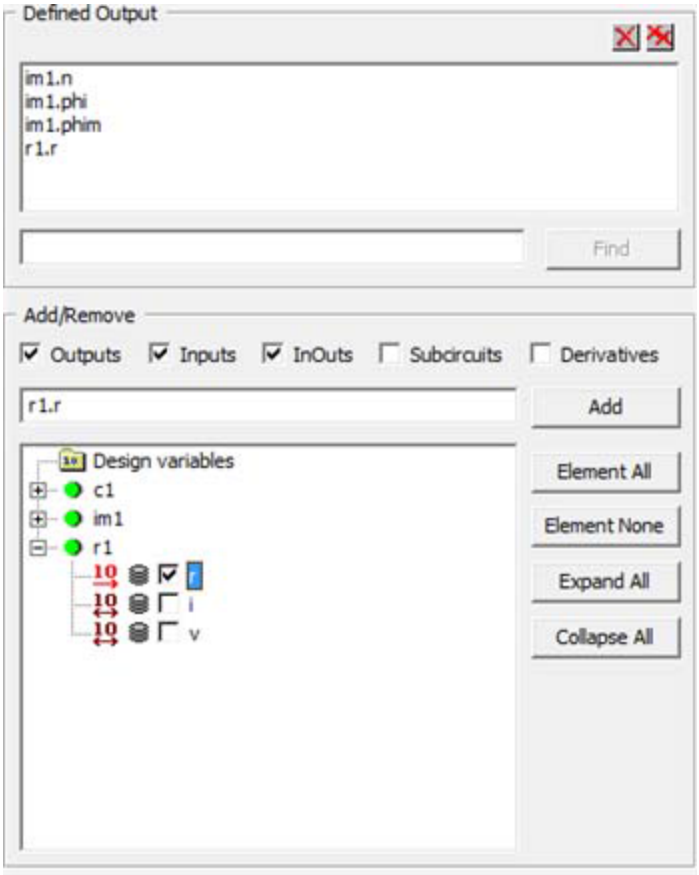
Other components may show other tabs, such as the **Signals** tab.

In addition, if a VHDL-AMS model has multiple architectures, then any one of the architectures can be selected for simulation. The **Parameter Values** tab also allows you to set the **Status** of the component to **Active**, **Inactive Open**, or **Inactive Short**. An **Info** button opens the help for detailed information on the component. See "Create a VHDL-AMS Model using the VHDL-AMS Editor" on page 5-9 and "Common Twin Builder Design Conventions" on page A-11.

Using Parameter Names

Model parameters can be used in expressions defining other model parameters and on the right side of equations.

Values of model parameters can be accessed using the following: Name.Qualifier, where Name refers to the name of the model and Qualifier refers to the name of the model parameter. For example, to access the resistance value *r* of the resistor model *resistor1*, use resistor1.r. Choose **Twin Builder** > **Output Dialog** and look at the **Output** dialog box, where all model parameters are listed with their qualifiers, to find the qualifier needed.



In Twin Builder, it is possible to connect components using the Name.Qualifier syntax. For example, the rotor speed (**n**) of an induction machine (**im1**) can be connected to the input of a gain block (**input**) by entering *im1.n* as the **Value** in the gain block's *input* parameter. While such connections can be made between VHDL-AMS models within the Twin Builder environment for simulation, it is not possible to generate and export the VHDL-AMS descriptions of schematics containing these types of connections. Therefore, if schematics need to be exported to ASCII files, then connections between components should be made explicitly with wires.

Parameter names used in Twin Builder are case sensitive while VHDL-AMS models are inherently case insensitive. All model names and parameter names for VHDL-AMS models in Twin Builder use lowercase.

See "Common Twin Builder Design Conventions" on page A-11.

Displaying Results

VHDL-AMS model inputs, outputs, locally defined elements, and values within submodels can be viewed using appropriate analog or digital Reports

Note:

Twin Builder allows you to monitor locally defined values by viewing them in display graphs, and also allows the use of these values as variables in other models. However, this is strongly discouraged if the Schematic has to be exported as a VHDL-AMS description.

Selecting Model Outputs

Reports, which can be created either by right-clicking **Results** on the **Project Manager Project** tab or via the **Twin Builder>Results** menu, allow the viewing of simulation outputs during and after the simulation. Simulation quantities can be selected, modified, and added to the report via the **Report** dialog box. Format characteristics of the representation can be set up before or after a simulation is run.

Note:

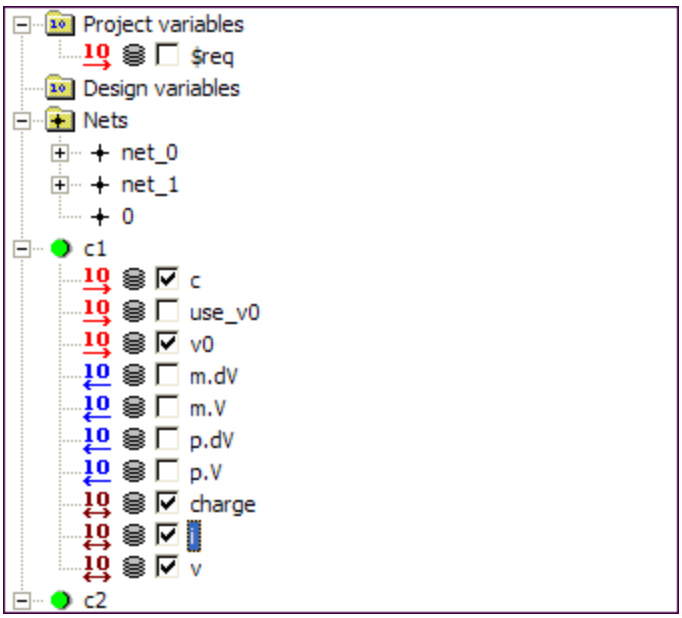
A **probe** is a quick way of selecting a parameter for display. The probe parameter can be set in the model's **Properties** dialog box. To add a probe report display to a sheet, right-click a model on the sheet and select **Probe**.

Outputs in Properties Dialog

Simulation quantities of a model instance on a schematic sheet can be saved in a database by checking its **SDB** box on the **Properties** dialog box. On the **Property Displays** tab, choose the quantities and other properties to make visible on the sheet.

Outputs in Reports

Reports allow the viewing of simulation quantities of a schematic sheet as online output. Reports can be stand-alone, or they can be placed directly on a sheet. To define the quantities that are to be made available for output, choose **Twin Builder>Output Dialog** to open the **Output** dialog box, and select the desired simulation quantities in the **Add/Remove** list. Only a model's defined simulation output quantities can be selected in a Report for display on the sheet. Each selected quantity, can have attributes such as color, displayed number component (real, imaginary, magnitude, or phase), Y-axis label and scale set for it. In the **Output** dialog box simulation quantity tree, model inputs are shown in red, model outputs are shown in blue, and model in/out quantities, as well as locally defined values, are shown in brown.



A quick explanation of how to set up a 2D **Rectangular Plot** follows. To learn about setting up a more complex report, read the next section, which contains a description of a Digital Plot.

Following is the general procedure for creating a new report:

1. On the **Twin Builder Circuit** menu or the Project tree, point to **Results**, select **Create Standard Report and from the cascading menu select Rectangular Plot** to open the **Report** dialog box.
2. In the **Context** section make selections from the following fields, depending on the design and solution type.
   a. **Solution** – a drop-down list of the available solutions, whether sweeps or adaptive passes.
   b. **Domain** – a drop-down list of domains relevant to the chosen **Solution**. For transient reports, the domain can be **Spectral** or **Time**. For AC reports, the domain is **Sweep**. For DC reports, the domain is **Time**. When **Spectral** is selected, additional fields display in which you can make settings for plotting spectral domain data.
   c. **Optimetrics setup** – a drop-down list of all defined Optimetrics analysis setups.
3. The **Update Report** section controls whether or not reports are updated in real time. By default, reports are updated in real time. If **Real time** is unchecked, use the **Update** button to update either the current report or all reports manually.
4. In the **Y** trace section of the dialog box make selections for the following:
   a. **Category** - contents of this list depend on the **Solution** type and the design. This field lets you specify the category of information for the Y component.

b. **Quantity** - lists the Y component items available in the selected **Category**.

> **Note:**
>
> The Quantity text field can be used to filter the Quantity list by typing in text. It is enabled if the Category selected produces a lengthy Quantities list.

c. **Function** - to apply to the Y quantities.

d. **Ytext box**- displays the currently specified **Quantity** and **Function**. You can edit this field directly.

> **Note:**
>
> The text color shows whether or not the expression is valid (blue for valid expressions, red for invalid).

e. **Range Function** button - opens the **Set Range Function** dialog box. This applies to the currently specified Quantity and Function.

5. In the **X (Primary Sweep)** section, make selections for the following:
   a. Select the Primary Sweep value(s) from the drop-down menu.
   b. If sweeps are available, you can select the browse button **[...]** to display a dialog that lets you select particular sweep or sweeps, or all sweeps.
6. On the **Families** tab, if families are available, make selections for the following:
   a. Select either **Sweeps** display or **Available variations** display radio buttons.
      - **Sweeps** allows you to edit the swept variable values that will be displayed by clicking the **[...]** button in the **Edit** field., and then choosing the desired variable values in the pop-up window. (By default, all values are selected.)
      - **Available variations** allows you to select individual combinations of values via checkboxes. Click the Select column heading to select or clear all checkboxes simultaneously. Click the variable name column heading to sort the listing in either descending or ascending order.
   b. In the **Nominals** field (disabled if none exist in the design) choose either **Set All Variables to Nominal**, **Set All Unswept Variables to Nominal**, or to **Choose Nominals** to open a dialog box in which you can select the variables you want to set to nominal values.
7. On the **Families Display** tab, make selections for the following:
   a. Choose **All Families** (default selection) to enable traces for all families selected on the **Families** tab to be displayed.

b. Choose **Statistics** to select various statistical functions to apply to the traces selected on the **Families** tab. The selected functions can then be plotted on a new report, or added to the currently active report.

c. Choose **Histogram** to generate a histogram plot based on the family of curves selected on the **Families** tab. Observe the following general guidelines when generating histograms:

- Use Time as a primary sweep for **TR** solutions. Use Frequency as a **Primary Sweep** for **AC** solutions.)
- For the X-axis definition, only the available **Primary Sweep** quantity can be used.
- For the Y-axis definition, any available quantity can be selected.
- A set of *N* variations, where *N* is greater than one, must be selected under the **Families** tab.
- Set the desired **Number of bins** (the maximum number of rectangles used to represent the number of outcomes). The number of bins must be between 2 and 1000.
- Set the **Value to sample at** - the instance value of the selected **Primary Sweep** – in milliseconds for Time, or kiloHertz for Frequency – at which the number of outcomes are computed among all *N* selected variations.

h. Use the **Report** dialog command buttons to create a new report with the settings you provided above, or to modify an existing report.

- **New Report**. Adds a report to the Project tree under the Results icon. The new report is displayed in the Project window.
- **Add Trace** – enabled when you have added a New Report, or selected an existing report to modify. Click this to add one or more traces to the report. When you add traces, the new traces are displayed in the Project window under the report. This is enabled when you have created or selected a report.
- **Apply Trace** – updates the selected traces in a report based on further processing or changes. When you edit a trace, this button applies the current values to that trace.
- **Output Variables** – opens the Output Variables dialog in which you can add output variables and expressions.
- **Options** – opens the Report Setup Options dialog box. This contains a check box for using the advanced mode for editing and viewing trace components. This mode is automatic if the trace requires it. It also contains a field for setting the maximum number of significant digits to display for numerical quantities.
- **Close** – closes the **Modify Report** dialog.

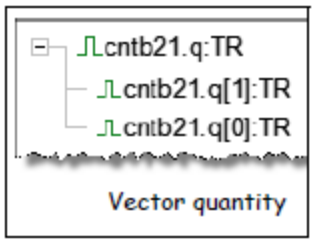
i. Click **New Report to create a new report in the Project tree.**

The report appears in the view window. It will be listed in the project tree under Results. Traces within the report also appear in the project tree. Some plots may take time to complete. Performing a **File>Save** in such cases after the plot has been created will permit you to review the plot later without having to repeat the calculation time when you reopen the project later.

Drag a report icon to the schematic sheet to create an on-sheet plot. Change the size of the on-sheet plot by selecting it and dragging the sizing handles with the cursor. To move the plot, select it, hold the mouse button down, and drag the plot to a new location.

> **Note:**
>
> Objects can be edited within the on-sheet plot directly. Right click on the plot and select **«Edit In Place»**. To edit a component of the Display Element, double-click it to open a Properties dialog where the font, color, and so on can be modified. Components include the legend, title, axis labels, and individual graph plots.

Digital Plot



The Digital Plot is especially designed to display digital signals of VHDL-AMS models. In contrast to the Rectangular Plot, each signal has its own coordinate system. The Digital Plot can show vector quantities, special display formats, and data types. To show and hide members of a vector, click the + sign in the diagram legend.

Double-clicking an on sheet plot opens its Report dialog box in which the assignment of simulation quantities such as traces are defined.

Double-clicking a trace, header, legend, or x-axis element on the plot provides access to the Properties tabs described in the following sections.

Attributes Tab

The assignment of trace name, type and line width, color, symbol style, etc. Depending on the data type and the display format, different data formats for representation can be specified. The decimal number 100 would be represented as *64* in hexadecimal format, as *144* in octal format, as *0110 0100* in binary format, and as *d* in ASCII format.

Digital Tab

Defines the Literal Foreground color, enables/disables use of a scrollbar, and controls the fit and height of the various trace types.

Grid Tab

Defines the visibility, color, line style of grid lines and of the trace area border.

General Tab

Defines the background and plot area colors, and the field width, precision, and notation type for markers. (To define a marker, right click on a trace and select **Marker>X Marker**.)

Legend Tab

Defines various properties of the plot legend including its position (left or right side of the plot); visibility of trace name, solution name, font, legend border and background color, and grid line width and color.

Header Tab

Defines the fonts for the plot title and sub-title; editing of the Company Name text; and the visibility of the design name.

4 - Case Studies

The case study examples in this chapter extend the basic set of features introduced so far. They include analog, mixed-signal, and mixed-technology models.

The first case explains different modeling methods using the modified battery model from the Automotive Powernet System example. The second case uses a fuse model to illustrate the use of the VHDL-AMS Model Editor to create a model and provide symbol animation. The third case shows the influence of model descriptions on simulation time using a detailed and an averaged claw-pole alternator. The fourth case uses a load model to explain the concept of multiple architectures for a single model, and the concept of component instantiation as a way to reuse a particular model behavior. The fifth case uses a linear drive system and a simple solenoid model to explain multidomain modeling. The sixth case uses a PWM controller to explain mixed-signal modeling, and an automotive alarm system to illustrate using a stimulus generator, exporting VHDL-AMS models, and using foreign models.

Each of the case studies contains:

- A *Concepts* section that describes the model design concept illustrated
- One or more *Background* sections that describe the physical systems being modeled
- One or more *Model* sections that describe the methods used to define the models

This chapter contains information on:

- Multiple modeling styles: battery model
- Model development using the VHDL-AMS Editor: fuse model
- Detailed and averaged model development: claw-pole alternator model
- Multilevel modeling techniques: load model
- Multidomain system modeling: linear drive system, solenoid
- Mixed signal modeling: DC-DC converter with PWM, automotive alarm system

Using Example Sheets

The examples described in this chapter and the required library *vhdlams_tutorial.asmd* are available on the Student Version CD. Twin Builder is needed to create and simulate the examples. The Student Version has some limitations in the model design. If an example cannot be executed with the student version, there will be a note on its sheet. Since the Student Version limits the size of a design, some models need to be excluded from the simulation. To exclude models, select the component instances that need to be excluded and choose **Edit** > **Deactivate (Open)**. The selected components will have a large red “X” superimposed on them, indicating that they are not included in the simulation.

The examples can be loaded from the tutorials files, or can be created from scratch. The project files referred to in this chapter are in the Tutorial Examples folder. The examples can be done in

any order, since each case explains a different concept and is not based on a previous case. If there are no comments about changing parameter values in these examples, the default values are used.

VHDL-AMS Modeling Features

The following table lists VHDL-AMS features and models used in the case studies (including the example file name), and indicates if the examples can be run in the Student Version (SV).

Case	VHDL-AMS Features	Model/Sheet Name	SV
1	• Graphical, structural, behavioral modeling methods • Component instantiation • Graphical and text subsheets	Battery *case_study_battery.aedt*	yes
2	• Automated model development • VHDL-AMS Model Wizard • Component instantiation • Symbol animation	Fuse *case_study_fuse_lamps.aedt*	partly
3	• Modeling descriptions to reduce simulation time • Graphical and text subsheets	Averaged and detailed claw-pole alternator *case_study_clawpole_avg.aedt* *case_study_clawpole_math.aedt*	no
4	• Multiple architectures • Component instantiation • Symbol animation	Load models *case_study_loads.aedt*	partly
5	• Multidomain modeling • Maxwell coupling	Linear Drive and Solenoid *case_study_em_linear_drive.aedt* *case_study_em_solenoid.aedt*	partly
6	• Mixed-signal modeling • Stimulus generator • Multidomain sensor modeling • Foreign models • Export of VHDL-AMS netlists	PWM Controller and Automotive Alarm System *case_study_pwm_dcdc.aedt* *case_study_automotive_alarm_system.aedt*	yes

Different Modeling Styles: Battery

Concepts - Battery Model

The first case study uses a battery model to illustrate how different modeling styles can be used in Twin Builder to obtain the same behavior.

The battery is modeled in three different ways:

- Graphical description using subcircuit modeling
- Structural description in ASCII text using component instantiation
- Behavioral description in ASCII text using differential algebraic equations

Each method provides the same results but has advantages and disadvantages. Although the three modeling styles provide the same output, there are some fundamental differences between the modeling styles.

Graphical Modeling Style - Battery Model

The graphical modeling style is the easiest method for developing the battery model, since all the necessary submodels are available in the library. This method does not require knowledge of VHDL-AMS syntax and programming, and the models can be dragged and dropped from a library onto the sheet directly. This modeling method is recommended for beginners.

- ADVANTAGE: This modeling method is easy to use.
- DISADVANTAGE: The building block models must be obtained from somewhere.

Structural Modeling Style - Battery Model

The structural modeling style is useful when a design needs to be developed using specific models in VHDL-AMS text format. When this style is used, the end result is a VHDL-AMS text netlist.

- ADVANTAGE: This method is good for users who are more comfortable in writing VHDL-AMS models in ASCII format.
- DISADVANTAGE: The building block models and advanced knowledge of AMS syntax are required.

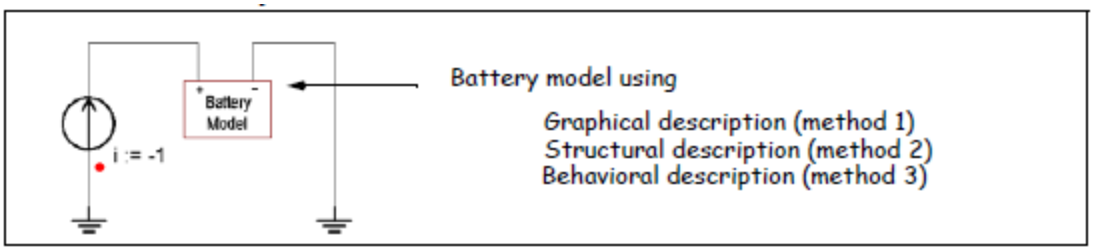
Behavioral Modeling Style - Battery Model

The behavioral modeling style is useful for describing model function in an abstract manner, using equations and programming constructs. This modeling method is recommended for experts.

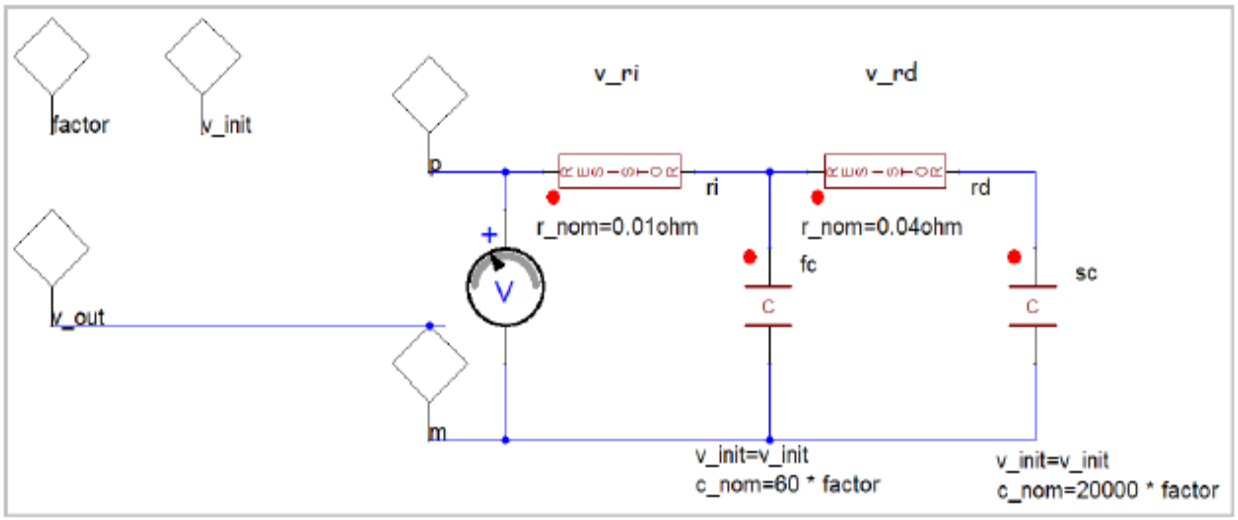
- ADVANTAGE: The modeling and simulation can be optimized because the model description is self-contained.
- DISADVANTAGE: The model description may not correspond to any real physical model.

Background - Battery Modeling Styles

The following figure shows the basic test circuit for all of the modeling styles. In each style, the battery is replaced with an equivalent model. The current source has a value of –1A, indicating that it draws current from the battery model.



The battery model is designed as a simple circuit with two resistors and two capacitors as shown in the following circuit:



The battery model also accepts two parameters: *v_init* specifies the initial voltage of the battery, and *factor*, a parameter that expresses the capacity factor and aids in simple scaling of the battery. The internal resistance (ri=10mOhm), diffusion resistance (rd=40m), the fast capacitor (fc=60*factor), and the slow capacitor (sc=20000*factor) have constant values.

The model calculates the circuit based on the following equations:

```
v_ri = i_ri * ri

d(v_fc)/dt = 1/(fc*factor) * i_fc

v_rd = i_rd * rd
```

d(v_sc)/dt = 1/(sc*factor) * i_sc

The factor value is used to scale the two capacitances from their default value. Thus, factor is a normalized charge capacity for the battery.

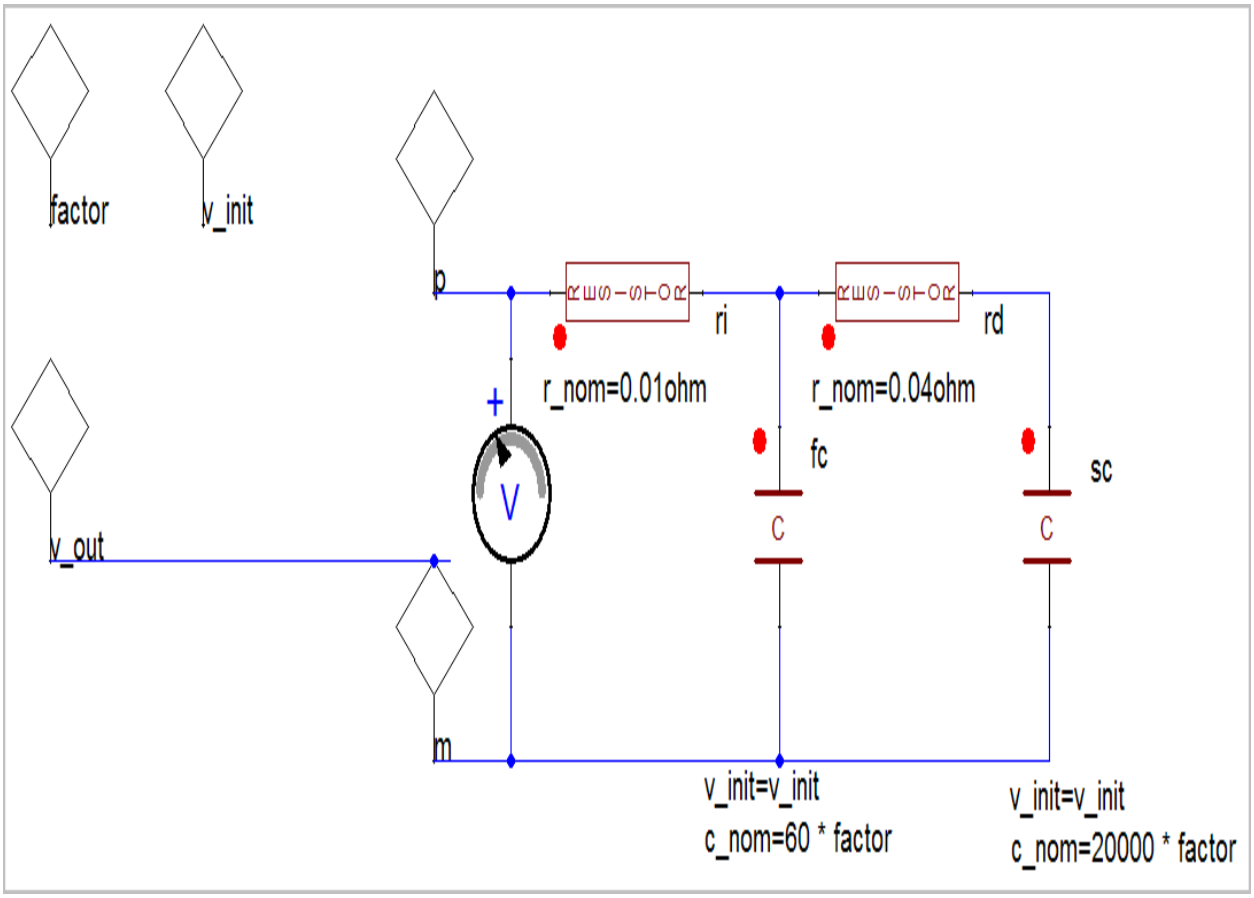
See "Battery Model" on page 2-15 also. Unlike the battery model in the Automotive Powernet System example, this battery model does not provide State of Charge (SoC) as an output parameter.

Model Style 1: Graphical Description

This modeling style uses models from the library to graphically generate the battery model.

To create the graphical battery subcircuit, do the following:

1. Select **Twin Builder** > **SubCircuit** > **Add SubCircuit**. A new, empty schematic subcircuit sheet appears. On the main schematic, a rectangular subsheet symbol also appears.
2. Place and arrange all models used corresponding to the following circuit:



Resistor and *Capacitor* are VHDL-AMS models from the **vhdlams_tutorial** folder under **System Libraries** in the **Component Libraries** dialog box. The *Voltmeter* is a model from the *Measurement* folder of the *Basic Elements VHDLAMS* library.

3. Create the model Interface:
   - Establish conservative nodes (terminal *p* and *n*) using **Draw** > **Interface Port**. Double-click the port symbol to open the **Port Properties** dialog box. Conservative nodes are connectors from the substructure to the next higher model level. Define the **Port Name** of the conservative node, and select **Conservative** for the domain and **Electrical** as the nature type. The new pin symbol can now be connected with the wiring.
   - Establish non-conservative nodes (*factor*, *v_init*, *v_out*) using **Draw>Interface Port**. Non-conservative nodes are used to change parameters. Select **Quantity** for the domain to establish the port as non-conservative. Leave the type as **real**. Non-conservative nodes need to be specified as **In**put, **Out**put, or input and output (**InOut**). In this example *factor* and *v_init* are input parameters, and *v_out* is an output parameter.

     > **Note:**
     >
     > A pin symbol appears for each non-conservative node, even when the pin is not connected through a wire to a model parameter.

4. Switch to the next higher model level using **Pop Up** in the subsheet shortcut menu. Double-click the subsheet symbol to open the **Properties** dialog box. On the **Quantities** tab the defined non-conservative terminals are listed.

To view the *v_out* value through a Probe Element on the sheet, click with the right mouse button on the subsheet symbol, and select **Probe** > **v_out**. An on-sheet plot with *v_out* as output quantity is added to the sheet. Drag it to the desired location and resize it as needed. Change the presentation format by right-clicking on the plot with the right mouse button and selecting **Edit in Place**.

Model Style 2: Structural Description

This modeling style uses the structure of the battery model developed with Style 1 to describe the battery model in a text format. In this example VHDL-AMS Style are used to instantiate the required components, similar to Style 1. This differs from Method 1, however, in that the netlist is specified using VHDL-AMS text, whereas it was generated from graphical specifications in Method 1.

For the battery model, we require two capacitor models and two resistor models that are available in the library. The models can be accessed from the library if the following statements are added at the beginning of the entity description:

```
LIBRARY WORK;
USE WORK.ALL;
```

The **USE** statement specifies that all models and packages that are defined within the *WORK* library are now accessible from any architecture of the battery model.

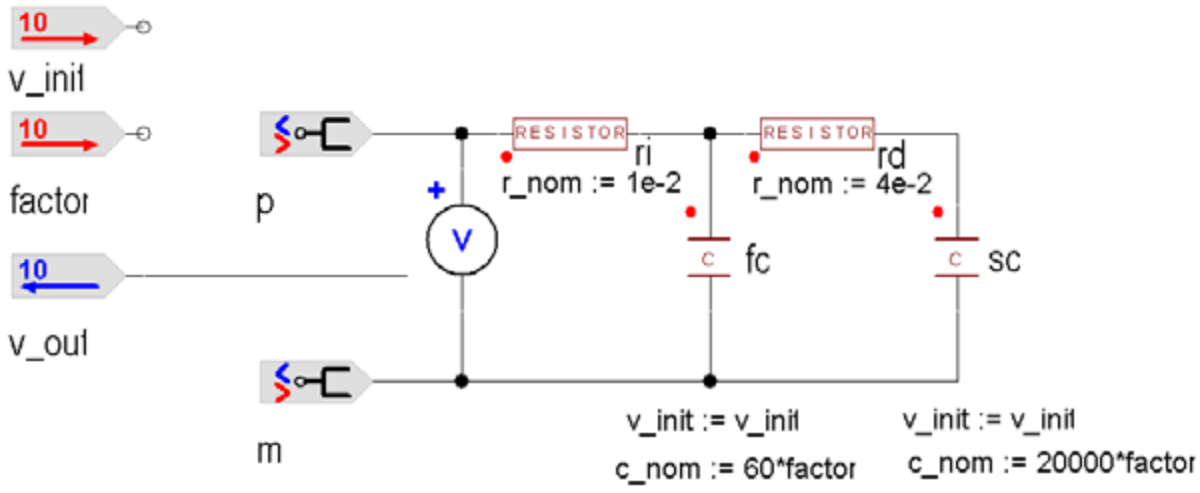
Entity Description - Structural Battery Model

The entity description of the structural battery model uses static parameters (constant value inputs evaluated only at the beginning of the simulation) and terminals in the interface. The battery model has two parameters for the factor and initial voltage (*factor*, *v_init*) as defined in the **GENERIC** declaration. These parameters are of type REAL with specific default values. The battery provides its electrical output through a pair of electrical terminals and provides the voltage across the battery through the *v_out* output defined in the **quantity** statement.

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
LIBRARY WORK;
USE WORK.ALL;
ENTITY bat_multi IS
GENERIC(
factor : REAL := 1.0;
v_init : VOLTAGE := 12.0);
PORT(
TERMINAL p,m : ELECTRICAL;
QUANTITY v_out : OUT VOLTAGE := 0.0);
END ENTITY bat_multi;
```

Architecture Description - Structural Battery Model

The *struct* architecture of the battery model defines two internal terminals of type ELECTRICAL, *t1* and *t2*, within the model of type ELECTRICAL. Also, the model defines constant resistance values for *rd* and *ri*, as well as constant capacitance values for *fc* and *sc*.



The following model instantiation uses the model *cap* from the **WORK** library. The **GENERIC MAP** and **PORT MAP** statements assign the model parameters and terminals to the *cap* entity names, *c_nom*, *v_init*, *p*, and *m*.

```
fc1: ENTITY cap(behav)
GENERIC MAP (c_nom => fc*factor, v_init => v_init)
PORT MAP (p => t1, m => m);
```

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE struct OF bat_multi IS
 CONSTANT ri: RESISTANCE := 1.0e-2;
 CONSTANT fc: CAPACITANCE := 60.0;
 CONSTANT rd: RESISTANCE := 4.0e-2;
 CONSTANT sc: CAPACITANCE := 2.0e4;
 TERMINAL t1,t2 : ELECTRICAL;
 QUANTITY v ACROSS p TO m;
BEGIN
 fc1: ENTITY cap(behav)
 GENERIC MAP (c_nom => fc*factor, v_init => v_init)
 PORT MAP (p => t1, m => m);
```

```
sc1: ENTITY cap(behav)
GENERIC MAP (c_nom => sc*factor, v_init => v_init)
PORT MAP (p => t2, m => m);
ri1: ENTITY res(behav)
GENERIC MAP (r_nom => ri)
PORT MAP (p => p, m => t1);
rd1 : ENTITY res(behav)
GENERIC MAP (r_nom => rd)
PORT MAP (p => t1, m => t2);
v_out == v;
END ARCHITECTURE struct;
```

Method 3: Behavioral Description - Battery Model

This modeling style uses differential algebraic equations (DAE) to describe the behavior of the battery model. It does not instantiate other components, so the resulting model is more compact and consequently, more efficient.

Entity Description - Behavioral Battery Model

The entity description of the behavioral battery model uses static parameters (constant value inputs evaluated only at the beginning of the simulation) and terminals in the interface. The battery model has two parameters for the factor and initial voltage (*factor*, *v_init*) as defined in the **GENERIC** declaration. These parameters are of type REAL with specific default values.

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
LIBRARY WORK;
USE WORK.ALL;
ENTITY bat_multi IS
 GENERIC(
 factor : REAL := 1.0;
 v_init : VOLTAGE := 12.0);
 PORT(
```

```
 TERMINAL p,m : ELECTRICAL;
 QUANTITY v_out : OUT VOLTAGE := 0.0);
END ENTITY bat_multi;
```

Architecture Description - Behavioral Battery Model

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE behav OF bat_multi IS
 TERMINAL t1, t2: ELECTRICAL;
 QUANTITY v_ri ACROSS i_ri THROUGH p TO t1;
 QUANTITY v_fc ACROSS i_fc THROUGH t1 TO m;
 QUANTITY v_rd ACROSS i_rd THROUGH t1 TO t2;
 QUANTITY v_sc ACROSS i_sc THROUGH t2 TO m;
 QUANTITY v ACROSS p TO m;
 CONSTANT ri: RESISTANCE := 1.0e-2;
 CONSTANT fc: CAPACITANCE := 60.0;
 CONSTANT rd: RESISTANCE := 4.0e-2;
 CONSTANT sc: CAPACITANCE := 2.0e4;
BEGIN
 BREAK v_fc => v_init, v_sc => v_init;
 v_ri == i_ri * ri;
 v_fc'DOT == 1.0/(fc*factor) * i_fc;
 v_rd == i_rd * rd;
 v_sc'DOT == 1.0/(sc*factor) * i_sc;
 v_out == v;
END ARCHITECTURE behav;
```

Two internal electrical terminals, *t1* and *t2*, are created as in Style 2. The four quantity statements define through and across quantities for the two resistors and capacitors. Together, these five statements define the internal topology of the model.

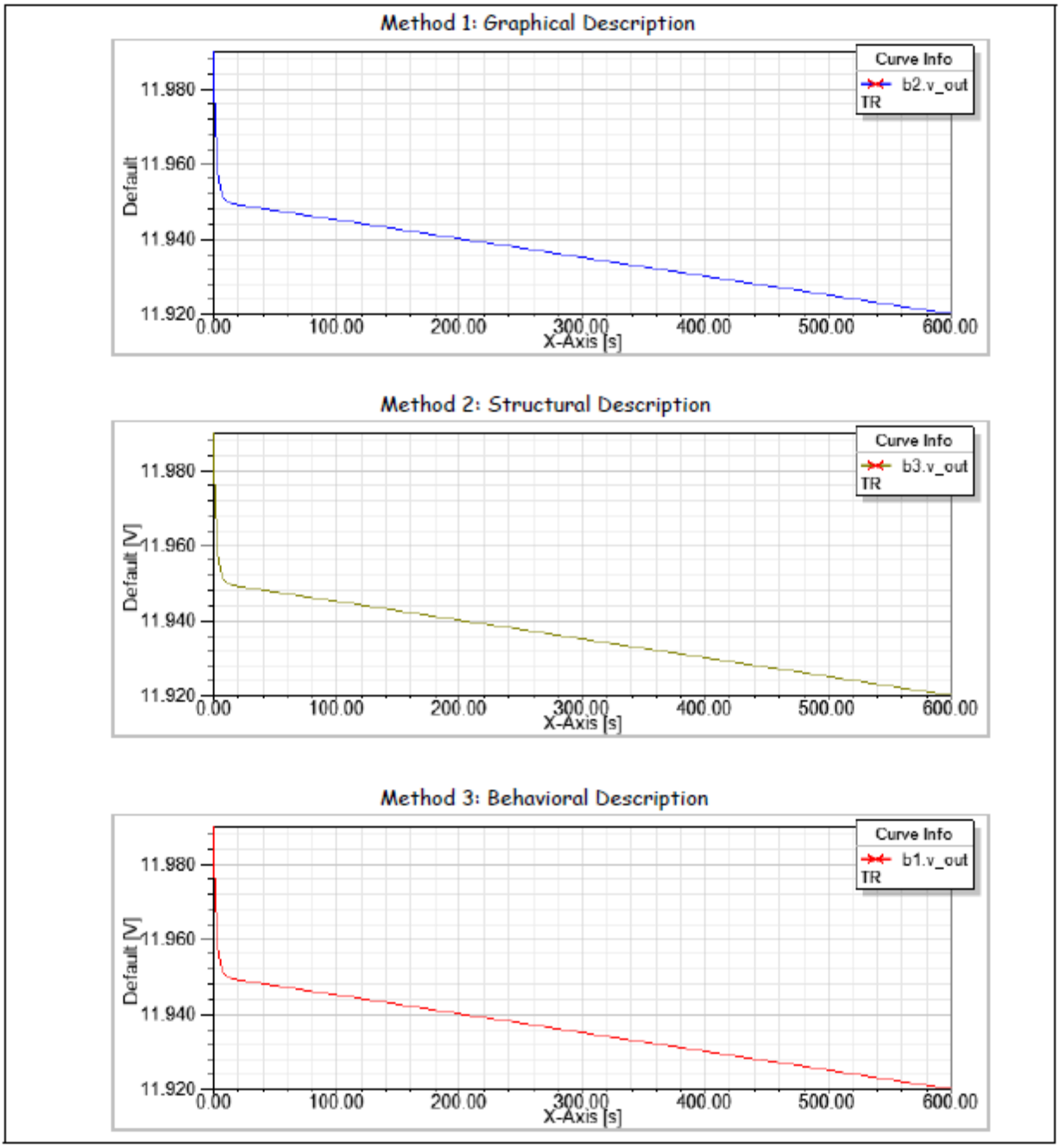
After the **break** statement, the four equations describe the branch characteristics of each internal resistor and capacitor. To avoid discontinuities during simulation, the **BREAK** statement is used to initialize the voltages across the fast and slow capacitors with the value of *v_init* (initial voltage).

The voltage between the terminals is provided as output voltage, with the simultaneous statement vout == v.

Results - Behavioral Battery Model

The Plots show the battery output voltage of the simulation. Since the battery model is connected to a constant current sink, the battery voltage decreases with time as the battery slowly discharges. A fast discharge, governed by capacitor *fc*, is followed by a slow discharge, governed by capacitor *sc*.

Even though different modeling styles have been adopted, the outputs of all three battery models are identical, indicating that all models display the same behavior.



Automated Model Development Using VHDL-AMS Wizard: Fuse

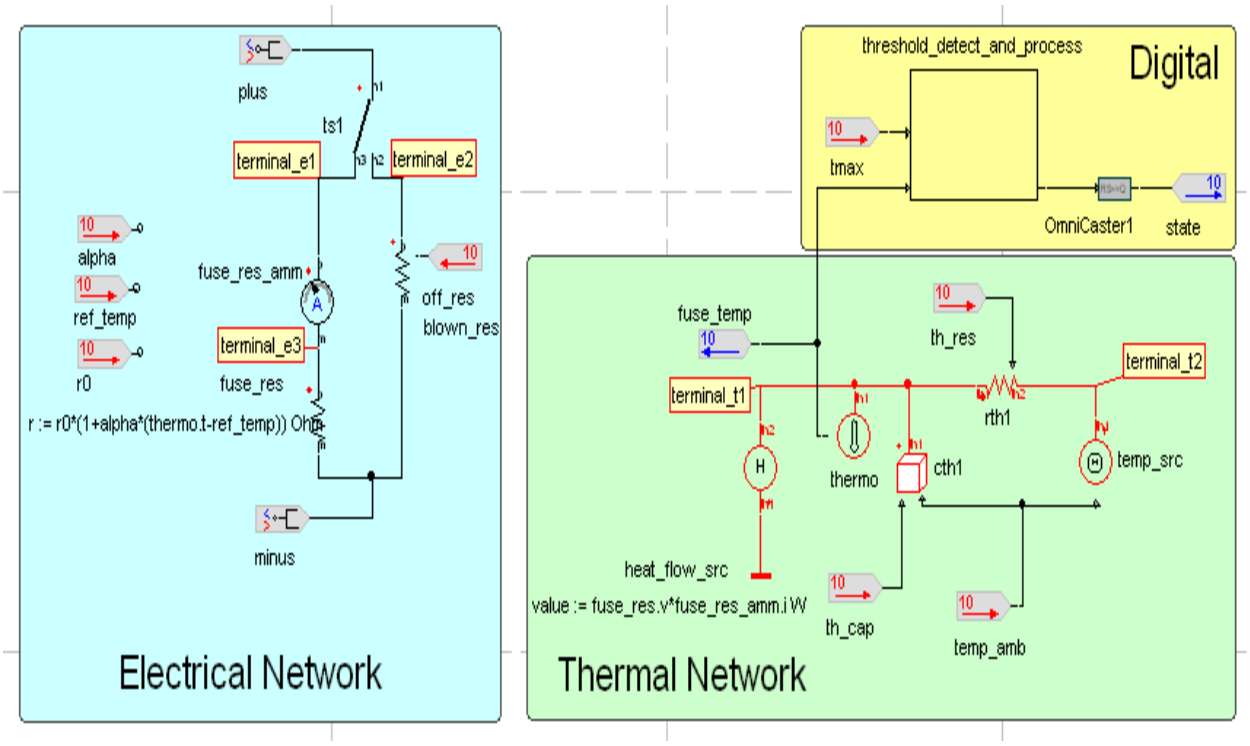
Concepts - Automated Model Development

This case study illustrates the use of the VHDL-AMS Model Wizard to develop a model, the creation of an animated symbol for the model, and the use of the model in a simulation.

Using the VHDL-AMS Wizard for Automated Model Development

Fuse Model Equivalent

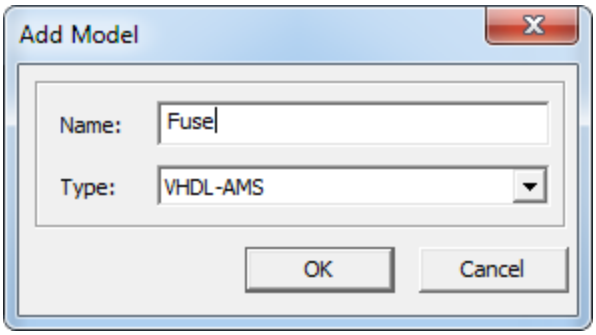
The following is the schematic equivalent of the VHDL-AMS text netlist that is created using the VHDL-AMS Wizard. The fuse model is an example of a multidomain, mixed-signal model involving the electrical and thermal domains. The fuse model can be divided into three parts: a thermal network, an electrical network, and a digital logic unit.



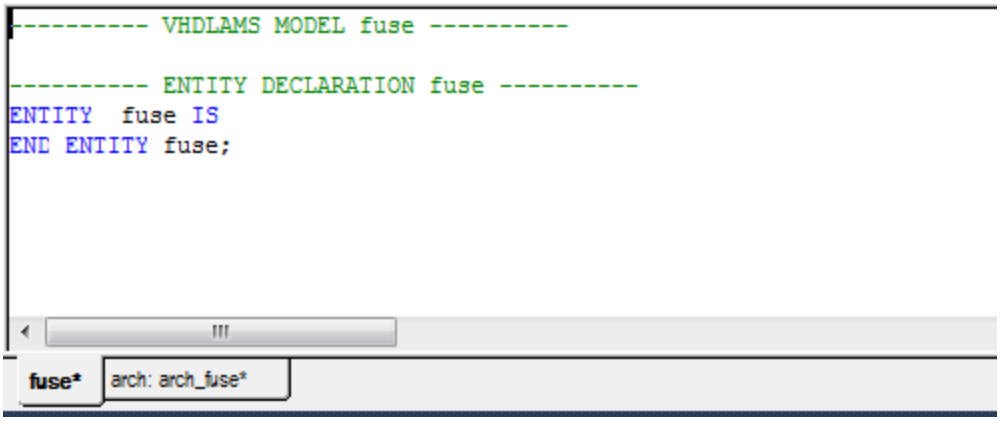
The thermal network models the self-heating behavior of the fuse. It also considers the fuse's dynamic heat exchange with the surroundings using a thermal resistance and a thermal capacitance. The influence of the ambient temperature on the fuse is modeled by a temperature source connected to the thermal network. The digital logic determines the threshold crossing of the fuse temperature over the specified maximum temperature. The output from the digital logic is used to control a switch in the electrical network that switches between the fuse resistance and an off resistance. The fuse resistance is calculated based on the temperature of the thermal network and the material-specific parameters, while the off resistance is provided as a parameter for the model.

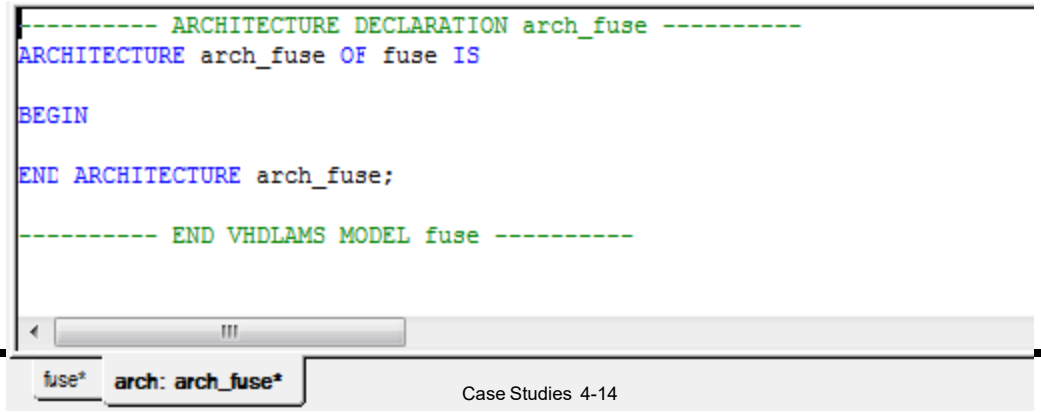
Create a VHDL-AMS Framework for the Fuse Model

1. Select **File** > **New** or the New document icon to open a project. Select **Tools** > **Edit Libraries** > **Models** to open the **Edit Libraries** dialog box.
3. Click **Add Model** to open the **Add Model** dialog box.

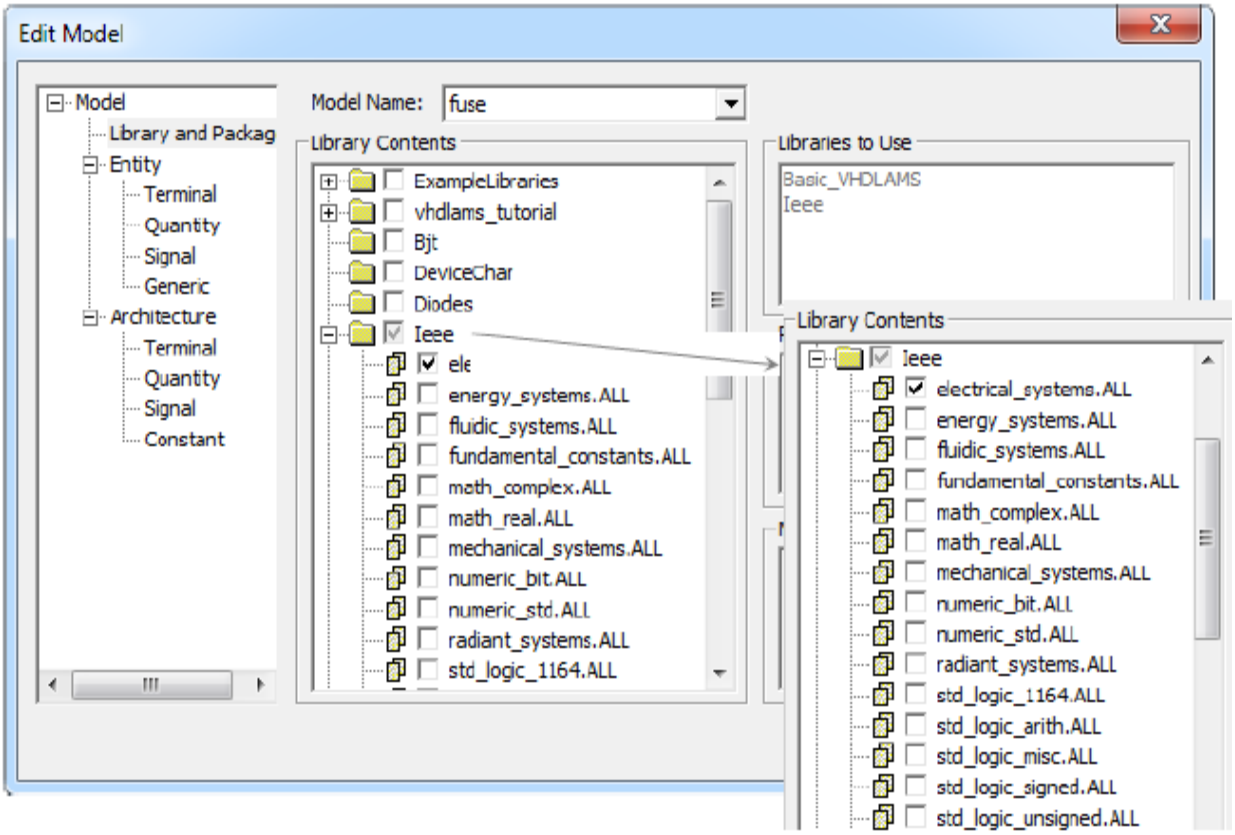


4. Enter *Fuse* as the **Name** and select **VHDL-AMS** as the **Type**.
5. Click **OK**. An empty VHDL-AMS code framework is created and displayed in an editing window containing two tabs bearing the model name. The left tab contains the fuse entity declaration framework. The right tab contains the architecture declaration framework for the model.





6. In the Twin Builder menu bar, select **VHDL Model Editor** > **Edit Libraries and Packages** to open the **Edit Model** dialog box.



7. In the **Library Contents** frame, select the *Ieee* and *Basic_VHDLAMS* libraries.
8. Click the "+" sign next to the *Ieee* folder to open it and view its contents. Select the *electrical_systems.all* and **thermal_systems.all** packages. Do not select any models in the *Basic_VHDLAMS* library.
9. Click **OK** to close the dialog box. Note that VHDL statements for the libraries and packages you chose have been added to the fuse entity declarations tab.
10. Save the model file as *fuse.vhd* by selecting **VHDL Model Editor** > **Save Fuse**.

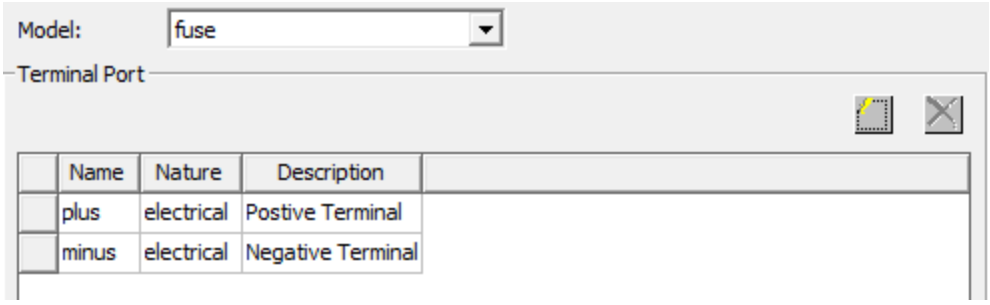
Describe the Entity - Fuse Model

This section describes how to add terminal, quantity, and signal ports as well as generics to the fuse entity.

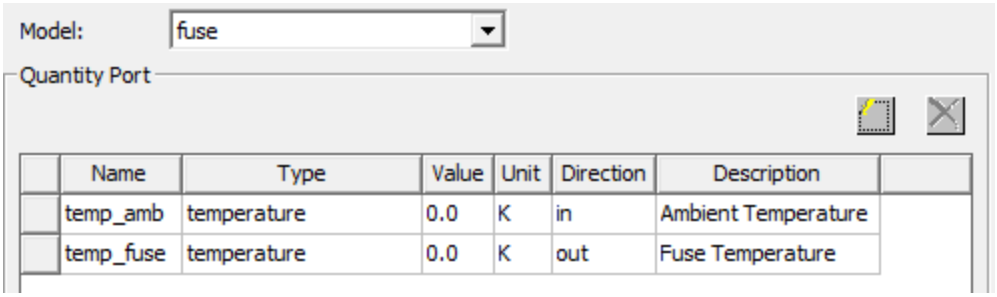
Note:

Port input quantities and signals can have an initial value. Generics can also have an initial value. To set an initial value, enter it in the **Value** field of that item in the dialog box.

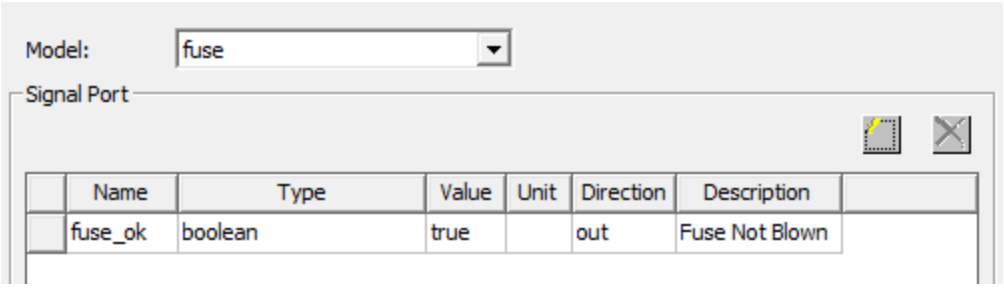
1. Select **VHDL Model Editor** > **Edit Entity** > **Terminal** to open the **Edit Model** dialog box. The Terminal entity in the model tree is highlighted and the Terminal Port frame is displayed to its right.
   a. In the **Terminal Port** frame, click **Add** twice to create two terminal ports. Configure the ports as shown below.



   b. Click **Quantity** in the model Entity tree, and in the **Quantity Port** frame click **Add** twice to create two quantity ports. Configure the ports as shown below.

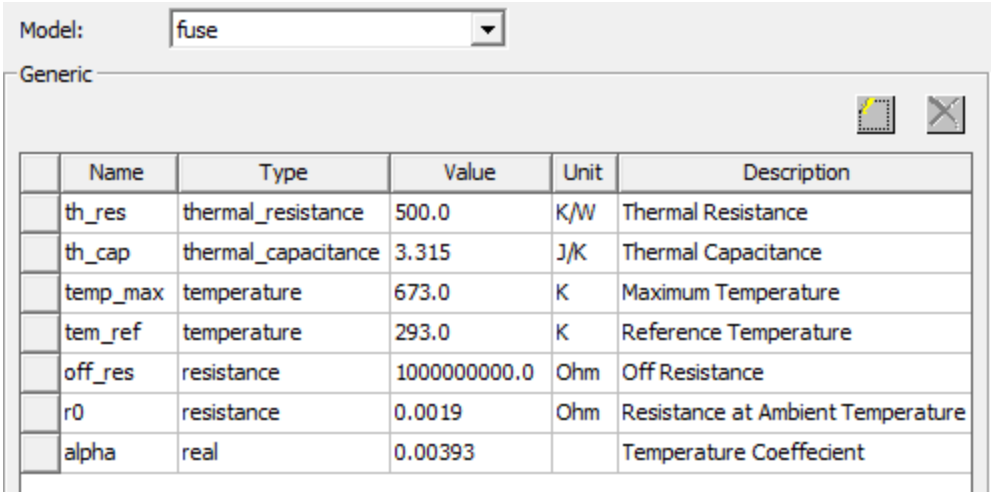


   c. Click **Signal** in the model Entity tree, and in the **Signal Port** frame, click **Add** to create one signal port. Configure the port as shown below.



Name	Type	Value	Unit	Direction	Description
fuse_ok	boolean	true		out	Fuse Not Blown

Click **Generic** in the model Entity tree, and in the **Generic** frame, click seven times on **Add** button to create seven generic value definitions. Configure them as shown below.



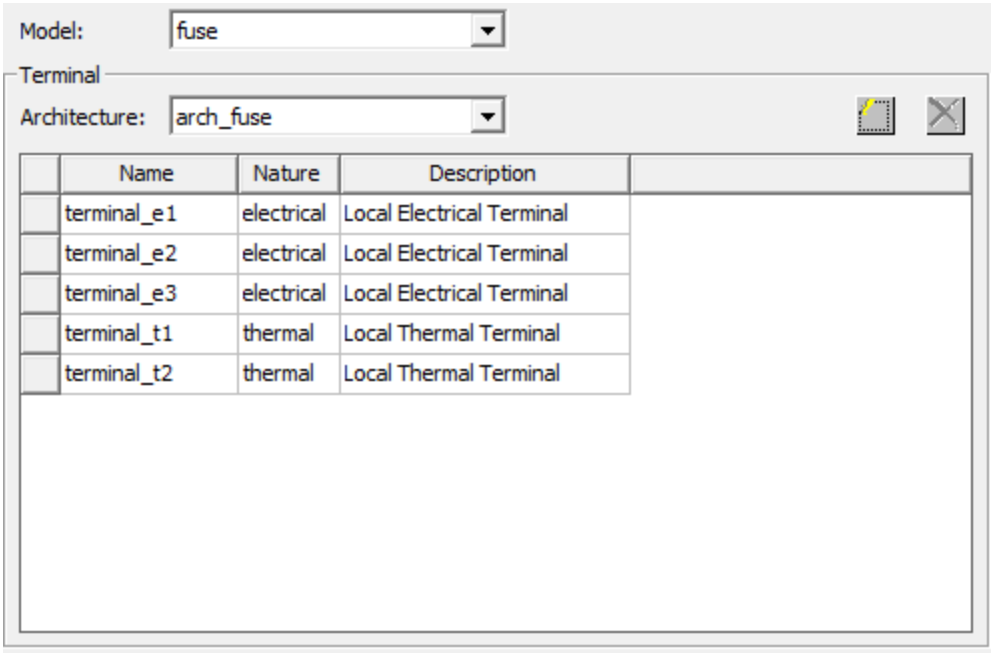
Name	Type	Value	Unit	Description
th_res	thermal_resistance	500.0	K/W	Thermal Resistance
th_cap	thermal_capacitance	3.315	J/K	Thermal Capacitance
temp_max	temperature	673.0	K	Maximum Temperature
tem_ref	temperature	293.0	K	Reference Temperature
off_res	resistance	1000000000.0	Ohm	Off Resistance
r0	resistance	0.0019	Ohm	Resistance at Ambient Temperature
alpha	real	0.00393		Temperature Coeffecient

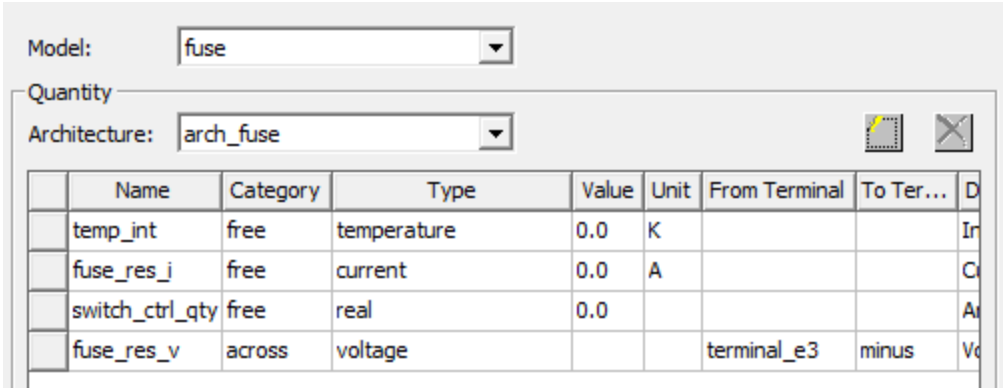
2. Click **OK**. The VHDL-AMS code for the entity declaration generates in the editor window.

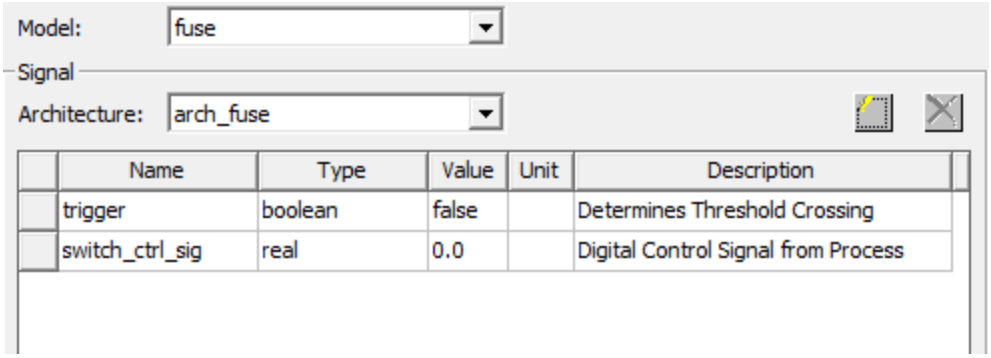
Describe the Architecture - Fuse Model

1. Select **VHDL Model Editor** > **Edit Architecture** > **Terminal** to open the **Edit Model** dialog box. The Architecture Terminal in the model tree is highlighted and the Terminal frame appears to its right.

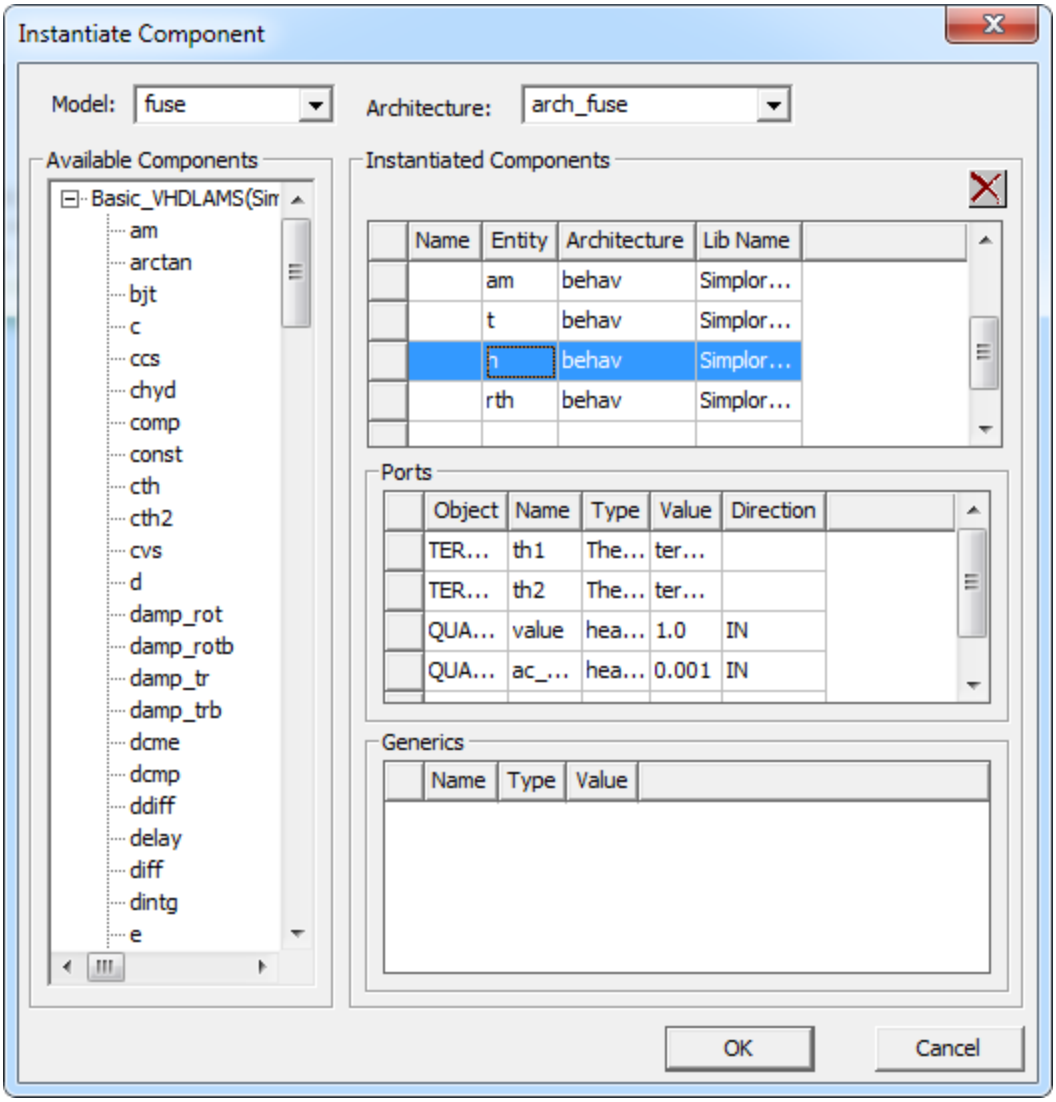
2. Add the following terminal, quantity, and signal declarations and click **OK**:







3. On the Twin Builder main menu bar, choose **VHDL Model Editor** > **Instantiate Component** to open the **Instantiate Component** dialog box.



4. In the *basic_vhdlams* library model tree, double-click on the following models to add them to the fuse model:

Electrical Network
r - Resistor (double-click twice for two models)
ts - Ideal Transfer Switch
am - Ammeter
Thermal Network
t - Temperature Source
h - Heat Flow Source

Electrical Network
rth - Thermal Resistance
cth - Thermal Capacitance
thm - Thermometer

5. In the **Instantiate Component** dialog box, click each model and set the entity names and port map values for each model as shown in the following table. Note that most values can simply be selected from a drop-down menu. Only two equations must be entered manually.

Entity	Name	Port Map	
		Name	Value
r	fuse_res	r	r0*(1.0+alpha*(temp_int-temp_ref))
		p	terminal_e3
		m	minus
r	blown_res	r	off_res
		p	terminal_e2
		m	minus
ts	three_port_sw	ctrl	switch_ctrl_qty
		n1	plus
		n2	terminal_e2
		n3	terminal_e1
am	fuse_res_amm	i	fuse_res_i
		p	terminal_e1
		m	terminal_e3
t	temp_src	value	temp_amb
		ac_mag	1m (leave as is)
		ac_phase	0 (leave as is)
		th1	terminal_t2
h	heat_flow_src	value	fuse_res_i*fuse_res_v
		ac_mag	1m (leave as is)
		ac_phase	0 (leave as is)
		th1	thermal_ref
		th2	terminal_t1

Entity	Name	Port Map	
		Name	Value
rth	therm_res	k	th_res
		th1	terminal_t1
		th2	terminal_t2
cth	therm_cap	c_th	th_cap
		t0	temp_amb
		th1	terminal_t1
thm	thermo	temp	temp_int
		th1	terminal_t1

6. Click **OK** to close the dialog box.
7. On the arch_fuse tab, insert the following text at the bottom of the existing code.

```
    temp_fuse == temp_int; -- Output

    trigger <= temp_int'ABOVE(temp_max);
    BREAK ON trigger;
    PROCESS IS
    BEGIN
      switch_ctrl_sig <= 0.0;
      WAIT UNTIL trigger;
      switch_ctrl_sig <= 1.0;
      fuse_ok <= FALSE;
      WAIT;
    END PROCESS;

    switch_ctrl_qty == switch_ctrl_sig'RAMP(0.0,0.0);

END ARCHITECTURE arch_fuse;
```

The fuse model has been successfully created. The next step is to include it in a schematic and animate the model symbol. Save the VHDL-AMS file and exit the VHDL-AMS editor.

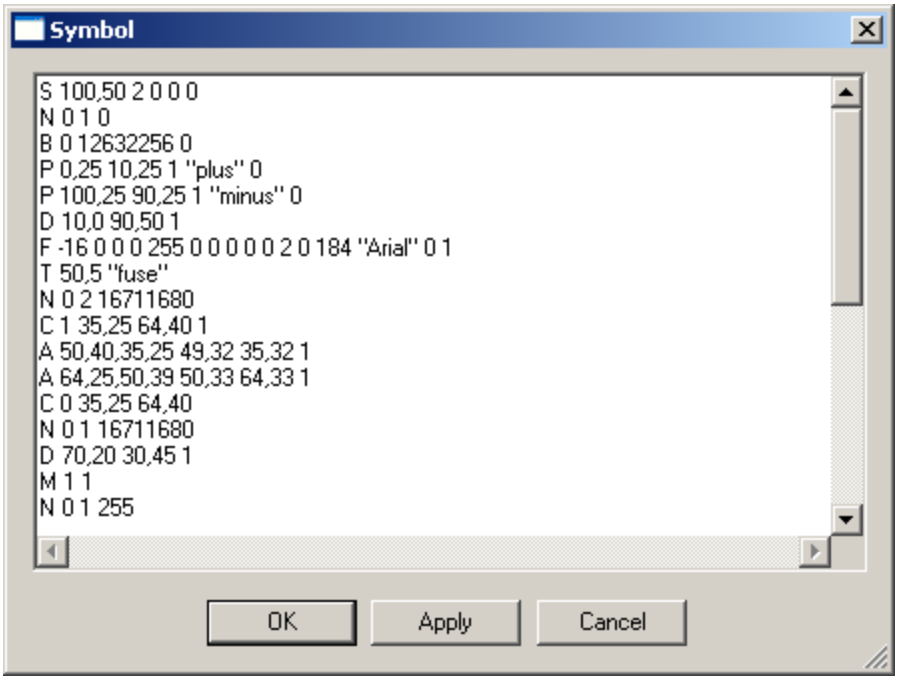
Import the Model into Schematic

If the case studies project is not open, open the *case_study_examples.ssc* project and create a new schematic in the project. In Schematic, choose **Sheet** > **Subsheet** > **New VHDL-AMS** and use the click and drag mouse action to create a new VHDL-AMS Text Subsheet. Within the subsheet, choose **«Import»** from the shortcut menu to browse to the new fuse model and import it into the subsheet. Select **«Level Up»** from the shortcut menu to return to the sheet.

Animating the Model Symbol

The method described here is a shortcut method that uses an existing symbol definition to animate a symbol. Typically, a symbol animation will be developed graphically in the Symbol Editor. See "Symbol Animation " on page 4-56 for more information on creating symbol animation graphics.

You can create an animated symbol for the fuse model with the contents of the *fuse_symbol.txt* file. Open the *fuse_symbol.txt* file in a text editor. In Schematic, right-click on the fuse model and select **«Edit Symbol»** to open the Symbol Editor. In the Symbol Editor, choose **Edit** > **Symbol Text** and replace the default text in the symbol text dialog box with the text from *fuse_symbol.txt*.



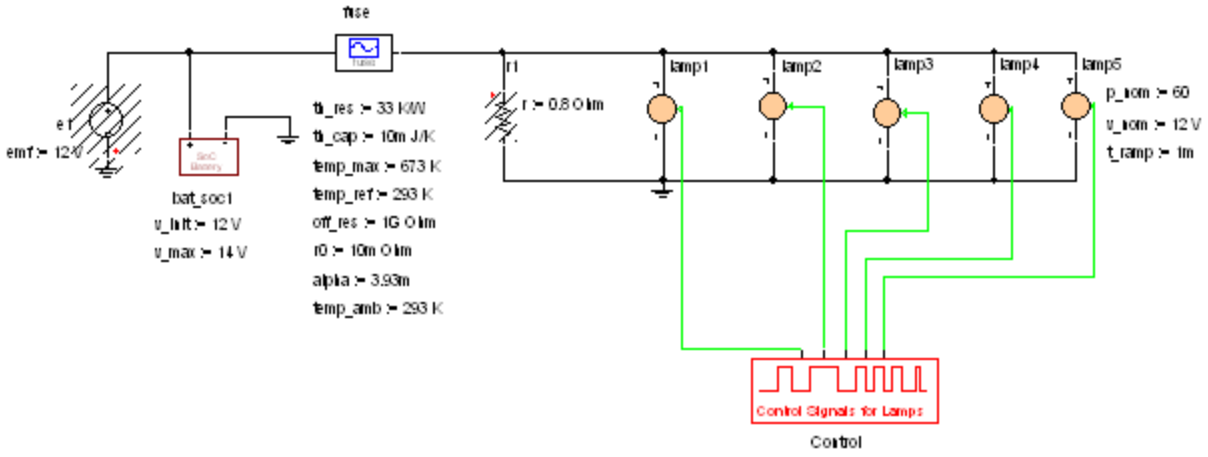
Click **OK** and exit the Symbol Editor. The animated symbol is now ready for use in the Schematic.

> **Note:**
>
> See "Symbol Animation " on page 4-56 to learn how to graphically create the symbol animation for a model.

Background: Fuse for Lamps in an Automotive Subsystem

The following example demonstrates the use of the fuse model in conjunction with several lamps in an automotive subsystem. The subsystem is powered by the SoC battery model and has five

lamp models. The lamp models are switched on sequentially using a digital control text subsheet model in VHDL-AMS. The sequential activation of the individual lamp models draws additional current from the battery model. The activation of the fifth lamp model blows the fuse and disconnects the lamps from the power system. For this example to run in the student version of Twin Builder, the *bat_soc1*, *lamp1*, *lamp2*, and *lamp3* models must be deactivated; *e1* and its associated ground pin, and *r1* must be activated.



Model: Digital Control Model

Entity Description - Digital Control Model

The digital control model has five output control signals of type REAL that will be used to switch each of the five lamps on and off. The equivalent VHDL-AMS description for the model interface is as follows:

```
ENTITY control IS
PORT( SIGNAL ctrl1 : OUT REAL := 0.0; --Control Signal 1
SIGNAL ctrl2 : OUT REAL := 0.0; --Control Signal 2
SIGNAL ctrl3 : OUT REAL := 0.0; --Control Signal 3
SIGNAL ctrl4 : OUT REAL := 0.0; --Control Signal 4
SIGNAL ctrl5 : OUT REAL := 0.0 --Control Signal 5
);
END ENTITY control;
```

Architecture Description - Digital Control Model

The model architecture describes the signal characteristics of the digital control model. It turns on each of the control signals at 10 second intervals using concurrent signal assignment statements. The signal assignments are delayed through the use of the **AFTER** keyword.

```
ARCHITECTURE arch_control OF control IS
BEGIN
 ctrl1 <= 1.0 AFTER 1ms;
 ctrl2 <= 1.0 AFTER 10ms;
 ctrl3 <= 1.0 AFTER 20ms;
 ctrl4 <= 1.0 AFTER 30ms;
 ctrl5 <= 1.0 AFTER 40ms;
END ARCHITECTURE admittance;
```

Model Parameters - Digital Control Model

The lamp models are parameterized to have a nominal power consumption of 60 Watts. The fuse model has the following parameters:

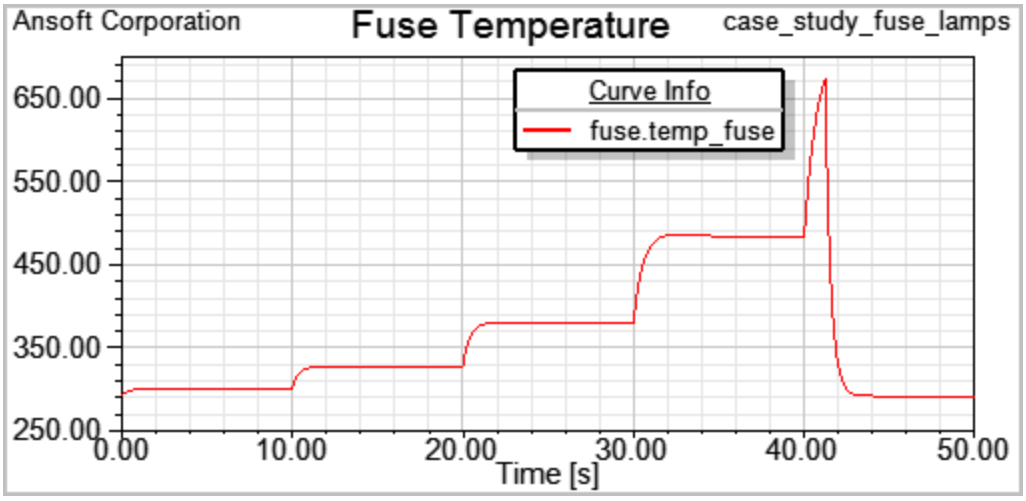
Name	Value	Unit	Evaluated Value
th_res	33	Kel_per_W	33Kel_per_W
th_cap	0.01	J_per_Kel	0.01J_per_Kel
temp_max	673	kel	673kel
temp_ref	293	kel	293kel
off_res	1000000000	ohm	1000000000ohm
r0	0.01	ohm	0.01ohm
alpha	0.00393		0.00393

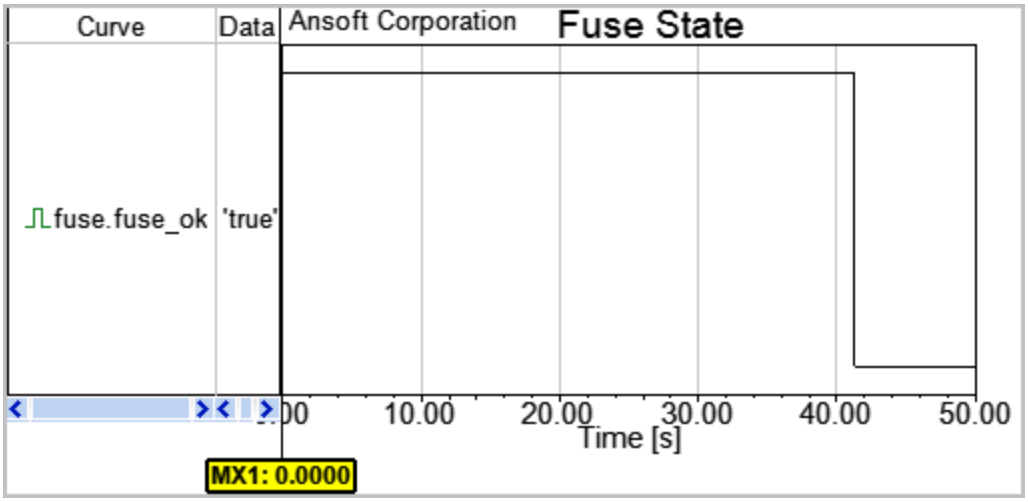
Most of the parameters of the fuse model are material-specific (such as temperature coefficient of resistance (alpha), maximum temperature (temp_max), resistivity (rho), and so on). Fuse model datasheets provide information about the fuse materials and blow-time characteristics. This data can be used with multirun simulations to determine appropriate values for thermal resistance and thermal capacitance.

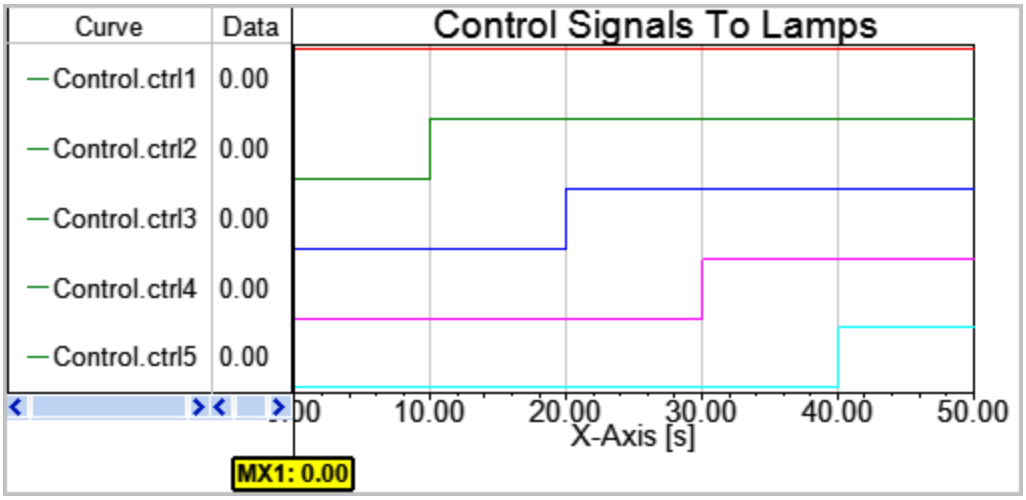
Simulation Parameters - Digital Control Model

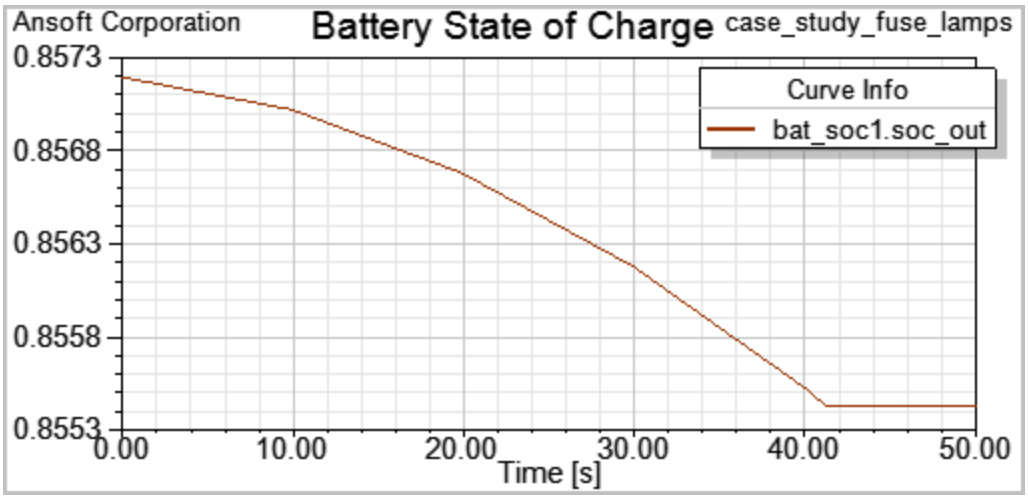
Select **Twin Builder** > **Edit Setup**, and change the default value for simulation End Time to *50* sec, Min Time Step to *100u* sec, and Max Time Step to *10m* sec. Click **OK** to apply the changes.

Results - Digital Control Model









The first plot illustrates the variation of the fuse temperature with time. As each lamp is turned on, the fuse temperature is found to increase and the switching on of the fifth lamp blows the fuse at approximately 40 sec. The second plot shows the fuse state value as an output. The third plot shows the digital control output signals to the lamps and illustrates the activation curves of the lamps. The fourth plot shows the State of Charge curve of the battery and illustrates the discharging of the battery as each lamp load is activated, and the “leveling off” after the fuse blows.

Detailed and Average Model Development: Claw-Pole Alternator

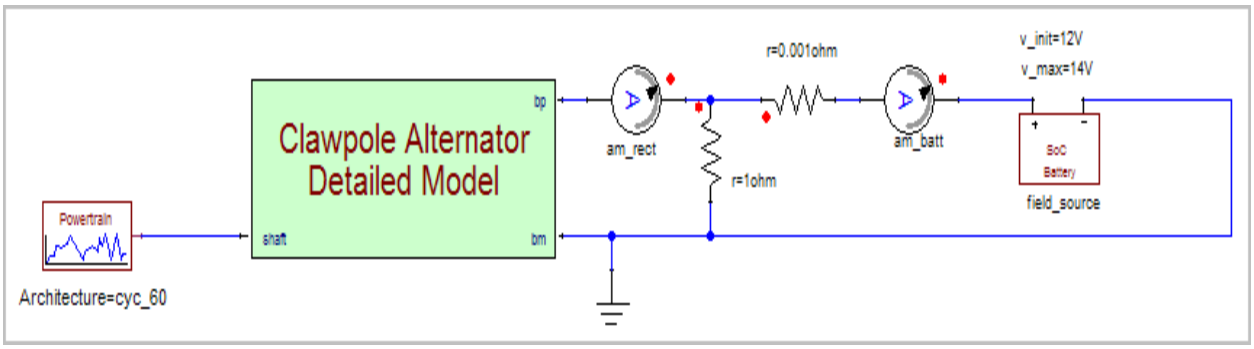
Concepts - Claw-Pole Alternator

This case study illustrates the modeling of detailed and averaged claw-pole alternator models, as discussed in Vahe Caliskan: *Modeling and Simulation of a Claw-Pole Alternator, Detailed and Averaged Models*[1].

This example emphasizes a key design choice to be made when simulating large circuits. A complicated three phase model with rectifier can be used to maintain accuracy, but at the expense of long simulation times. A simple averaged model can be used to decrease simulation time, with some sacrifice in accuracy. Both design methodologies, as presented in the publication, are described in this chapter.

A detailed model is used when the processes are highly transient, with spike currents, switching, and so on. The detailed claw-pole model contains a a rectifier bridge with six diodes to model the switching characteristics accurately. An averaged model is used to model energy exchange over extended periods of time, such as load balance simulations. The averaged claw-pole model contains simplified models of the armature and rectifier.

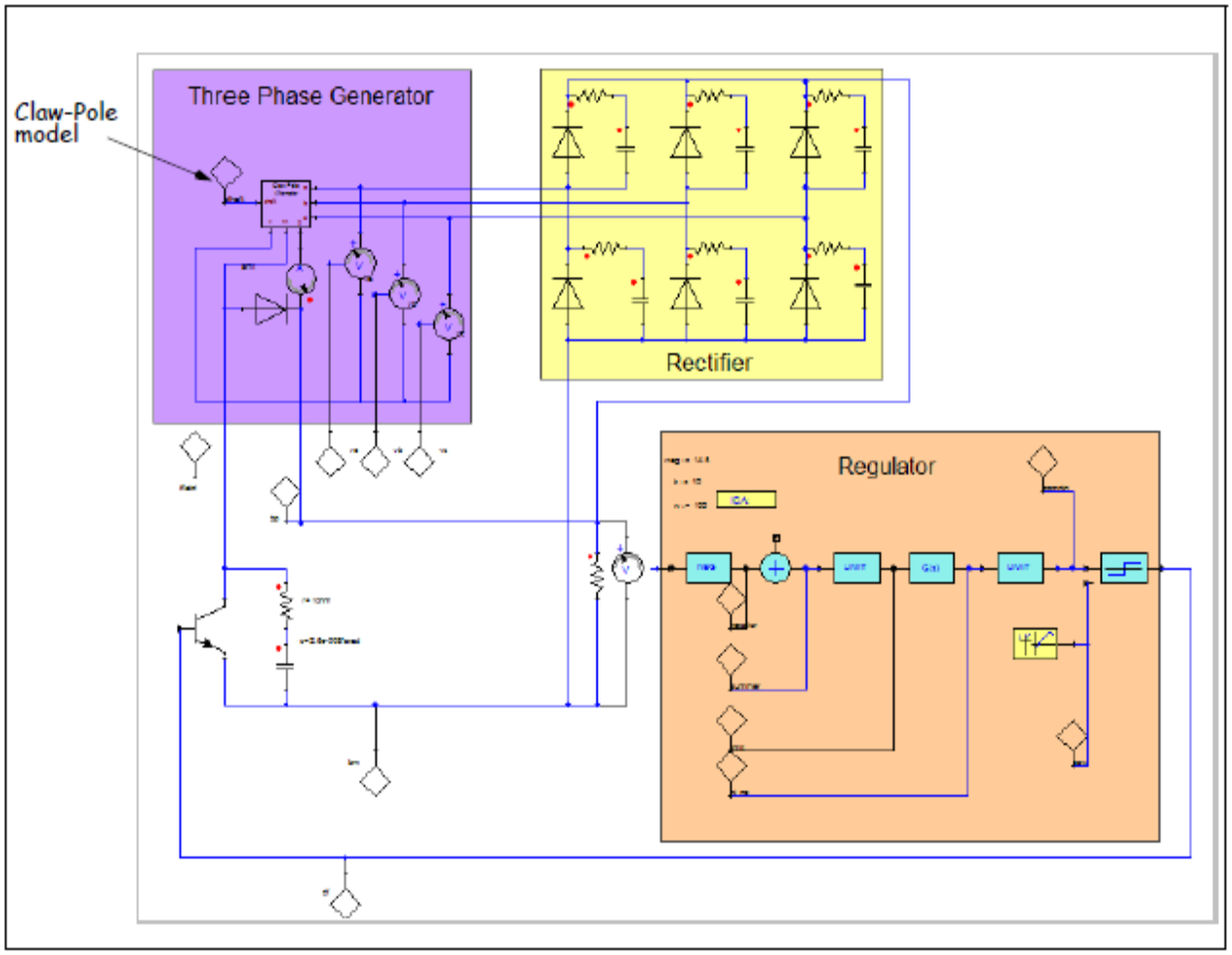
The two alternator design circuits are developed as graphical subsheet models, using VHDL-AMS library models and textual subsheets. The models are used in conjunction with the *SOC Battery* model and the *Powertrain* model.



1. Vahe Caliskan: *Modeling and Simulation of a Claw-Pole Alternator, Detailed and Averaged Models*, LEES Technical Report TR-00-009, Laboratory for Electromagnetic and Electronic Systems, Massachusetts Institute of Technology, Cambridge, MA, October 2000

Background: Detailed Model of a Claw-Pole Alternator

The detailed alternator model is separated into three modules: a three phase generator, a three-phase rectifier, and a switching voltage regulator. The following figure shows the graphical subsheet of the model:



The three-phase rectifier model uses diodes from the *Circuit* folder of the *Basic Elements Vhdlams* library. The control circuit uses block models from the *Blocks* folder in the same library. The *Claw-Pole Alternator* model, *cp_math*, is available from the **vhdlams_tutorial folder under System Libraries** in the **Component Libraries window**.

Model: Mathematical Claw-Pole Model

The parameters and terminals for the mathematical model are as follows:

Interface	Name	Property	Default Value	Description
GENERIC	rs	RESISTANCE	33m	Stator Winding Resistance [W]
	lls	INDUCTANCE	18u	Stator Leakage Inductance [H]
	lms	INDUCTANCE	180u	Stator Magnetizing Inductance [H]
	rr	RESISTANCE	3.44	Rotor Winding Resistance [W]
	llr	INDUCTANCE	200m	Rotor Leakage Inductance [H]
	lmr	INDUCTANCE	300m	Rotor Magnetizing Inductance [H]
	p	REAL	12	Number of Poles
	theta0	ANGLE	0	Initial Rotor Angle [rad]
	k	REAL	1	Coeff. Mag. Coupling Stator-Rotor
TERMINAL	a, b, c	ELECTRICAL		Phase a, b, c voltage of stator
	n	ELECTRICAL		Neutral terminal of alternator
	fp, fm	ELECTRICAL		Field positive/negative terminal
	shaft	ROTATIONAL_V		Rotor mechanical angular velocity connection

The equations provided for the generator model are as follows:

$$\frac{d\lambda_a(t)}{dt} = \upsilon_a(t) - R_s i_a(t)$$

$$\frac{d\lambda_b(t)}{dt} = \upsilon_b(t) - R_s i_a(t) \qquad \lambda_a(t) = L_s i_a(t) + L_{ss} i_b(t) + L_{ss} i_c(t) + L_{ar}(\theta_e) i_r(t)$$

$$\frac{d\lambda_c(t)}{dt} = \upsilon_c(t) - R_s i_b(t) \qquad \lambda_b(t) = L_{ss} i_a(t) + L_s i_b(t) + L_{ss} i_c(t) + L_{br}(\theta_e) i_r(t)$$

$$\frac{d\lambda_r(t)}{dt} = \upsilon_r(t) - R_r i_r(t) \qquad \lambda_c(t) = L_{ss} i_a(t) + L_{ss} i_b(t) + L_s i_c(t) + L_{cr}(\theta_e) i_r(t)$$

$$\frac{d\theta_r}{dt} = \omega_r \qquad \theta_e = \left(\frac{p}{2}\right)\theta_r \qquad \lambda_r(t) = L_{ra}(\theta_e) i_a(t) + L_{rb}(\theta_e) i_b(t) + L_{rc}(\theta_e) i_c(t) + L_r i_r(t)$$

$$L_{ar}(\theta_e) = L_{ra}(\theta_e) = M \cdot \cos(\theta_e)$$
$$L_{br}(\theta_e) = L_{rb}(\theta_e) = M \cdot \cos(\theta_e - \phi)$$
$$L_{cr}(\theta_e) = L_{rc}(\theta_e) = M \cdot \cos(\theta_e + \phi)$$

These equations describe an ideal three-phase generator, with windings distributed to generate perfectly sinusoidal voltages. But a real claw-pole alternator will generate voltages with

significant distortion. This effect can be accounted for in finite element models, or by incorporating high order harmonics in the inductance terms.[1]

M is the mutual inductance between stator coils and rotor, θ_r is the rotor angle, θ_e is the electrical angle, L_s is the stator self-inductance, and L_r is the stator self-inductance.

$$L_s = ll_s + l_{ms} \qquad L_{ss} = -\frac{1}{2} \cdot l_{ms}$$
$$L_r = ll_r + l_{mr} \qquad M = k \cdot \sqrt{l_{ms} \cdot l_{mr}}$$

The rectifier and field diodes all require RC snubbers to properly function during simulation. The values are set differently, according to the stator and rotor inductances. The voltage regulator will switch the field ON and OFF at approximately 140 Hz, so the snubber C is chosen for resonance at 1400 Hz. Then the snubber R is chosen for critical clamping.

Stator:

$$L_s = ll_s + l_{ms} = 18\mu H + 180\mu H = 198\mu H$$
$$C_{sns} = \frac{1}{L_s \cdot \omega^2} = \frac{1}{198\mu H \cdot (2 \cdot \pi \cdot 1400Hz)^2} = 65\mu F$$
$$R_{sns} = \sqrt{\frac{L_s}{C_{sns}}} = \sqrt{\frac{198}{65}} = 1.7\Omega$$

Rotor:

$$L_r = ll_r + l_{mr} = 200mH + 300mH = 500mH$$
$$C_{snr} = \frac{1}{L_r \cdot \omega^2} = \frac{1}{500mH \cdot (2 \cdot \pi \cdot 1400Hz)^2} = 26nF$$
$$R_{snr} = \sqrt{\frac{L_r}{C_{snr}}} = \sqrt{\frac{500m}{26n}} = 4.4k\Omega$$

The detailed model of the claw-pole can be developed as a VHDL-AMS model with the following description:

```
LIBRARY IEEE;

USE IEEE.ELECTRICAL_SYSTEMS.ALL;
```

```
USE IEEE.MECHANICAL_SYSTEMS.ALL;
USE IEEE.MATH_REAL.ALL;
ENTITY cp_math IS
 GENERIC(
 rs : RESISTANCE := 33.0e-3; -- Stator winding resistance
 lls : INDUCTANCE := 18.0e-6; -- Stator leakage inductance
 lms : INDUCTANCE := 180.0e-6; -- Stator magnetizing inductance
 rr : RESISTANCE := 3.44; -- Rotor winding resistance
 llr : INDUCTANCE := 200.0e-3; -- Rotor leakage inductance
 lmr : INDUCTANCE := 300.0e-3; -- Rotor magnetizing inductance
 p : REAL := 12.0; -- Number of poles
 theta0 : ANGLE := 0.0; -- Initial rotor angle
 k : REAL := 1.0); -- Coefficient of magnetic coupling stator-rotor
 PORT(
 TERMINAL a,b,c : ELECTRICAL;
 TERMINAL n,fp,fm : ELECTRICAL;
 TERMINAL shaft : ROTATIONAL_V);
END ENTITY cp_math;
```

```
ARCHITECTURE behav OF cp_math IS
 CONSTANT p2 : REAL := p/2.0;
 CONSTANT phase : REAL := MATH_2_PI/3.0; -- phase shift among stator coils
 CONSTANT ls : INDUCTANCE := lls + lms;
 CONSTANT lr : INDUCTANCE := llr + lmr;
 CONSTANT lss : INDUCTANCE := -0.5*lms;
 CONSTANT m : REAL := k * SQRT(lmr*lms);
 QUANTITY va ACROSS ia THROUGH a TO n; -- four electrical coils
 QUANTITY vb ACROSS ib THROUGH b TO n;
 QUANTITY vc ACROSS ic THROUGH c TO n;
```

```
QUANTITY vr ACROSS ir THROUGH fp TO fm;
QUANTITY omega_r ACROSS shaft TO ROTATIONAL_V_REF; --one mechanical shaft
QUANTITY theta_r,theta_e : ANGLE := 0.0;
QUANTITY la,lb,lc : INDUCTANCE := 0.0;
QUANTITY lambda_a,lambda_b,lambda_c,lambda_r : FLUX := 0.0;
BEGIN
omega_r == theta_r'DOT;
theta_e == theta_r * p2;
la == m*cos(theta_e); -- phase to rotor inductances
lb == m*cos(theta_e - phase);
lc == m*cos(theta_e + phase);
lambda_a == ls*ia + lss*ib + lss*ic + la*ir; -- flux linkage
lambda_b == lss*ia + ls*ib + lss*ic + lb*ir;
lambda_c == lss*ia + lss*ib + ls*ic + lc*ir;
lambda_r == la*ia + lb*ib + lc*ic + lr*ir;
va == rs*ia + lambda_a'DOT; -- voltage from faraday's law
vb == rs*ib + lambda_b'DOT;
vc == rs*ic + lambda_c'DOT;
vr == rr*ir + lambda_r'DOT;
END ARCHITECTURE behav;
```

First, the flux linkage is defined in terms of time-varying inductances and currents. Then, the four coil voltage equations are written using Faraday's Law.

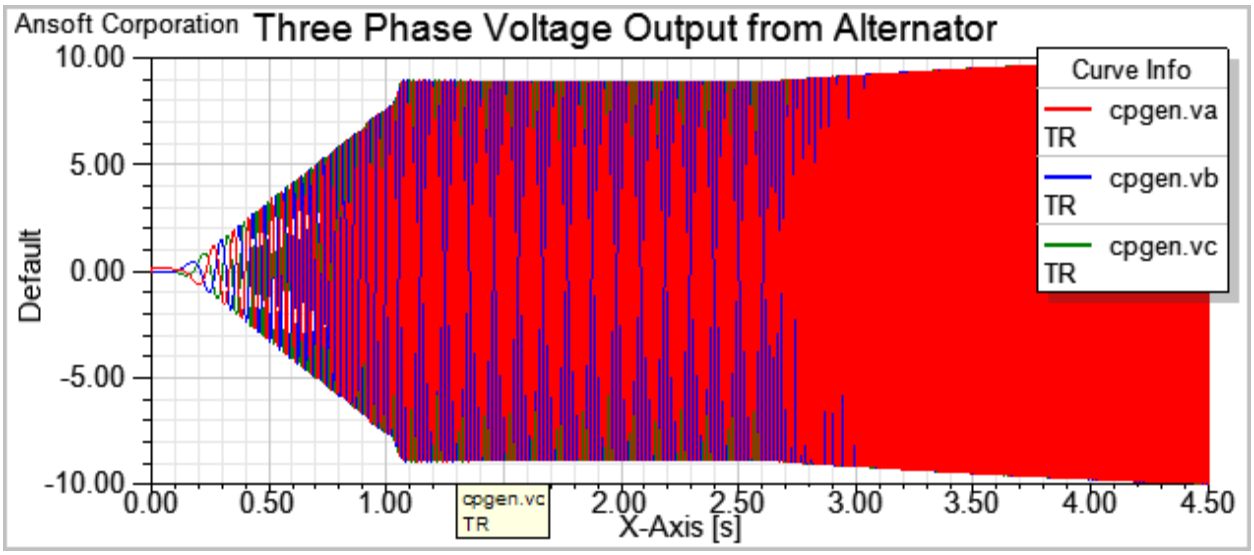
The outputs include coil flux linkage, phase inductances, rotor angle, and electrical angle. **CONSTANT** definitions are used to pre-process the inductance parameters.

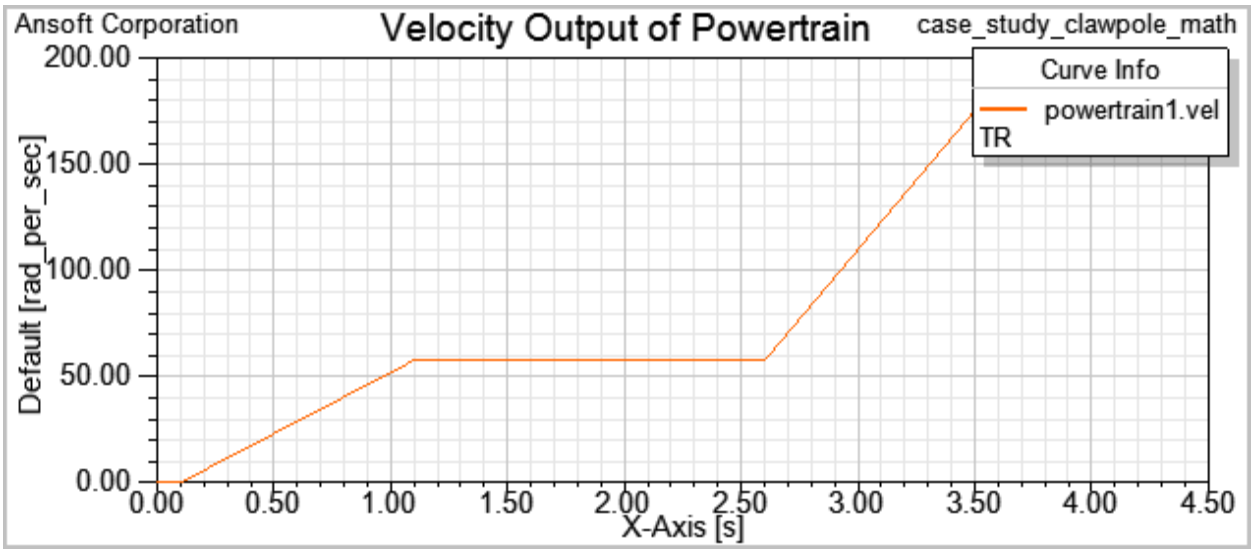
This detailed model of the claw-pole requires small time steps and a long simulation time to accurately simulate the switching characteristics (HMIN=50μs, HMAX=5ms, TEND=4.5s).

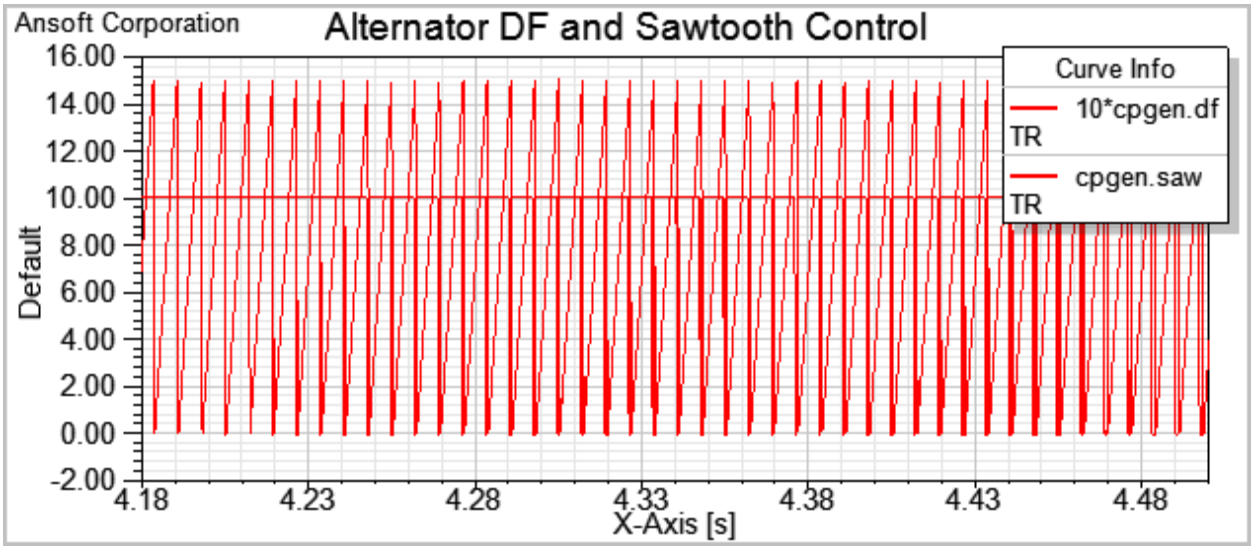
1. Bai, H.; Pekarek, S.; Techenor, J.; Eversman, W.; Buening D.; Holbrook, G.; Hull, M.; Krefta, R.; Shields, S.: *Analytical Derivation of a Coupled-Circuit Model of a Claw-Pole Alternator with Concentrated Stator Winding*, IEEE Transaction on Energy Conversion, March 2002, pp 32-38

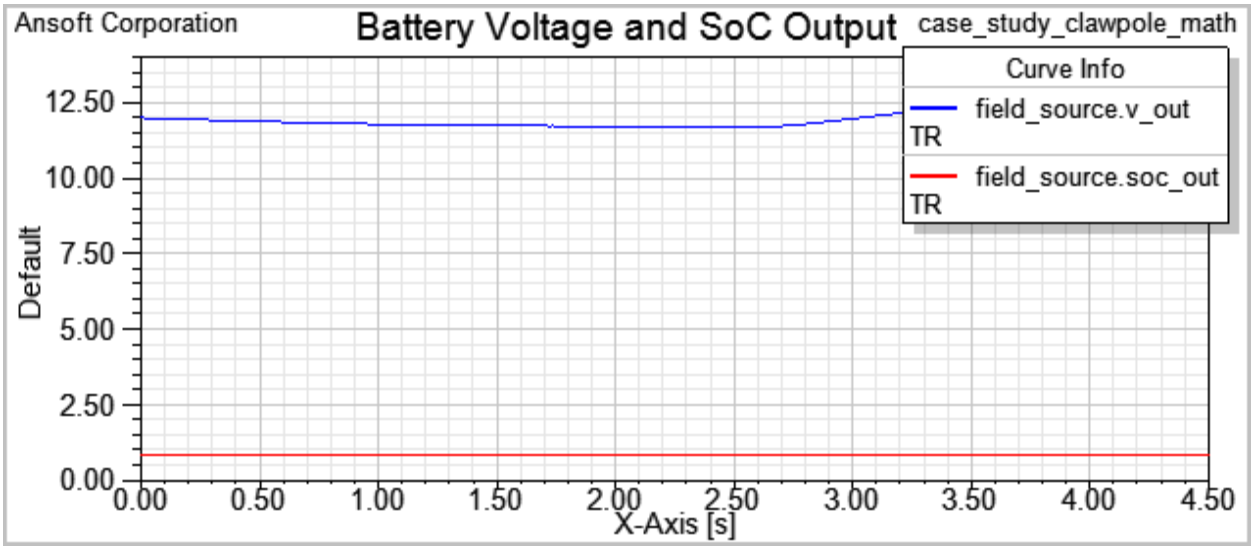
Results - Mathematical Claw-pole Model

The first Display Element shows the three-phase output voltages from the mathematical claw-pole model. The *Powertrain* model's velocity output is plotted in the second graph. The voltage and frequency increase linearly with the powertrain velocity. The field switching curve is displayed as *cpgen.df*, and the battery output is displayed in the fourth graph.





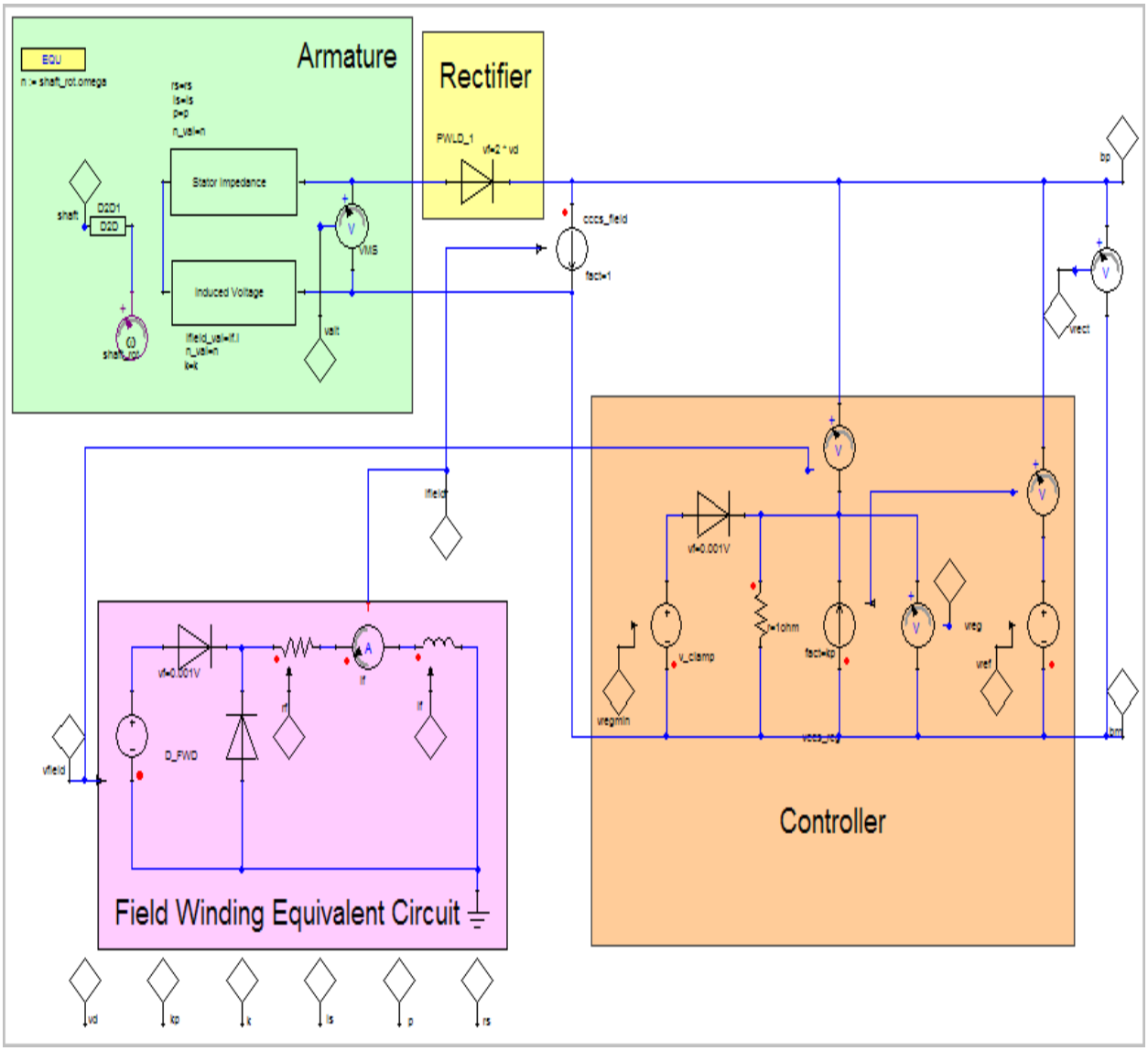




Background: Averaged Model of Claw-Pole Alternator

In this averaged model, the three rectified phases are simplified to an equivalent DC machine. The green portion is the equivalent armature, representing the speed voltage generated in the stator. The blue portion contains the field circuit, and the orange portion contains the voltage regulator. The yellow portion represents the rectifier voltage drop, and prevents motor operation of the alternator. This model is better suited for systems where transient characteristics are not important.

The following figure shows the design of the averaged alternator model:



Claw-Pole Alternator Model

Speed-Voltage Model

The averaged claw-pole alternator uses a speed-voltage model that is modeled as a current source. It provides a terminal voltage value that is proportional to the alternator speed and field current, according to the equation: $v = k \cdot n \cdot i$

This is the same equation describing a separately excited DC machine, where *i* is the field current.

The speed-voltage model is developed as a VHDL-AMS text subsheet. The entity and architecture description are as follows:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
USE IEEE.MATH_REAL.ALL;
ENTITY speedvolt IS
 GENERIC( k : REAL := 1.0);
 PORT( QUANTITY n_val : REAL := 0.0;
 QUANTITY ifield_val : CURRENT := 0.0;
 TERMINAL pos,neg : ELECTRICAL);
END ENTITY speedvolt;
ARCHITECTURE behav OF speedvolt IS
 QUANTITY v ACROSS i THROUGH pos TO neg;
BEGIN
 v == k*n_val*ifield_val;
END ARCHITECTURE behav;
```

Stator-Impedance Model

The stator impedance in the averaged model of the claw-pole alternator uses a speed-dependent resistive drop given by the following equation:

$$Z(n) = \sqrt{R_s^2 + \left(\frac{\pi}{30} \cdot \left(\frac{p}{2}\right) \cdot nL_s\right)^2}$$

The model accepts the stator resistance, stator inductance, number of poles, and alternator speed as inputs. It provides a voltage/current output according to the impedance equation.

Z(n) is the magnitude of the equivalent stator impedance. The inductive part of this impedance increases with frequency, or equivalently, with rotor speed n. When R_s is greater than zero, Z(n) is always greater than zero.

The stator-impedance is also modeled as a VHDL-AMS text subsheet. The entity and architecture description are as follows:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
USE IEEE.MATH_REAL.ALL;
ENTITY statorz IS
 GENERIC( rs : RESISTANCE := 33.0e-3;
 ls : INDUCTANCE := 177.0e-6;
 p : REAL := 12.0);
 PORT( QUANTITY n_val : REAL := 0.0;
 TERMINAL pos,neg : ELECTRICAL);
END ENTITY statorz;
ARCHITECTURE behav OF statorz IS
 QUANTITY v ACROSS i THROUGH pos TO neg;
 QUANTITY z : REAL := 0.0;
BEGIN
 z == sqrt(rs**2.0 + ((MATH_PI/30.0)*(p/2.0)* n_val*ls)**2.0);
 i == v/z;
END ARCHITECTURE behav;
```

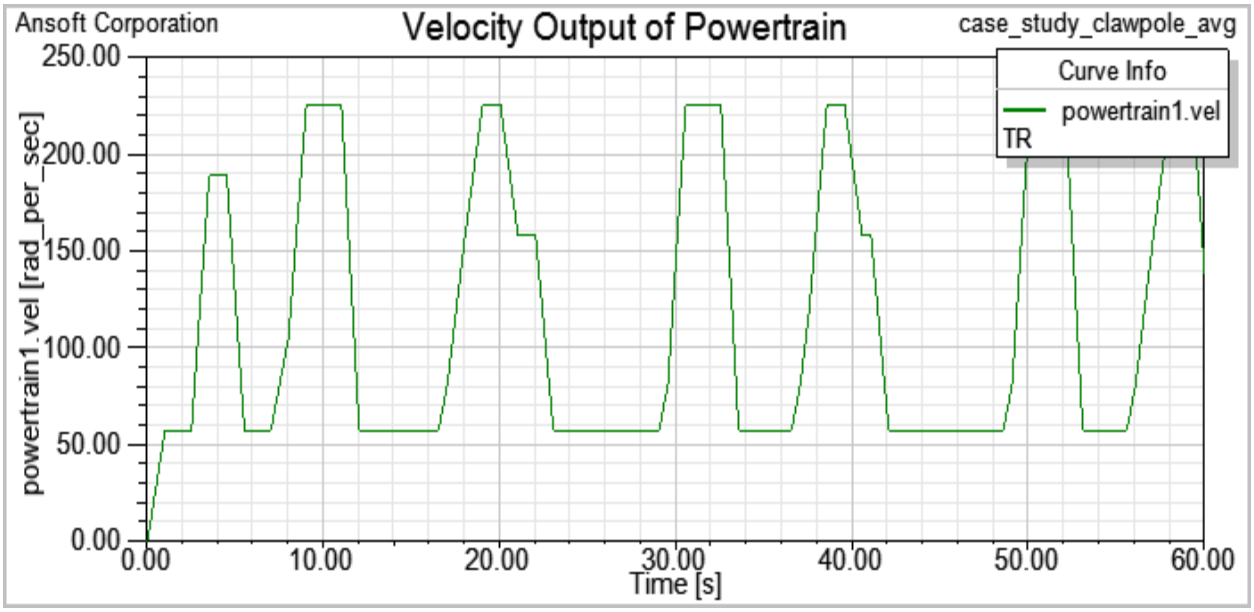
The *dc* voltage drop in the three-phase rectifier circuit is accounted for by providing an equivalent diode with voltage drop of 2*Vd*, where *Vd* is the forward drop of a single diode in the actual rectifier.

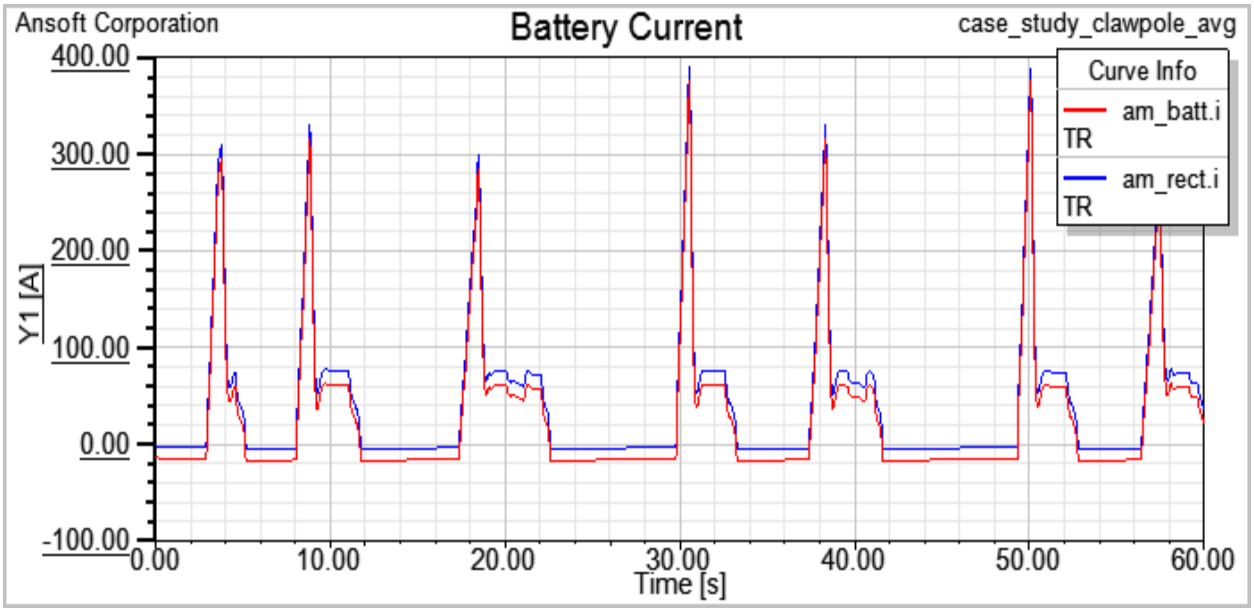
The other models in the subsheet are as shown in the figure at the beginning of this section, and are obtained from the *Basic Elements VHDLAMS* library. This circuit requires far less time for simulation than the detailed claw-pole alternator (**HMIN**=10µs, **HMAX**=1ms, **TEND**=60s).

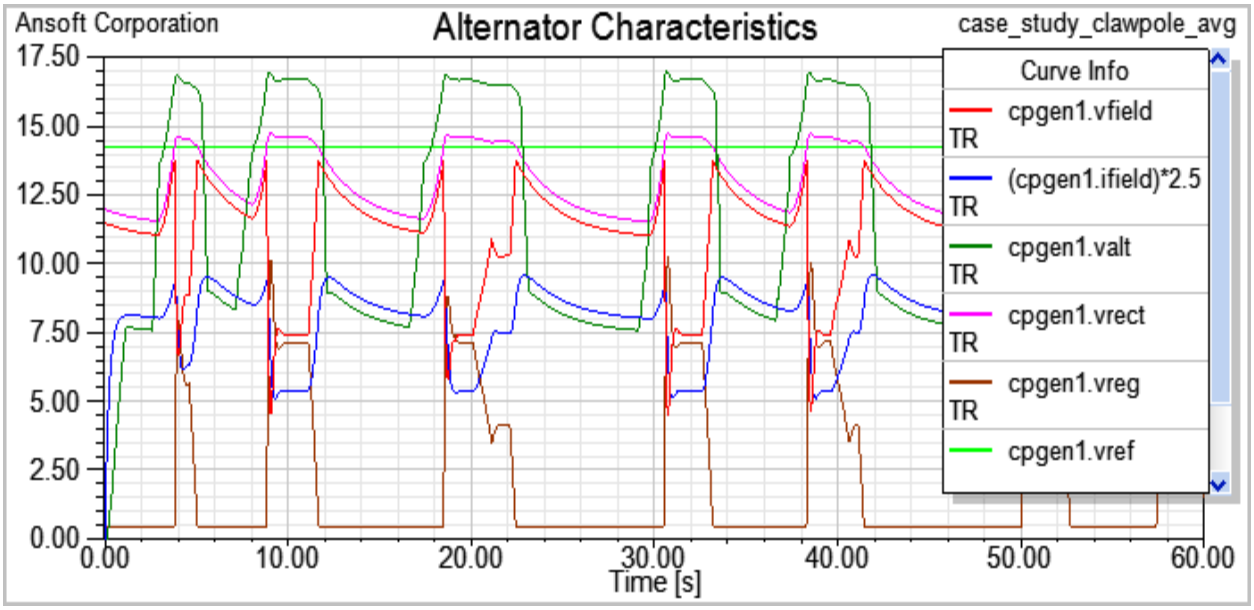
Results - Claw-Pole Alternator

The graph outputs show the variation of the battery current, voltage and state of charge with respect to the velocity of the powertrain model. The battery voltage follows the speed characteristics of the powertrain model. The first Display Element shows the velocity output from

the *Powertrain* model. The battery current and the rectified current are plotted in the second graph. The alternator outputs, are displayed in the third graph.







Multilevel Modeling Techniques: Loads

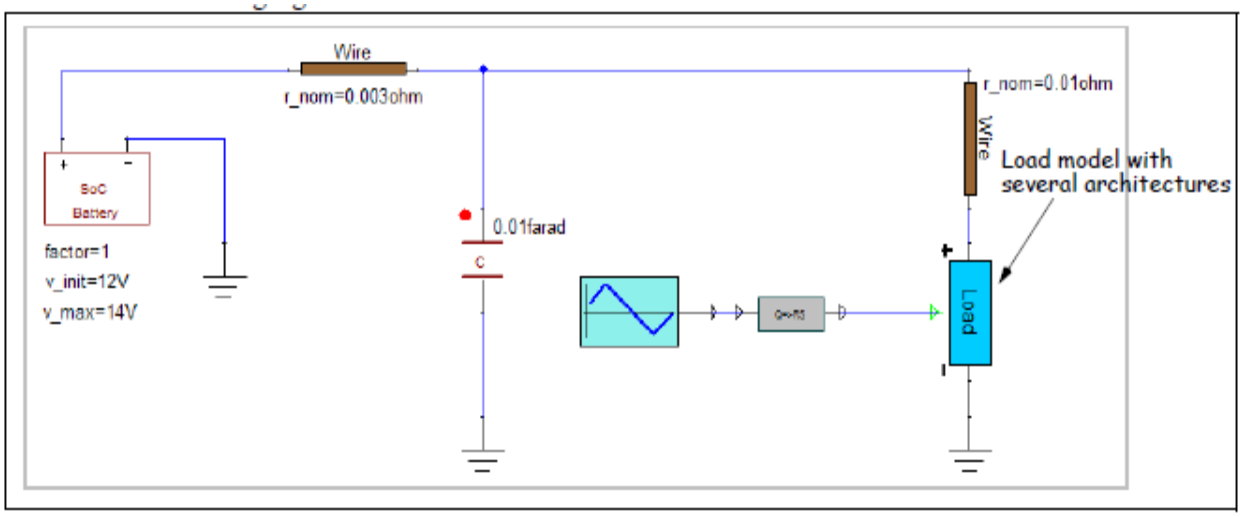
Concepts - Multilevel Modeling

This case study uses a load model to illustrate the concept of having multiple architectures with different behaviors defined within an individual model. It explains three different architectures that can be used to characterize a load used in the powernet system. In addition, the concept of component instantiation is described, using a lamp model to show how a particular load behavior can be reused in another model.

Background - Multilevel Modeling

The powernet system supports three different types of loads, each load being different only in its behavior and not its interface. The first type of load uses an admittance load behavior where the current in the model varies according to the control quantity in an ohmic manner. The second type of load uses a nominal load behavior, which is described with a nominal power and voltage specification. The third load behavior is for switched models where the control quantity switches a nominal load ON and OFF. This switched load behavior is then reused in a lamp model to illustrate component instantiation.

The following figure shows the basic circuit used for the load model with several architectures:



Hint	If the pins of two different data types are connected directly, a flexible OmniCaster will automatically be inserted between them. The flexible OmniCaster model selects the correct transformation based on the data types of the pins connected to it.

Model: Load

Entity Description - Load

The entity description indicates that three parameters are accepted by the load model: nominal power *p_nom* in Watts, nominal voltage *v_nom* in Volts, and ramp time *t_ramp* in seconds. It is not necessary to use all parameters specified in the **GENERIC** declaration within each architecture of the model.

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY load IS
GENERIC(
 p_nom: REAL := 1.0;
 v_nom: VOLTAGE := 14.0;
 t_ramp: REAL := 1.0e-5);
PORT(
TERMINAL p,m : ELECTRICAL;
```

```
SIGNAL on_ctrl : IN REAL := 0.0);
END ENTITY load;
```

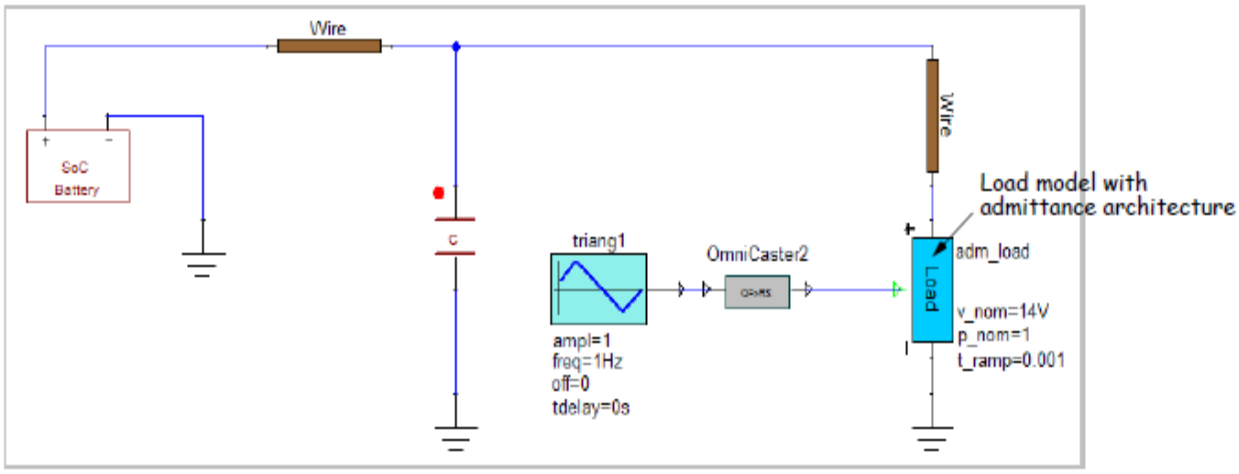
The **ENTITY** description also defines two electrical ports, *p* and *m*, for the load model. This declaration requires the use of the *ieee.electrical_systems* package. A signal port for the control signal *on_ctrl* of type REAL is also defined.

Architecture Description: Admittance Load

The *admittance* architecture models the current as proportionally dependent on the control signal value. The control signal value will be the admittance y.

$$i = y \cdot v$$

This architecture uses the *t_ramp* parameter, but not *v_nom* or *p_nom*.



This architecture first transforms the input *on_ctrl* (*REAL***SIGNAL** value) to *ctrl_qty* (REAL **QUANTITY** value) using the '*RAMP* attribute as follows:

```
ctrl_qty == on_ctrl'RAMP(0.0,0.0);
```

The '*RAMP* attribute transforms a signal to a quantity which follows the corresponding value of the signal with the delay of the specified rise time and fall time. If these parameters are 0.0, then the value of the quantity follows that of the signal instantaneously. If the value of any parameter

is greater than 0.0, the corresponding value change is linear from the current value of the signal to its new value, whenever that signal has an event.

The architecture then introduces a PT1 behavior on the control quantity to follow a first-order low-pass response and generates a new control quantity *ctrl_ramp*, according to the specified ramp time *t_ramp*. The following equation is used:

```
ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
```

If the *ctrl_ramp* quantity is positive, it is used as the admittance value for the load model according to the following equation:

```
IF on_ctrl <= 0.0 USE
 i == 0.0;
ELSE
 i == v * ctrl_ramp;
END USE;
```

Since the '*RAMP* attribute forces a synchronization between the analog and digital simulators, similar to the **BREAK** statement, a separate **BREAK** statement is not necessary for the *on_ctrl* signal.

The complete architecture description is as follows:

```
ARCHITECTURE admittance OF load IS
 QUANTITY v ACROSS i THROUGH p TO m;
 QUANTITY ctrl_ramp: REAL := 0.0;
 QUANTITY ctrl_qty: REAL := 0.0;
BEGIN
 ctrl_qty == on_ctrl'RAMP(0.0,0.0);
 ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
 IF on_ctrl <= 0.0 USE
 i == 0.0;
```

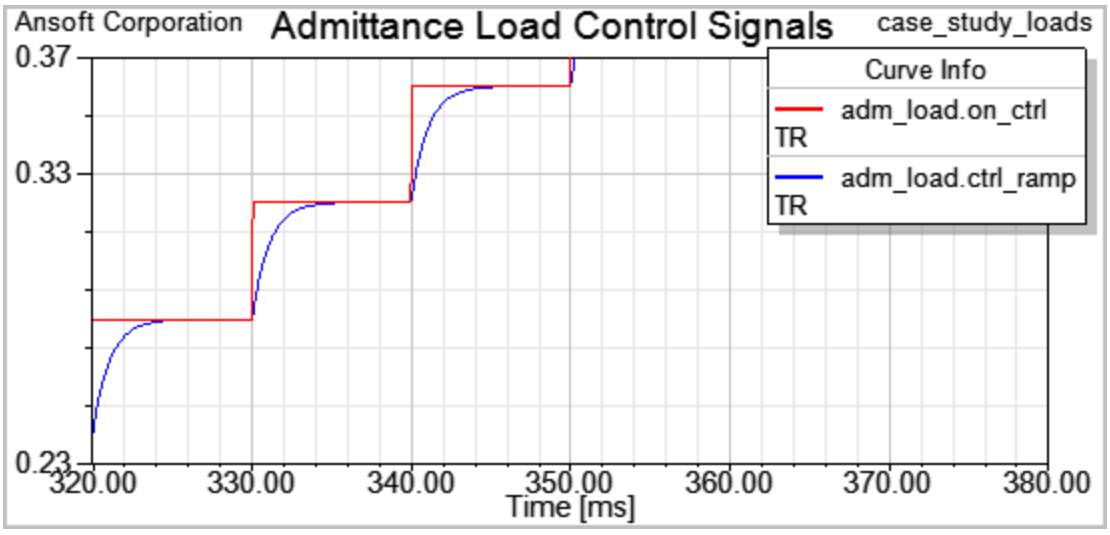
```
ELSE
i == v * ctrl_ramp;
END USE;
END ARCHITECTURE admittance;
```

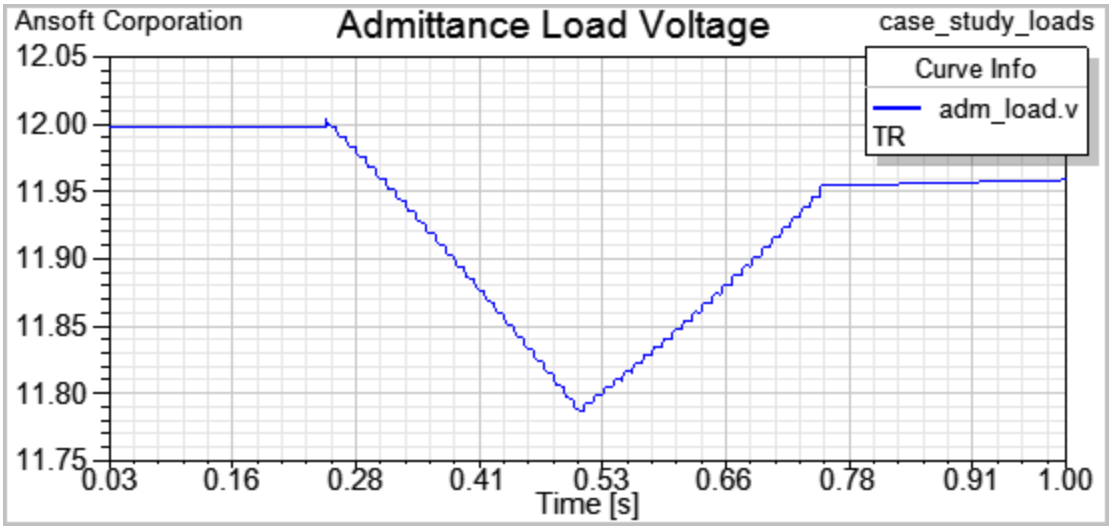
The *ctrl_ramp* and *ctrl_qty* are declared as free quantities within the admittance architecture.

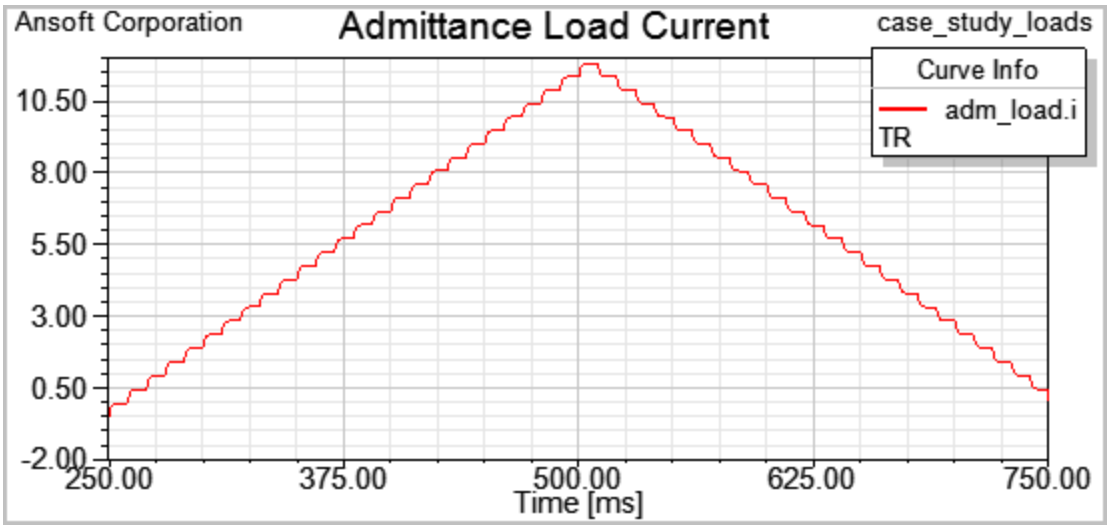
The input control admittance is provided by a triangular source time function, An OmniCaster model transforms the REAL **QUANTITY** output of the time function to a REAL **SIGNAL** value input for the load model. See "Using Transformation Models" on page 3-10 also.

Results - Admittance Load

The first plot shows the PT1 behavior of the transformed control signal. The second and third plots show the variation of the voltage and current of the admittance load. The current peaks at 12 amperes when the admittance is 1 siemens. But there is a small voltage drop at peak load. The voltage does not recover completely when the load is removed, because the battery has discharged slightly.



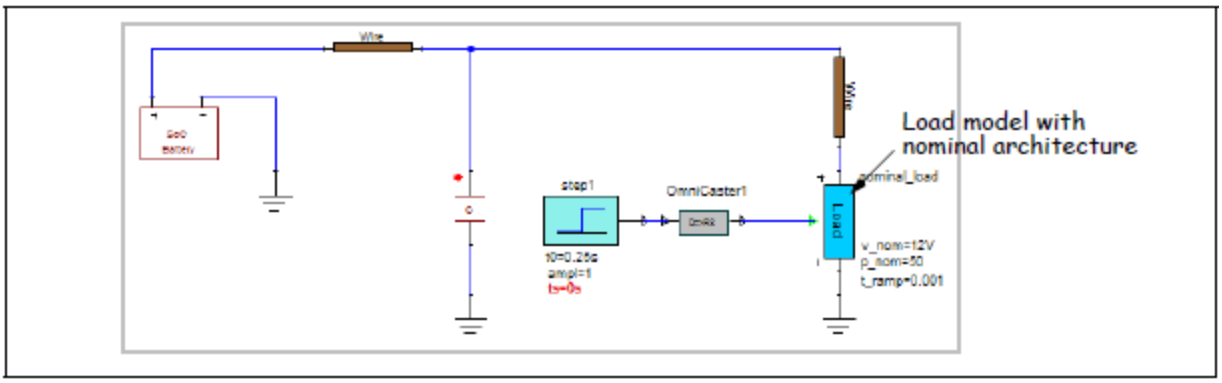




Architecture Description: Nominal Load

The *nominal* architecture of the *load* entity models an ohmic load with a nominal power and voltage rating, the current being scaled by the control signal. This architecture uses all three declared parameters, nominal voltage (*v_nom*), nominal power (*p_nom*) and ramp time (*t_ramp*).

The *Step* source block provides the control signal input and an *OmniCaster* model is used to provide a *REAL* signal input to the load model. The *p_nom* value used in this example is 50W, and the *v_nom* value is 12V, corresponding to the system bus voltage. The nominal resistance is then 2.88Ω.



Similar to the *admittance* architecture, this architecture first transforms the input *on_ctrl* (REAL **SIGNAL**) to *ctrl_qty* (REAL **QUANTITY**) using the '*RAMP* attribute, as follows:

```
ctrl_qty == on_ctrl'RAMP(0.0,0.0);
```

The architecture then introduces a PT1 behavior on the control quantity to follow a first-order low-pass response and generates a new control quantity *ctrl_ramp* according to the specified ramp time. The following equation is used:

```
ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
```

The *nominal* architecture differs from the *admittance* architecture in that for a positive value of *on_ctrl*, the current is modeled in terms of nominal resistance, as follows:

```
IF (on_ctrl <= 0.0) USE
 i == 0.0;
ELSE
 i == (v/v_nom) * ctrl_ramp* (p_nom/v_nom);
END USE;
```

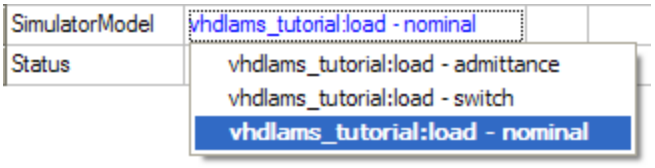
Since the '*RAMP* attribute forces a synchronization between the analog and digital simulators similar to the **BREAK** statement, a separate **BREAK** statement is not necessary for the *on_ctrl* signal.

The complete architecture description is as follows:

```
ARCHITECTURE nominal OF load IS
 QUANTITY v ACROSS i THROUGH p TO m;
 QUANTITY ctrl_ramp: REAL := 0.0;
 QUANTITY ctrl_qty: REAL := 0.0;
BEGIN
 ctrl_qty == on_ctrl'RAMP(0.0,0.0);
 ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
 IF (on_ctrl <= 0.0) USE
 i == 0.0;
 ELSE
 i == (v/v_nom) * ctrl_ramp* (p_nom/v_nom);
 END USE;
END ARCHITECTURE nominal;
```

The *ctrl_ramp* and *ctrl_qty* are declared as free quantities within the *nominal* architecture.

To change the architecture used on the sheet, double-click the *Load* symbol to open the Properties dialog box, and click the **Parameter Values** tab. Choose the **load-nominal** architecture from the **SimulatorModel** Value list. Click **OK** to apply the changes.

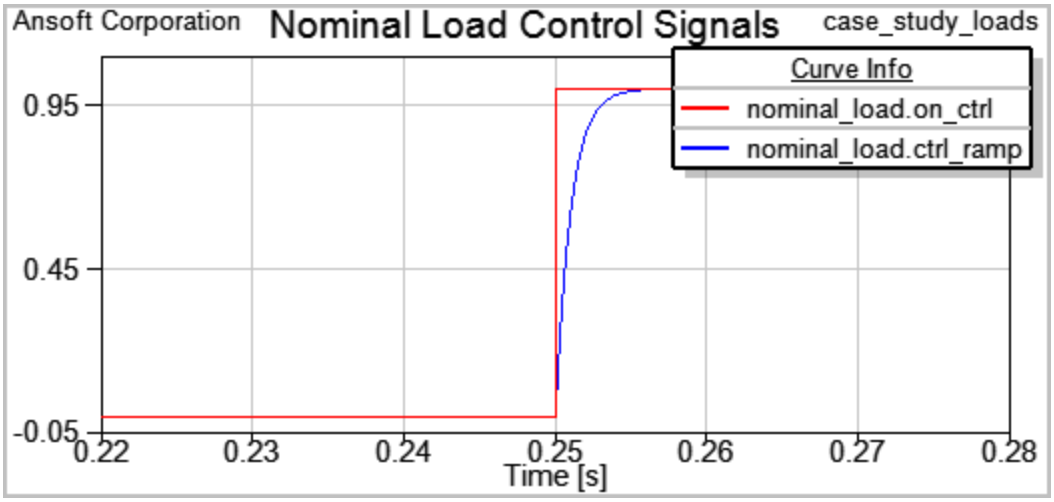


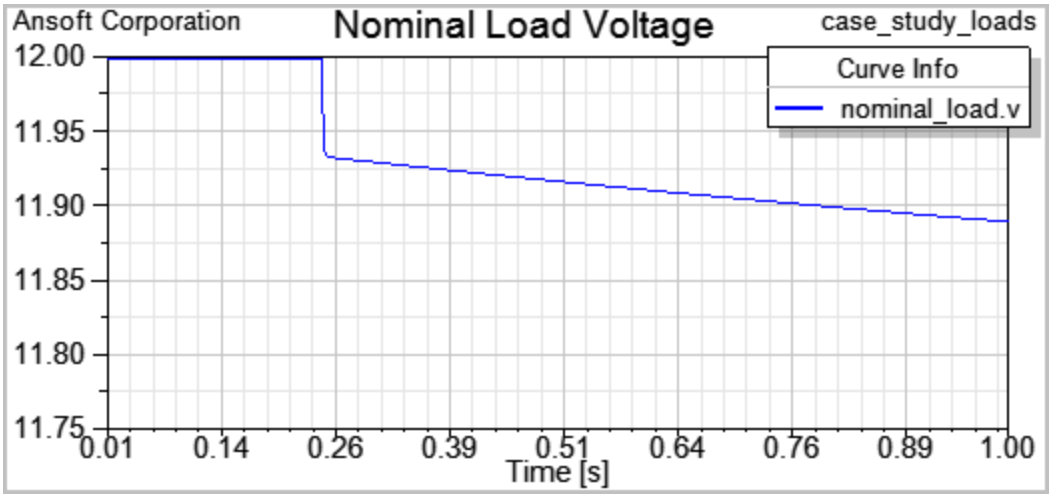
Note:

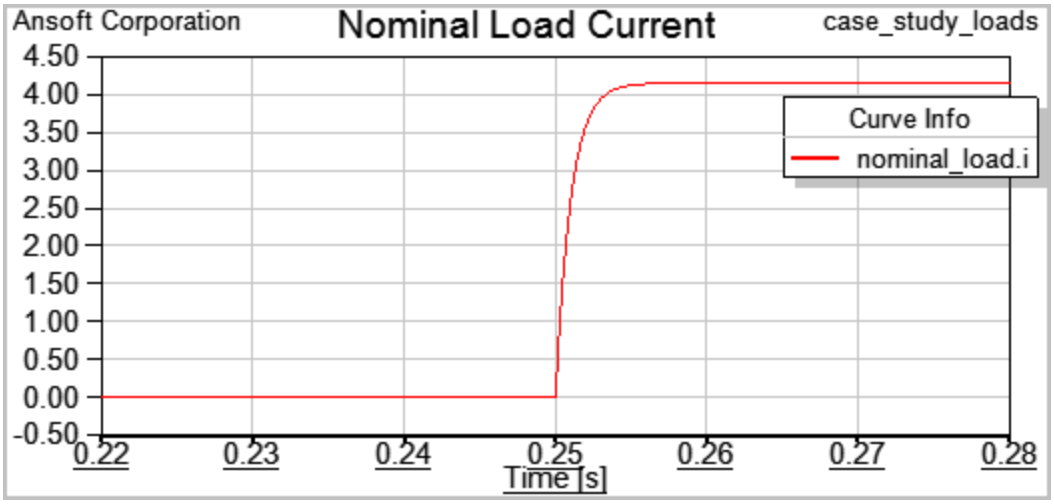
When a VHDL-AMS model from a library is first placed on the sheet, the default architecture is selected.

Results - Nominal Load

The first plot illustrates the PT1 behavior of the transformed control signal. The second and third plots show the variation of the current and voltage of the power load. Because of the voltage drop, actual power is less than the nominal 50W.



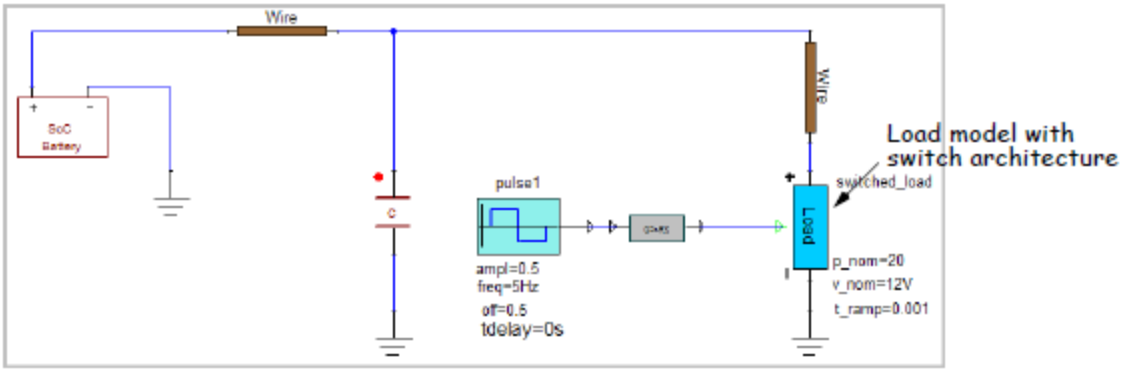




Architecture Description: Switched Load

The *switch* architecture models the on-off switch control for an ohmic load with a nominal power and voltage rating, the load current being further scaled by the control signal value.

The Pulse time function provides the control signal input, and an OmniCaster model is used to provide a REAL signal input to the load model. The *p_nom* value is 20W, and the *v_nom* value is 12V, corresponding to the system bus voltage. This represents a constant resistance *r_nom* of 7.2Ω.



The *switch* architecture does not transform the *on_ctrl* signal input to *ctrl_qty* quantity output using the '*RAMP* attribute as the previous two architectures do. Instead, it uses an **IF-USE** statement to specify an ON/OFF condition of 0/1, as shown in the following code snippet:

```
IF on_ctrl <=0.5 USE
```

```
 ctrl_qty == 0.0;
ELSE
 ctrl_qty == 1.0;
END USE;
BREAK ON on_ctrl;
```

To synchronize between the analog and digital statements, a separate **BREAK** statement must be introduced.

The architecture then introduces a PT1 behavior on the control quantity to follow a first order low pass response and generates a new control quantity called *ctrl_ramp* according to the specified ramp time. The following equation is used:

```
ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
```

The *switch* architecture is similar to the *nominal* architecture in that for a positive value of *on_ctrl*, the current is modeled in terms of nominal resistance, as follows:

```
IF (on_ctrl <= 0.0) USE
 i == 0.0;
ELSE
 i == (v/v_nom) * ctrl_ramp* (p_nom/v_nom);
END USE;
```

The complete architecture description is as follows:

```
ARCHITECTURE switch OF load IS
 QUANTITY v ACROSS i THROUGH p TO m;
 QUANTITY ctrl_ramp: REAL := 0.0;
 QUANTITY ctrl_qty: REAL := 0.0;
BEGIN
```

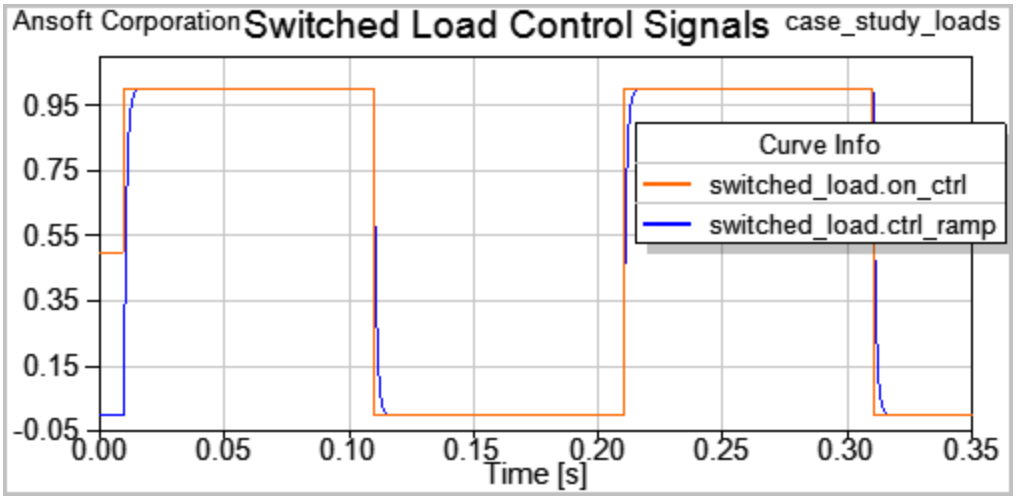
```
IF on_ctrl <=0.5 USE
ctrl_qty == 0.0;
ELSE
ctrl_qty == 1.0;
END USE;
ctrl_ramp'DOT == (1.0/t_ramp) * (ctrl_qty - ctrl_ramp);
IF ctrl_ramp <= 0.0 USE
i == 0.0;
ELSE
i == (v/v_nom) * ctrl_ramp* (p_nom/v_nom);
END USE;
BREAK ON on_ctrl;
END ARCHITECTURE switch;
```

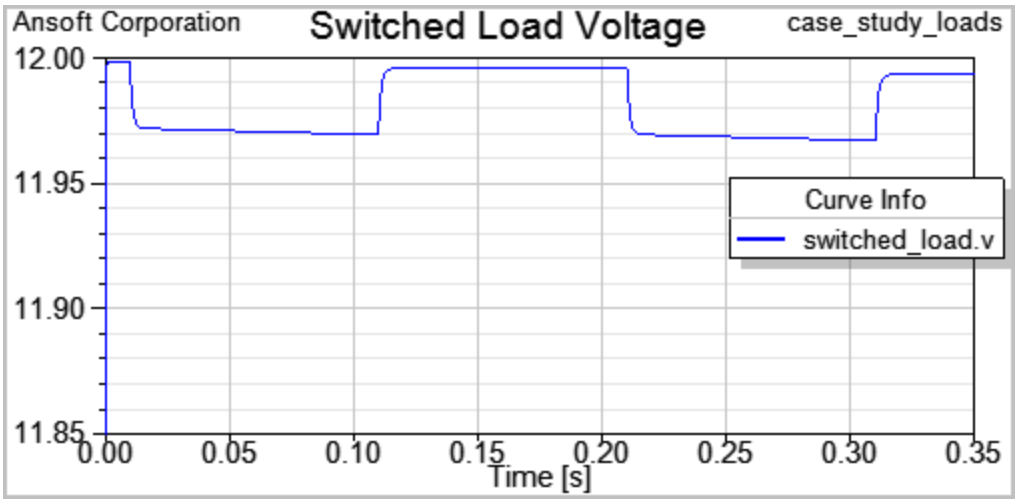
The *ctrl_ramp* and *ctrl_qty* are declared as free quantities within the *switch* architecture.

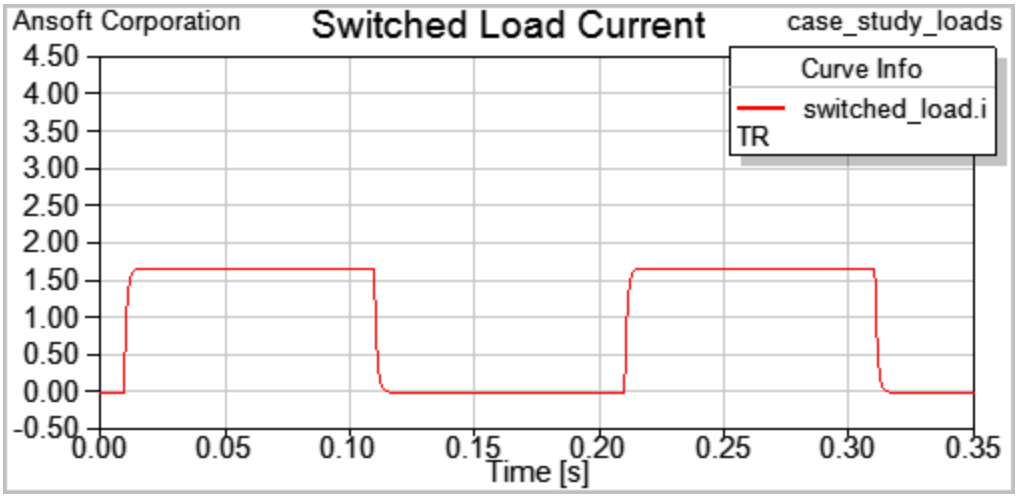
To change the architecture used by a model, double-click the *Load* symbol to open the **Properties** dialog box, and click the **Parameter Values** tab. Choose the **switch** architecture from the **«SimulatorModel** list. Click **OK** to apply the changes.

Results - Switched Load

The first plot illustrates the PT1 behavior of the transformed control signal. The second and third plots show the variation in the voltage and current of the switched load. Because of the voltage drop, actual power is less than the nominal 20W.

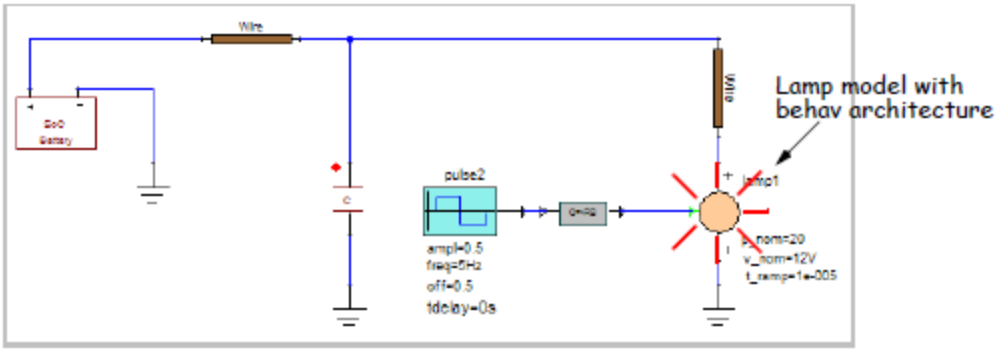






Model: Lamp

This example illustrates the reuse of existing models within new models using component instantiation. The following example illustrates a *Lamp* model whose definition includes the *switch* architecture of the *load* model.



Entity Description - Lamp

The interface of the *lamp* model is similar to that of the *load* model in the following ways: it accepts nominal voltage (*v_nom*), nominal power (*p_nom*), and ramp time (*t_ramp*) as parameter inputs, a control signal port input, and provides the electrical output between two pins, *p* and *m*.

The voltage output from this model is also provided by the *v_outOUT* quantity port. This output value is intended to allow the animation of the lamp symbol.

The complete entity description is as follows:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY lamp IS
 GENERIC(
 p_nom: REAL := 20.0;
 v_nom: VOLTAGE := 12.0;
 t_ramp: REAL := 1.0e-5);
 PORT (
 TERMINAL p,m : ELECTRICAL;
 SIGNAL on_ctrl : IN REAL := 0.0;
 QUANTITY v_out : OUT VOLTAGE := 0.0);
END ENTITY lamp;
```

Architecture Description - Lamp

The *Lamp* model instantiates only the model that is needed in its architecture description. This is in contrast to the component instantiation used in the *Battery* model (Method 2), where all models of the *WORK* library are available for use in an architecture. This methodology should be used if only a few models of the *WORK* library are needed. Every model that is visible in an architecture must be compiled before simulation, which lengthens the simulation time.

The *load* entity is instantiated with the component name *lp* and the *switch* architecture is associated with the component, using the following line:

```
lp: ENTITY WORK.load(switch)
```

This method of instantiation is called direct instantiation. There is no **USE** WORK.**ALL** declaration in the entity description. The generic values and the port values of the lamp model are mapped to those in the load model using the following lines in the architecture:

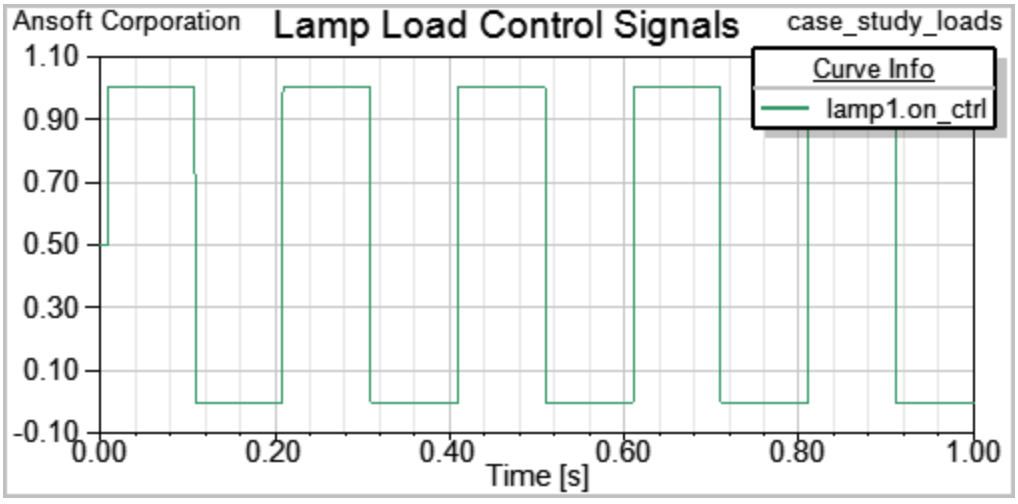
```
GENERIC MAP (p_nom => p_nom, v_nom => v_nom, t_ramp => t_ramp)
PORT MAP (p => p, m => m, on_ctrl => on_ctrl);
```

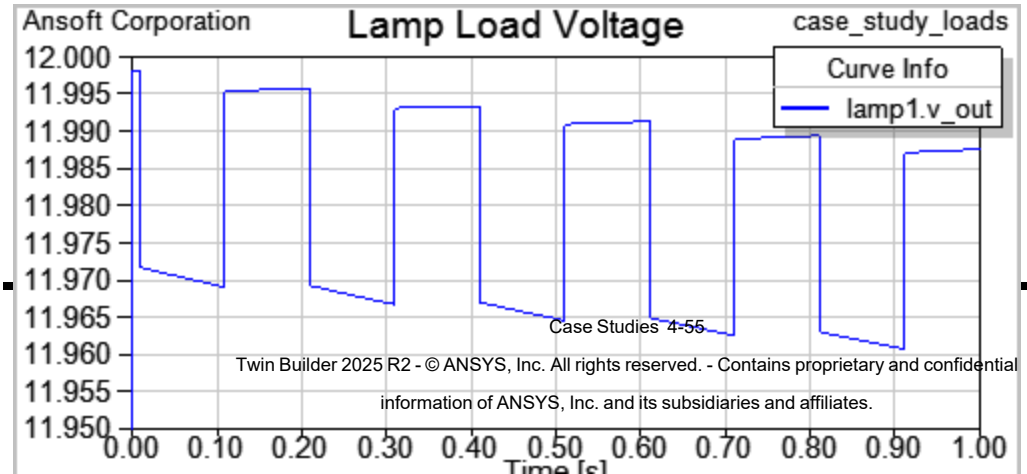
The complete architecture description of the lamp model is as follows:

```
ARCHITECTURE behav OF lamp IS
 QUANTITY v ACROSS p TO m;
BEGIN
 lp: ENTITY WORK.load(switch)
 GENERIC MAP (p_nom => p_nom, v_nom => v_nom, t_ramp => t_ramp)
 PORT MAP (p => p, m => m, on_ctrl => on_ctrl);
 v_out == v;
END ARCHITECTURE behav;
```

Results - Lamp Model

The plots show the voltage response of the lamp model when it is switched ON and OFF using the control signal. The first graph shows the control signal and the second graph the lamp voltage.

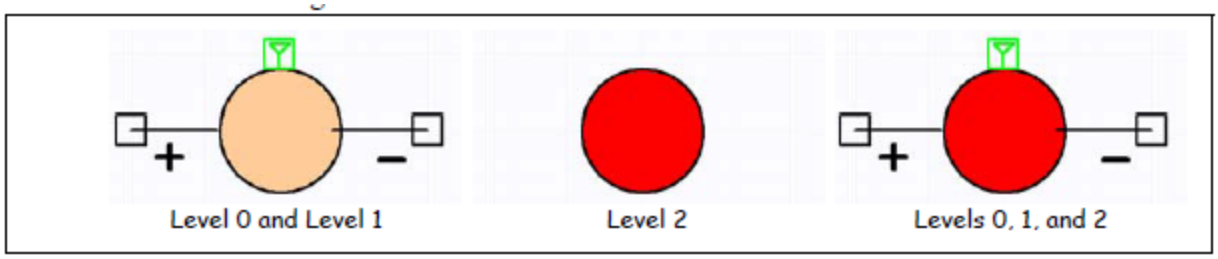




Symbol Animation

An animated symbol changes appearance (shape, color, etc.) during the simulation process, depending on conditions applied to simulation quantities (for example, voltage, current, speed). The lamp circuit uses an animated symbol to illustrate the lamp states ON and OFF. If the lamp output voltage *v_out*> 11.97, levels 0, 1, and 2 of the lamp symbol are displayed. But if *v_out*< 11.97, the symbol displays its default levels 0 and 1, indicating the lamp is OFF.

To obtain different symbol views, graphical elements must be placed on different symbol levels. The conditions defined in animations can only select between different levels in the symbol, not between different drawing elements.



The following steps describe how to implement simulation-dependent animation for the *Lamp* model symbol:

1. Right-click the *lamp* on the schematic sheet, and select **Edit Symbol**. The Symbol Editor opens. Delete the text and the rectangle, select the filled circle icon, draw a circle and center it between the two terminals. Double-click the circle and change the fill color to a beige or another neutral color. Use the Text icon on the Symbol ribbon to create the '+' and '-' symbols. The graphic just created is associated with Level 0 of the model. Copy the circle.

   Make sure the Object Browser is displayed in the left pane and click the Add button in the Object Browser to create Level 1. Click the eye next to Level 0 to turn off display of the graphics and click the eye next to Level 1 to turn on its graphics. Click on the Level 1 folder and past the circle onto the edit sheet. Change the circle's color to red and position it in the same place as the circle in Level 0. Select **File** > **Save**.

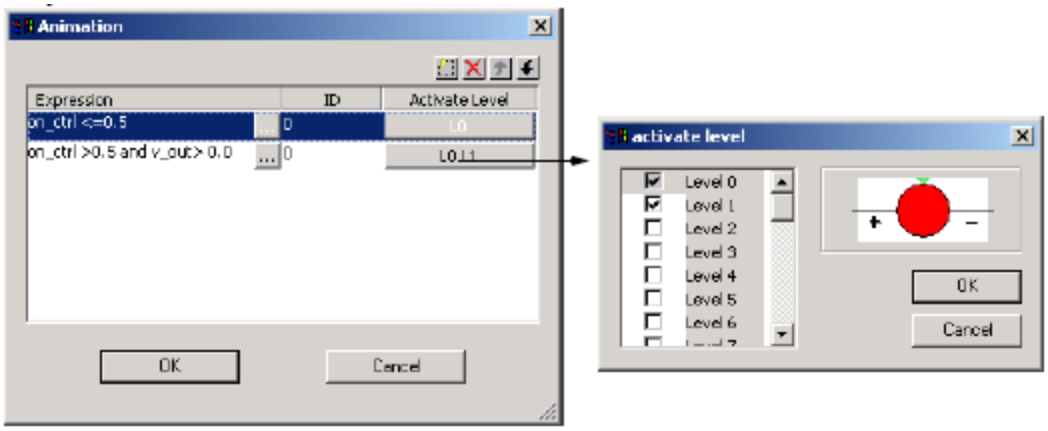
Note:

To work with the symbol animation using an existing model, create an editable copy of the model by clicking the **«Users»** tab of the ModelAgent, selecting the *User* library, and right-clicking on *Packages*. Choose Insert>Model(s) from VHDLA File. Browse to the VHDL-AMS Tutorial folder. From the Tutorial Examples folder, select *lamp.vhd*. In the Insert Model window, check the box for *my_lamp* and click **OK**. The model will be added to the user library and will be editable.

2. Start the symbol editor, then select **Symbol** > **Animate**.

3. Click  to create a new expression for the symbol animation.
4. Define the condition *v_out> 11.97*.

   The model parameter *v_out* controls the symbol view. Make certain the parameter name is used correctly



5. Level 0 is automatically selected for all conditions. Click **Activate Level** for the second condition, and select Level 2. Click **OK** to apply the changes.
6. Select **File** > **Save** and exit the Symbol Editor.

Multidomain System Modeling: Linear Drive System, Solenoid

Concepts - Multidomain System Modeling

This case study uses an electromechanical subsystem in a vehicle powernet to illustrate multidomain modeling. The first example in this case study illustrates the modeling of linear drive

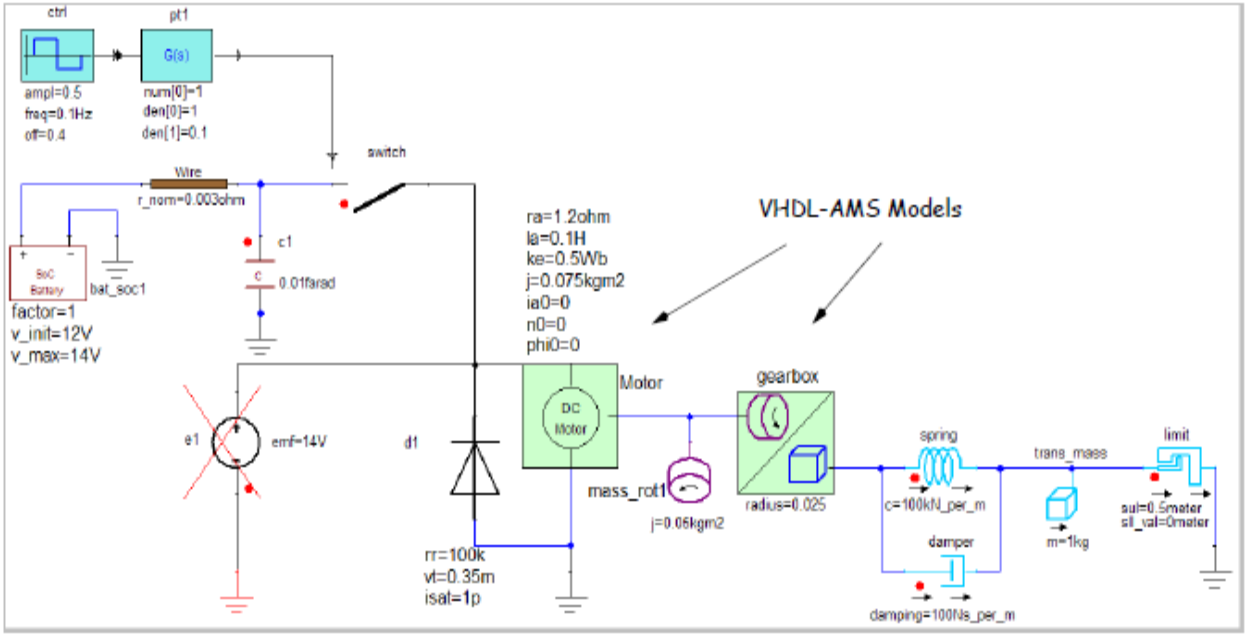
systems that might be used, for example, in a power window system. The second example illustrates the modeling of a simple solenoid that may be used to control hydraulic valves in a vehicle.

Background - Multidomain System Modeling

This linear drive system is powered by the SoC Battery model and is connected to the battery through a switch. The switch control is provided through a pulse wave time function that is passed through a G(s) block to model the PT1 (first-order low-pass response) switching behavior. The motor and gearbox are described by VHDL-AMS models. A free-wheeling diode model is connected in parallel to the motor model to limit the motor voltage, so that the switch is protected when the battery is switched off.

> **Note:**
>
> Note that the student version of Twin Builder cannot use some of the elements that are used to provide the motor voltage through the switch. After opening this case study, select the *bat_soc1*, *c1*, *wire1*, *ctrl*, *pt1*, and *switch* elements, right-click and select **Deactivate (Open)** to deactivate these elements. To use the deactivated element *e1* in place of the other elements, right-click and select **Activate** to activate it.

The rotational mechanical output of the motor model drives a rotational mass that models the inertia. The output from the mass then passes through a gear box model. The translational output of the gear box model displaces the translational mass, the amount of displacement is controlled by a limit stop model. A spring and damper combination is included in the translational domain.



Model: Motor

Entity Description - Motor

The motor model has three conservative pins: *n1* and *n2* are electrical inputs and *ROT* is the mechanical output. The model also accepts the armature resistance as an input quantity *ra*, and provides the rotor angle *phi* and rotor speed *n* as outputs. Because the model defines *ra* as an input **QUANTITY**, it could vary during simulation. None of the other inputs (*ls*, *ke*, *j*, *ia0*, *n0*, *phi0*) can vary during simulation. The other model parameters used in the model are listed in the following table:

Interface	Name	Property	Default Value	Value	Description
GENERIC	la	*INDUCTANCE*	0.01	100m	Armature Inductance [H]
	ke	*FLUX*	1.0	0.5	Rotor Flux Constant [Vs]
	j	*MOMENT_ INERTIA*	0.075	0.075	Rotor Moment of Inertia [kg*m²]
	ia0	*CURRENT*	0.0	0.0	Initial Armature Current [A]
	n0	*VELOCITY*	0.0	0.0	Initial Armature Speed [rpm]
	phi0	*ANGLE*	0.0	0.0	Initial Rotor Position [rad]
QUANTITY	ra	*IN REAL*	1.0	1.2	Armature Resistance [W]

Interface	Name	Property	Default Value	Value	Description
	n, phi	*OUT REAL*	0.0	0.0	Speed [rpm], Rotor Angle [rad]
TERMINAL	n1, n2	*ELECTRICAL*			Electrical Terminals
	rot	*ROTATIONAL_V*			Mechanical Terminal

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
USE IEEE.MECHANICAL_SYSTEMS.ALL;
USE IEEE.MATH_REAL.MATH_PI;
ENTITY motor IS
 GENERIC(
 la : INDUCTANCE := 0.01;
 ke : FLUX := 1.0;
 j : MOMENT_INERTIA := 0.075;
 ia0 : CURRENT := 0.0;
 n0 : VELOCITY := 0.0;
 phi0 : ANGLE := 0.0);
 PORT(TERMINAL n1, n2 : ELECTRICAL;
 TERMINAL rot : ROTATIONAL_V;
 QUANTITY ra : IN RESISTANCE := 1.2;
 QUANTITY n : OUT VELOCITY;
 QUANTITY phi : OUT ANGLE);
END ENTITY motor;
```

The model needs to use the *electrical_systems* and *mechanical_systems* packages of the *IEEE* library in order to use the *ELECTRICAL* and *MECHANICAL* conservative pins, respectively. Additionally, since the model architecture requires π, the *math_real* package needs to be used.

Architecture Description - Motor

For conversions between translational and rotational values in the mechanical domain, the model architecture defines the following two constants:

Name	Value
n2om	2.0 * p / 60.0
om2n	60.0 / (2.0 * p)

The across and through quantities for the conservative pins are defined as follows:

Conservative Pins	Across	Through
n1, n2	v	i
rot	omega	torque

The DC motor behavior is based on the following equations:

$$\phi = L_a \cdot i \qquad mi = ke \cdot i$$

$$\frac{\partial}{\partial t}(angle) = \omega \qquad \frac{\partial \omega}{\partial t} = \frac{(mi + inertia)}{j}$$

$$v = i \cdot ra + \frac{\partial \phi}{\partial t} + ke \cdot \omega$$

The speed and position of the rotor are output through *n* and *phi*. Three free quantities *angle*, *flux*, and *mi* are declared within the model architecture. Initial values are provided to *angle*, *omega*, and *flux* quantities using the **BREAK** statement.

The equivalent VHDL-AMS description for defining the model architecture is as follows:

```
ARCHITECTURE behav OF motor IS

 QUANTITY v ACROSS i THROUGH N1 TO N2;

 QUANTITY omega ACROSS torque THROUGH rot TO rotational_v_ref;
```

```
 QUANTITY local_mi, local_angle, local_flux : REAL;
 CONSTANT n2om : REAL := 2.0 * math_pi / 60.0;
 CONSTANT om2n : REAL := 60.0 / (2.0 * math_pi);
BEGIN
 BREAK
 local_angle => phi0,
 omega => n0 * n2om,
 local_flux => ia0 * la;
 v == i * ra + local_flux'DOT + ke * omega;
 local_mi == ke * i;
 omega'DOT == (1.0 / j) * (local_mi + torque);
 local_angle'DOT == omega;
 local_flux == i * la;
 n == omega * om2n;
 phi == local_angle;
END ARCHITECTURE behav;
```

Model: Gearbox

Entity Description - Gearbox

This model accepts rotational input (velocity-torque representation) and provides output on a translational (displacement-force representation) pin. The gear radius is defined as a **generic** REAL input value. The parameters used in the model are listed in the following table:

Interface	Name	Property	Default Value	Value	Description
GENERIC	radius	REAL	0.1	0.025	Gear radius [m]
TERMINAL	rot	ROTATIONAL_V			
	trans	TRANSLATIONAL			

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
LIBRARY IEEE;
USE IEEE.MECHANICAL_SYSTEMS.ALL;
ENTITY gearbox IS
GENERIC(radius : REAL := 0.1);
PORT(TERMINAL rot : ROTATIONAL_V;
TERMINAL trans : TRANSLATIONAL);
END ENTITY gearbox;
```

Architecture Description - Gearbox

The gearbox model transforms the rotational input (velocity-force) to translational output (distance-torque representation) based on the specified gear radius. The across and through quantities for the conservative pins are defined as follows:

Conservative Pins	Across	Through
ROT, ROTATIONAL_V_REF	omega	torque
TRANS, TRANSLATIONAL_REF	position	force

The transformation is performed by the following two equations:

$$\frac{\partial}{\partial t}(position) = \omega \cdot r \qquad \tau = -Force \cdot r$$

The equivalent VHDL-AMS description for the model architecture is as follows:

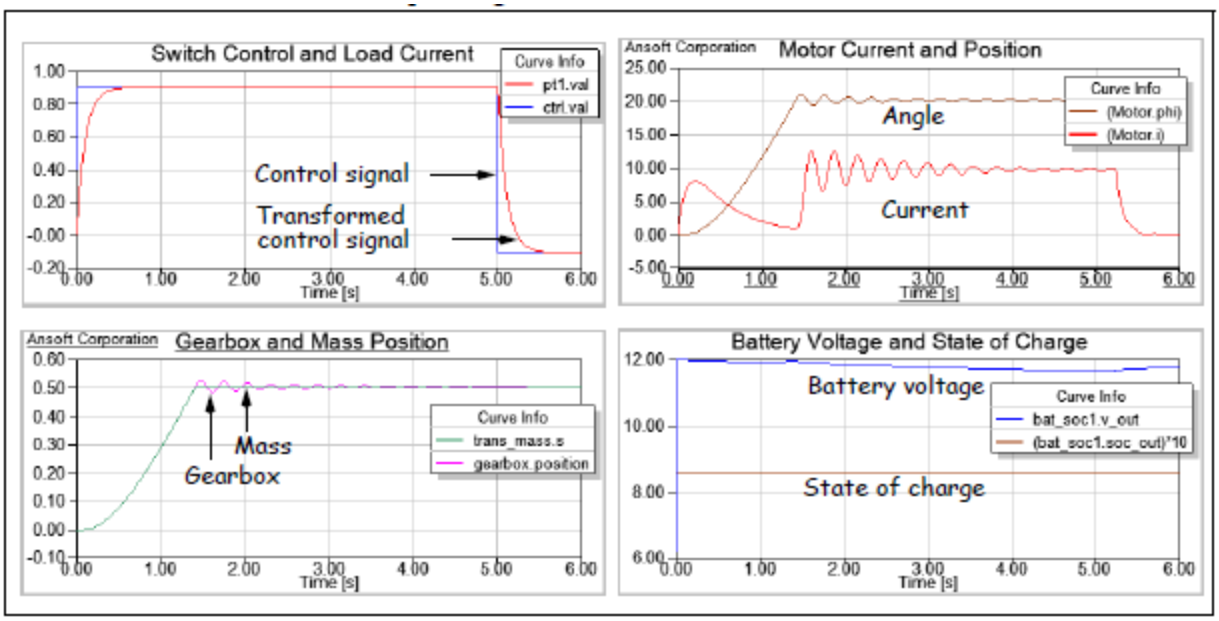
```
ARCHITECTURE behav OF gearbox IS
 QUANTITY omega ACROSS torque THROUGH rot TO ROTATIONAL_V_REF;
 QUANTITY position ACROSS force THROUGH trans TO TRANSLATIONAL_REF;
BEGIN
 position'DOT == omega * radius;
 torque == -force * radius;
```

```
END ARCHITECTURE behav;
```

The translational force output displaces the translational mass of 1 kg (this might represent the power window in a vehicle). The displacement of the mass is limited to 50cm by the *Limit Stop* model. Appropriate damping/spring rate coefficients are set for *Limit Stop*, as well as for other mechanics in the TRANSLATIONAL domain.

Results - Multidomain System Modeling

The first graph shows the PT1 behavior of the switch control signal modeled with a Laplace Transform. The second graph shows the motor current and position characteristics. The third graph shows the displacement of the mass, limited at an upper limit of 0.5m, and the damping effects. The fourth graph shows the battery voltage and state of charge characteristics. The 'window' closes at about 1.5 sec, but the motor is kept energized for another 3.5 sec.



Solenoid System

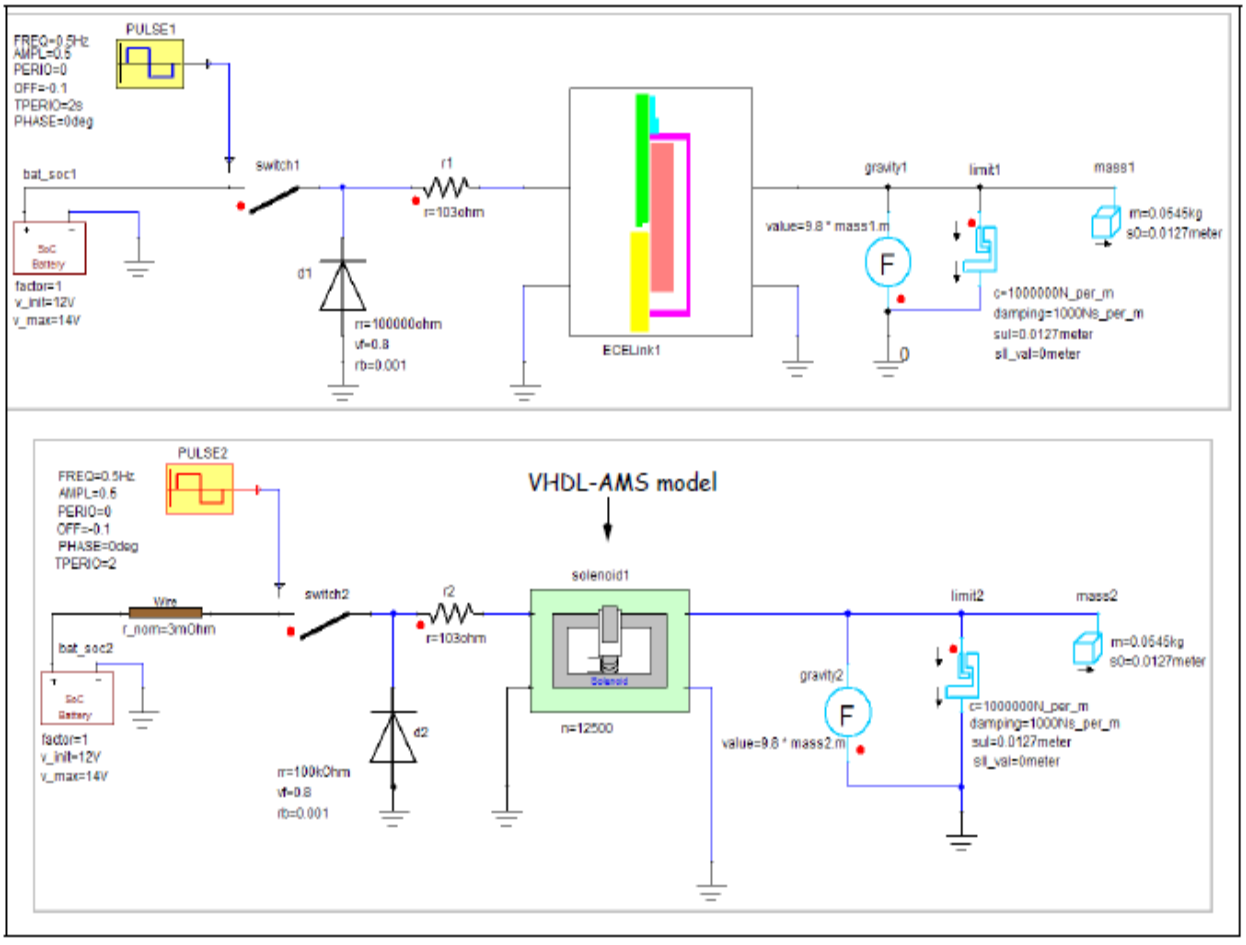
Background - Solenoid System

Twin Builder allows the use of finite element data generated from a Maxwell circuit simulation through the ECE (Equivalent Circuit Export) link. In this example, two solenoid models are simulated. The first model uses finite element data from the ECE link (Maxwell model) and the second uses an approximated model developed using VHDL-AMS (VHDL-AMS model).

The solenoid is powered by the SoC battery and is controlled by a switch that functions through a pulse model. Each solenoid model has two electrical pins for input, and two translational pins

for output. Note that solenoid models can also convert energy in the other direction, mechanical to electrical. The mechanical force, measured using a force meter, is used to displace a translational mass representing the plunger. The opposing force of gravity on the plunger is represented by a force (f=mass*9.8m/s²), and the displacement of the plunger is controlled by a limit stop model.

The VHDL-AMS model uses an idealized equation based on the input current value and the position of the plunger to determine the flux linkage in the solenoid and the corresponding force on the plunger. The Maxwell model is more accurate since it uses the finite element data that models fringing and saturation effects.



Model: VHDL-AMS Solenoid

Entity Description - Solenoid

The VHDL-AMS solenoid is described by equations derived in Woodson and Melcher: *Electromechanical Dynamics*[1]. The Solenoid model accepts three parameters: the maximum

inductance value per turn at minimum air gap L0, an inductance coefficient K, and the number of coil turns N. These parameters are used for the computation of the instantaneous inductance value, as follows (x is the displacement of the plunger).

$$L(x) = \frac{N^2 \cdot L_0}{1 + Kx}$$

The gap x begins at 0.0127m (0.5in) and ends at 0.0, when the solenoid is fully closed.

The value of K might be derived from analyzing the magnetic circuit, from measurement, or in this case, from finite element solutions. But given that L(x) follows this equation, the coil voltage is:

$$V = \frac{d\lambda}{dt} = \frac{d}{dt}(L \cdot i) = N^2 \cdot L_0 \cdot \frac{d}{dt}\left\{\frac{i}{1 + Kx}\right\}$$

Both x and the coil current i may vary with time. Carrying out the derivatives:

$$V = N^2 \cdot L_0 \left\{\frac{i}{1 + Kx}\frac{di}{dt} - \frac{iK}{(1 + Kx)^2}\frac{dx}{dt}\right\}$$

The di/dt term is a transformer voltage, and the dx/dt term is a speed voltage.

On the mechanical side, an energy conversion process generates a force according to:

$$f = \frac{d}{dx}W_m' \qquad W_m' = \int_0^i \lambda(i', x)di'$$

where W_m' is the magnetic coenergy and i' is a dummy variable for integration.

If the core material is linear, then the magnetic energy and coenergy are the same. Substituting for L and then carrying out the integration:

$$\lambda(i', x) = \frac{N^2 \cdot L_0 \cdot i'}{1 + Kx} \qquad W_m' = \frac{N^2 \cdot L_0}{1 + Kx} \cdot \frac{i^2}{2}$$

The force is calculated by taking a partial derivative with respect to x:

$$f = \frac{N^2 \cdot L_0 \cdot i^2}{2} \frac{\partial}{\partial x}\left\{\frac{1}{1+Kx}\right\} \qquad f = -\frac{N^2 \cdot L_0 \cdot K \cdot i^2}{2 \cdot (1+Kx)^2}$$

The negative sign means that f acts to decrease x, for any current $i \neq 0$.

The VHDL-AMS model uses flux, so that the following simplifications are possible:

$$\lambda = \frac{N^2 \cdot L_0 \cdot i}{1+Kx} \qquad f = -\frac{\lambda^2 \cdot K}{2 \cdot L_0 \cdot N^2} \qquad V = \frac{d\lambda}{dt}$$

The force is λ^2 times a constant called Fk. The term $N^2 L_0$ appears in two equations, and it is a constant. Therefore:

$$L_{max} = N^2 \cdot L_0 \qquad \lambda = \frac{L_{max} \cdot i}{1+Kx} \qquad F_K = \frac{K}{2 \cdot L_{max}}$$

$$V = \frac{d\lambda}{dt} \qquad f = -\lambda^2 \cdot F_K$$

L_{max} is the maximum inductance, which occurs when x is zero.

The VHDL-AMS model includes two external and two mechanical pins. Given a current i and position x, the model equations calculate λ and f. The value of f will be passed to the mechanical pins as a through variable. The quantity $d\lambda/dt$ appears across the electrical pins.

One might follow a similar process to develop the VHDL-AMS models for other multidomain components. First, the equations for stored energy in each of the domains, in terms of the through and across variable, must be obtained. Then, the energy (or coenergy) is equated. Lossy elements could be added separately.

The model accepts a current value from electrical pins *p* and *m*, and a position value from the mechanical pins, *pos1* and *pos2*. The parameters used in the model are listed in the following table:

Interface	Name	Type	Default Value	Description
GENERIC	l0	INDUCTANCE	1.25 e-7	Max inductance per turn at min gap

Interface	Name	Type	Default Value	Description
				[H]
	k	REAL	197.735	Inductance coefficient
	n	REAL	1.0	Number of coil turns
TERMINAL	p, m	ELECTRICAL		Electrical pins
	pos1	TRANSLATIONAL		Mechanical pins of translational domain
	pos2	TRANSLATIONAL		

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
USE IEEE.MECHANICAL_SYSTEMS.ALL;
ENTITY solenoid IS
 GENERIC (
    L0 : INDUCTANCE := 1.25e-7;
    K : REAL := 197.735;
    N : REAL := 1.0);
 PORT (
    TERMINAL p,m : ELECTRICAL;
    TERMINAL pos1, pos2 : TRANSLATIONAL);
    QUANTITY force_out : OUT REAL := 0.0);
END ENTITY solenoid;
```

1. Woodson and Melcher: *Electromechanical Dynamics, Part 1*, 1968

Architecture Description - Solenoid

The across and through quantities for the conservative pins are defined as follows:

Conservative Pins	Across	Through
p, m	voltage	current
pos1, pos2	position	force

The equations for this model are as follows:

$$L_{max} = N^2 \cdot L_0 \qquad L = \frac{L_{max}}{1 + Kx} \qquad F_k = \frac{K}{2 \cdot L_{max}}$$

$$\lambda = L \cdot i \qquad v = \frac{\partial \lambda}{\partial t} \qquad F = -\lambda^2 \cdot F_K$$

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ARCHITECTURE behav OF solenoid IS
 CONSTANT Lmax : INDUCTANCE := L0 * N * N;
 CONSTANT Fk : FORCE := K / (2.0 * Lmax);
 QUANTITY v ACROSS i THROUGH p TO m;
 QUANTITY position ACROSS force THROUGH pos1 TO pos2;
 QUANTITY L : INDUCTANCE;
 QUANTITY flux : FLUX;
BEGIN
 IF (position > 0.0) USE
 L == Lmax / (1.0 + K * position);
 ELSE
 L == Lmax;
 END use;
 flux == L * i;
```

```
v == flux'DOT;
force == flux * flux * Fk;
force_out == -force;
END ARCHITECTURE behav;
```

In this example, N=12500 turns, but the constant K is not readily known. It may come from a magnetic circuit analysis of the device, or it may come from two measurements at different gap spacings. In this example finite element solutions at two values of x are used:

x	i	L	f
0.0	0.084	1.251e-7	-27
0.0127	0.084	3.56e-8	-0.84

Here the value of i comes from an excitation of 1050 ampere-turns, and L is the inductance per turn.

For one approach, we could choose K to match L at both values for x. This should represent the electrical side better, over the whole range of plunger travel.

$$3.56\text{e-}8 = \frac{1.251\text{e-}7}{1 + 0.0127 \cdot K} \qquad K = 197.735$$

Another approach is to match both L and f, when x is zero.

$$L_{max} = 12500^2 \cdot 1.251\text{e-}7 = 19.547$$
$$\lambda = L_{max} \cdot i = 19.547 \cdot 0.084 = 1.642$$
$$F_K = \frac{27}{1.642^2} = 10$$
$$K = 2 \cdot F_K \cdot L_{max} = 2 \cdot 10 \cdot 19.547 = 391.52$$

These two values of K are quite different. It is not possible to match both L and f over the whole operating range because of the solenoid's departure from ideal behavior.

In this example, we choose the lower K value, because we want to represent the coil current as accurately as possible. If the mechanical side is of more concern, choose the higher K value or a compromise value (300).

The VHDL-AMS solenoid model is limited since it assumes that the core material is linear, and ignores the effects of flux fringing. A better model might result from finite element analysis.

Results - Solenoid Model

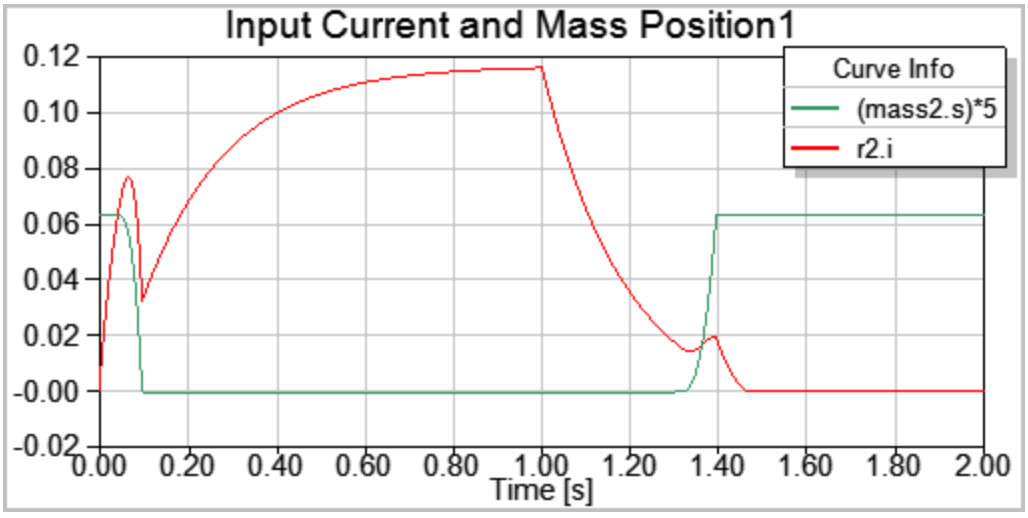
The characteristics plotted in the graph show the behavior of the circuit using the VHDL-AMS solenoid when the battery is connected to the circuit for a period of 1 sec.

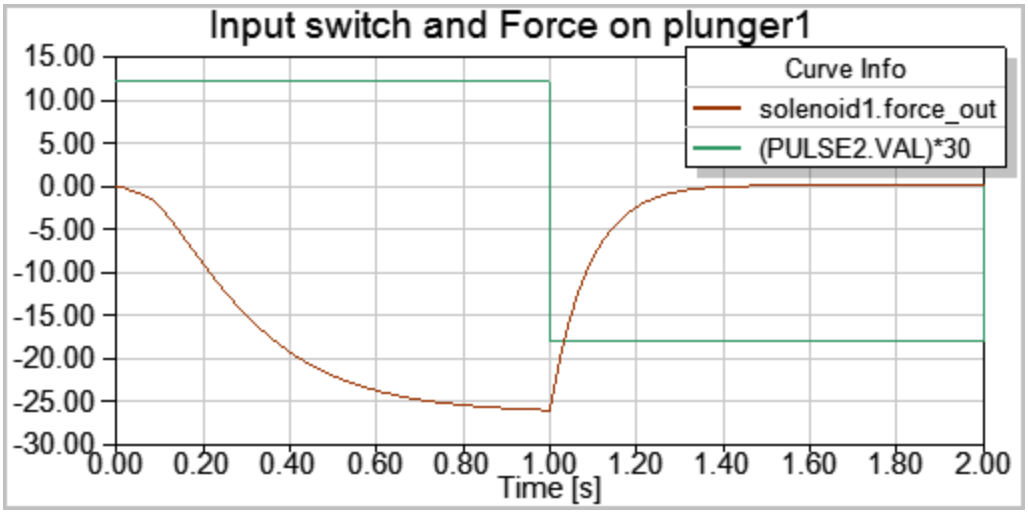
The first graph shows coil current and position of the mass model. The second graph shows the input switch signal and the force on the plunger.

The mass model is moved from an initial position of 0.0127m to a final position of 0m within 0.1 sec, thereby closing the plunger gap. When the battery is disconnected from the circuit after 1 sec, the force of gravity causes the plunger to open again.

When the solenoid is first energized, the coil current increases with a fast L/R time constant because L is minimum at the open position.When the plunger begins to close, it generates a back EMF that causes a transient dip in coil current. When the solenoid is closed and L is maximum, the current increases with a slow time constant. The final coil current is determined by the battery voltage and coil resistance.

When the switch opens at 1 sec, free-wheeling diodes are necessary to permit gradual decay of the inductive coil current. When the current has been delayed enough, gravity overcomes the electromagnetic force, and the solenoid begins to open. As the plunger moves, it generates a back EMF that causes a transient increase in the coil current.





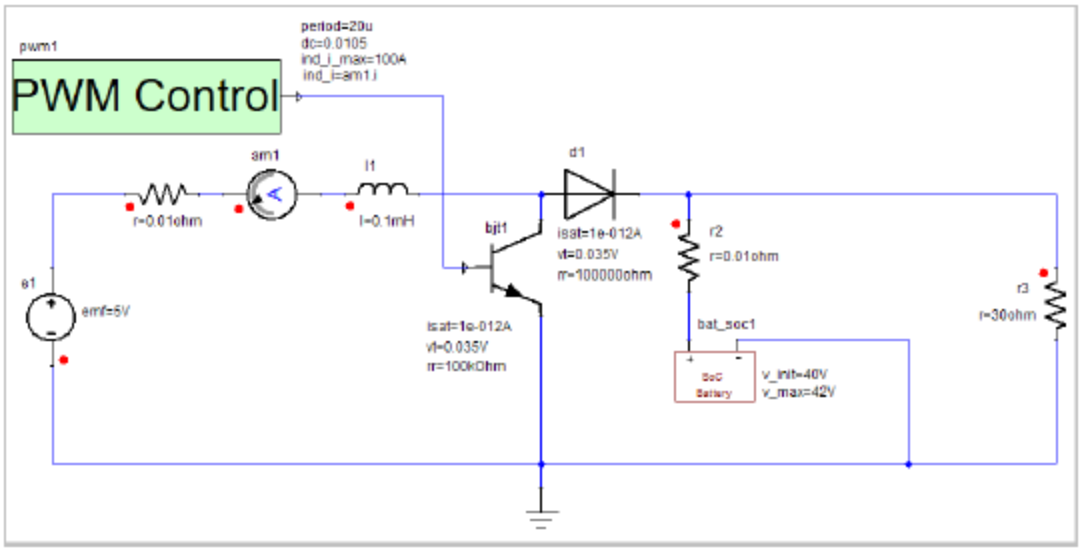
Mixed-Signal Modeling: DC-DC Model with PWM, Automotive Alarm System

Concepts - Mixed-Signal Modeling

This case study illustrates several facets of digital modeling. A VHDL-AMS model of a Pulse-Width Modulator (PWM) controlling a simple DC-DC boost converter circuit illustrates digital circuit modeling. An automotive alarm system illustrates the use of a stimulus generator, and shows how to export VHDL-AMS models and use foreign models.

DC-DC Model

The boost converter in the example steps up a 5V ideal source to a 42V battery-fed system. A PWM control based on the actual and reference current values switches an ideal BJT transistor. The focus of this example is the development of the PWM controller in VHDL-AMS.



A reference clock signal (cond1) is created within the PWM controller block with a period of 20µs and a duty cycle of 0.0105. A second signal (cond2) is created within the PWM model every time the current in the inductor l1 exceeds a reference value of 100A. The PWM controller block outputs an ON control signal if both the reference clock is ON (cond1) and the actual inductor current is less than the reference value (NOT cond2). Consequently, the control signal for the transistor is pulse-width modulated. It is switched on for the period of time when the condition (cond1 AND NOT cond2) is true.

Model: PWM Controller

Entity Description PWM Controller

The PWM controller modeled for this example accepts three parameters: a reference clock period (*period*), a duty cycle value for the reference clock (*dc*), and a maximum reference current value for the inductor current (*ind_i_max*). It has two port quantities: the actual inductor current value (*ind_i*) as provided via the ammeter, and a control signal output for switching the BJT (*ctrl*). Default values are provided for all generic and port values.

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY pwm IS

GENERIC(

 period : REAL := 1.0e-3; -- clock period

 dc : REAL := 0.5; -- duty cycle
```

```
 ind_i_max : CURRENT := 100.0); -- reference inductor current
PORT(
QUANTITY ind_i : IN CURRENT := 0.0; -- Inductor Current
QUANTITY ctrl : OUT REAL := 0.0); -- Control Signal Output
END ENTITY pwm;
```

Entity Description PWM Controller

The PWM controller modeled for this example accepts three parameters: a reference clock period (*period*), a duty cycle value for the reference clock (*dc*), and a maximum reference current value for the inductor current (*ind_i_max*). It has two port quantities: the actual inductor current value (*ind_i*) as provided via the ammeter, and a control signal output for switching the BJT (*ctrl*). Default values are provided for all generic and port values.

The equivalent VHDL-AMS description for defining the model interface is as follows:

```
ENTITY pwm IS
 GENERIC(
 period : REAL := 1.0e-3; -- clock period
 dc : REAL := 0.5; -- duty cycle
 ind_i_max : CURRENT := 100.0); -- reference inductor current
 PORT(
 QUANTITY ind_i : IN CURRENT := 0.0; -- Inductor Current
 QUANTITY ctrl : OUT REAL := 0.0); -- Control Signal Output
END ENTITY pwm;
```

Architecture Description - PWM Controller

The model architecture is divided into two basic functional parts that are represented as processes. Constant values are declared for *time_period*, *hi_period*, and *lo_period* of the reference clock signal, based on the specified period and duty cycle of the signal, as follows:

$$hi_{period} = dutycycle \cdot timeperiod = dc \cdot period \qquad lo_{period} = timeperiod - hi_{period}$$

Two *BOOLEAN* signals *cond1* and *cond2* are used along with a REAL signal *on_ctrl*. The second signal *cond2* uses the '*ABOVE* attribute to identify if the actual current value *ind_i* has exceeded the maximum reference current value *ind_i_max*.

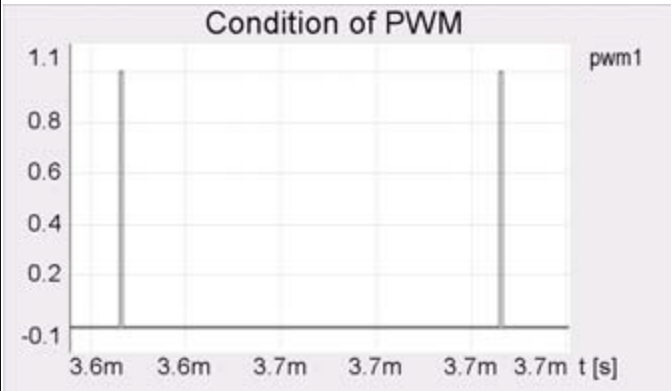
```
cond2 <= ind_i'ABOVE(ind_i_max);
```

The previous statement is equivalent to the pseudocode

```
IF (actual current > maximum reference current) THEN cond2 <= TRUE.
```

A process statement specifies the sequential behavior of a model. It can be triggered by a set of signals, conditions, or at specific time instances. The first process in the model (*ref_clock*) outputs the reference clock signal as a BOOLEAN value (cond1: TRUE=>ON) according to the specified values of *lo_period* and *hi_period*. It uses a **loop** statement along with a **WAIT** statement to model the output BOOLEAN clock signal. The **WAIT** statement suspends the process for a specific period of time. The following figure shows this PWM clock signal.

```
ref_clock : PROCESS
BEGIN
 LOOP
 cond1 <= FALSE;
 WAIT FOR lo_period;
 cond1 <= TRUE;
 WAIT FOR hi_period;
 END LOOP;
END PROCESS ref_clock;
```



The second process, called *ctrl*, has a sensitivity list of the signals that affect the process. In this case, the process is sensitive to events that occur on *cond1* and *cond2* signals. This means that the sequential **IF** statement within the process is executed whenever either *cond1* or *cond2* changes value from *TRUE* to *FALSE* or vice versa. The **IF** statement in the model implements the following pseudocode:

```
IF (actual current > maximum reference current) THEN control <= OFF
(Switch BJT OFF)

ELSE
```

```
IF (actual current < maximum reference current) AND (reference clock
is ON)

THEN control <= ON (Switch BJT ON)
```

```
pwm_ctrl : PROCESS (cond1,cond2)
BEGIN
 IF (cond2) THEN
 ctrl_sig <= 0.0;
 ELSIF (cond1 AND (NOT cond2)) THEN
 ctrl_sig <= 1.0;
 END IF;
END PROCESS pwm_ctrl;
```

If neither condition holds true, then *ctrl_sig* remains unchanged. The BJT can be switched ON at each clock pulse shown in the previous figure, unless the current already exceeds the reference level. The BJT will then switch OFF when the current reaches the reference level.

The digital control signal is transformed to an analog quantity using the '*RAMP* attribute.

The equivalent VHDL-AMS description for defining the model interface is as follows:

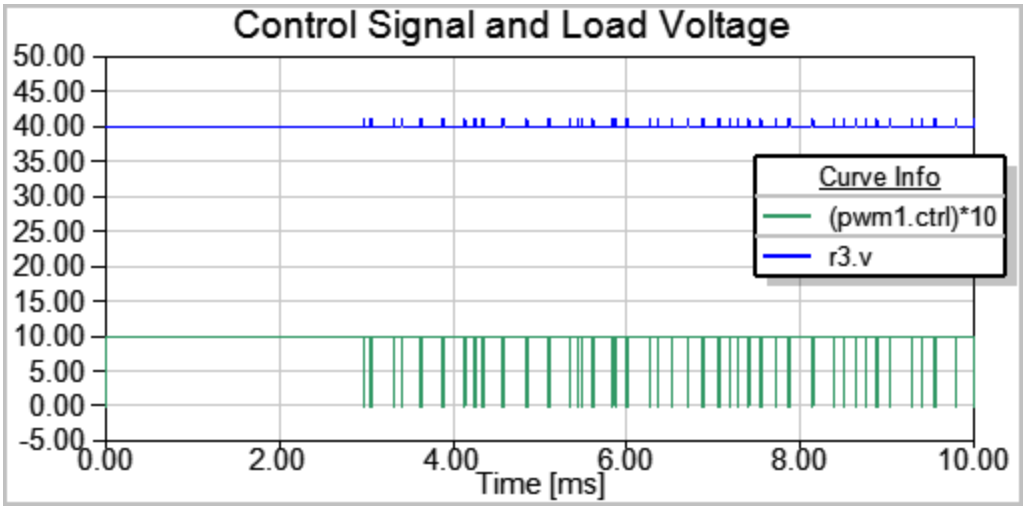
```
ARCHITECTURE behav OF pwm IS
 CONSTANT time_period : TIME := 1 sec*period;
 CONSTANT hi_period : TIME := 1 sec*(dc*period);
 CONSTANT lo_period : TIME := time_period - hi_period;
 SIGNAL cond1, cond2 : BOOLEAN := FALSE;
 SIGNAL ctrl_sig : REAL := 0.0;
BEGIN
 cond2 <= ind_i'ABOVE(ind_i_max);
 ref_clock : PROCESS
 BEGIN
 LOOP
```

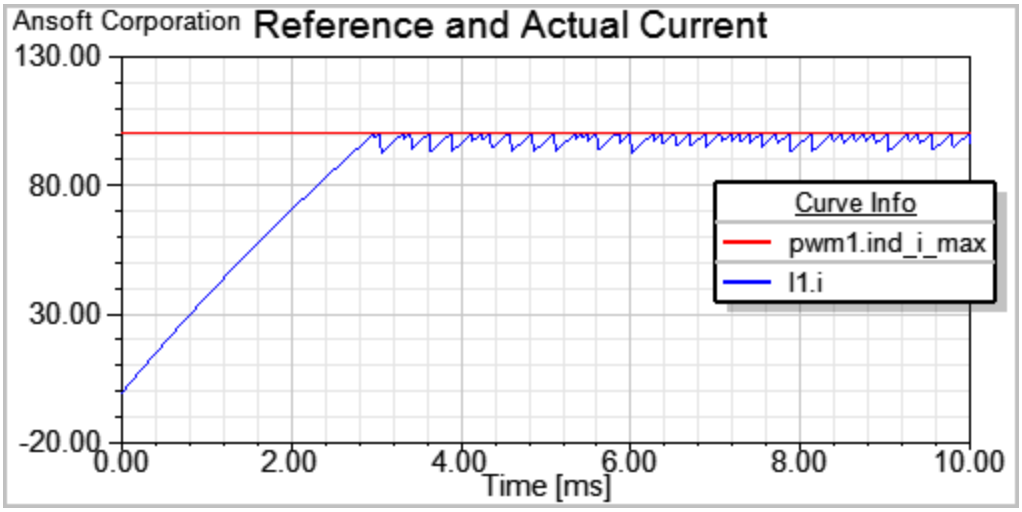
```
cond1 <= FALSE;
WAIT FOR lo_period;
cond1 <= TRUE;
WAIT FOR hi_period;
END LOOP;
END PROCESS ref_clock;
pwm_ctrl : PROCESS (cond1,cond2)
BEGIN
IF (cond2) THEN
ctrl_sig <= 0.0;
ELSIF (cond1 AND (NOT cond2)) THEN
ctrl_sig <= 1.0;
END IF;
END PROCESS pwm_ctrl;
ctrl == ctrl_sig'RAMP(0.0,0.0);
END ARCHITECTURE behav;
```

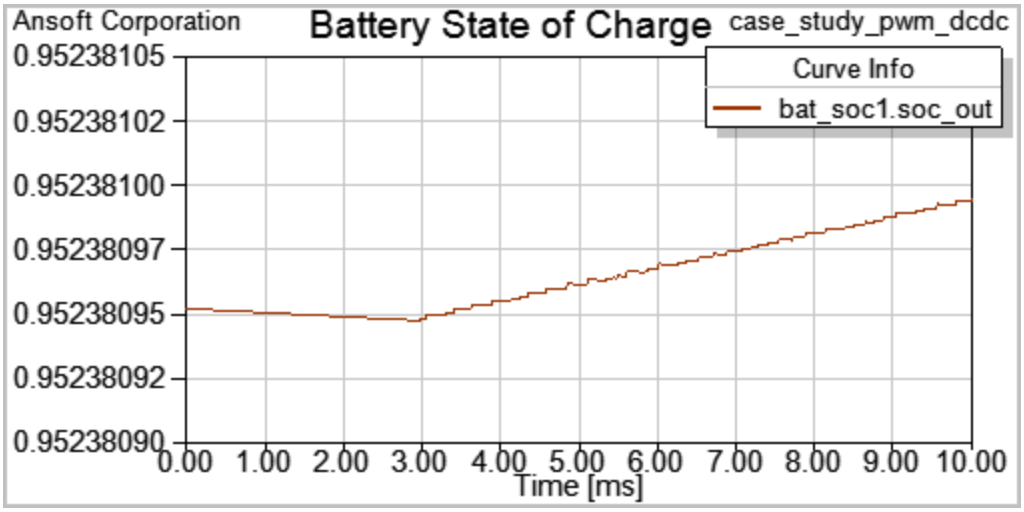
Results - PWM Controller Model

The first graph shows the control output of the PWM controller and the corresponding voltage of the load resistor. The outputs are pulse-width modulated. The second graph shows the variation of the actual inductor current which is being controlled with respect to the maximum allowable inductor current. The third graph shows the variations of the battery state of charge as a consequence of the switching action of the PWM controller

The BJT is switched ON at each clock pulse during the first 3ms,as the current builds to 100A. Then the BJT begins to switch OFF at a variable time during each clock period. The BOOST converter begins to charge the battery each time the BJT switches OFF.







Automotive Alarm System

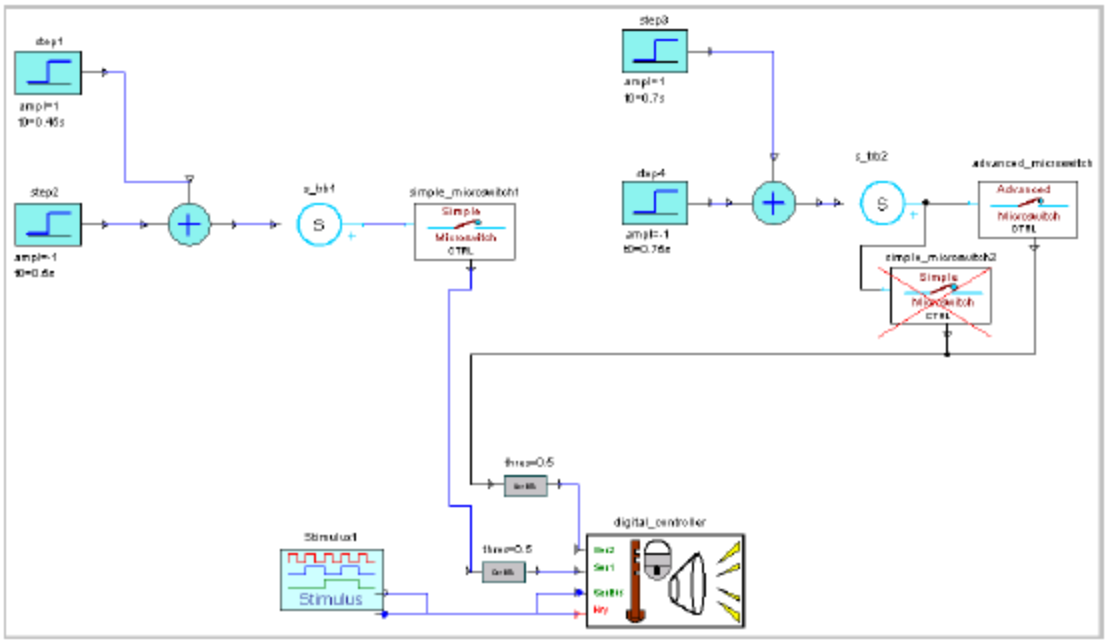
Background - Automotive Alarm System

The automotive alarm system is a simplified example of an electronic vehicle security protection system. It illustrates modeling concepts for multidomain sensors and digital logic and the use of a stimulus generator to create stimulus input patterns.

The system has three main parts:

- Multidomain microswitch sensor models that detect alarm conditions and provide control signals indicating that an alarm situation has occurred
- Digital stimulus generator that provides the remote key switch and digital vector sensor inputs
- Digital controller that arms the vehicle and provides a siren output if the security of an

armed vehicle has been compromised



This system uses two types of mechanical microswitches (simple and advanced) that act as linear position sensors. The mechanical inputs for the sensor models are connected to translational domain position source models. These mechanical domain source models use a combination of step inputs to generate a pulse characteristic for the displacement value. The sensor model outputs vary as analog quantities and are converted to digital signals by OmniCaster models. Note that OmniCasters are automatically inserted when a connection is made between the analog control output of the sensor and the digital control input of the digital controller.

In order to simulate the example as-is, a license for the Twin Builder Sensor Library is required. If this license is not available, you can simulate the example by deactivating the advanced_ microswitch model on the sheet and enabling the simple_microswitch2 model. Right-click the model and select **Deactivate (Open)**.

Model: Simple Microswitch

The first microswitch model (*simple_microswitch*) is a simplified sensor and is developed as a text subsheet in VHDL-AMS. It accepts a threshold position parameter and a varying position input. It determines when the sensor position has crossed the threshold and outputs a control quantity that is used by the digital controller.

Entity Description - Simple Microswitch

This is a mechanical model and consequently, includes the *mechanical_systems* package. It accepts the *position_threshold* parameter as a generic input, and the mechanical input through

a conservative TRANSLATIONAL domain pin. When the mechanical displacement on this conservative pin exceeds the parameter value of *position_threshold*, the *control_output* quantity signal is activated.

```
LIBRARY IEEE;
USE IEEE.MECHANICAL_SYSTEMS.ALL;

ENTITY simple_microswitch IS
 GENERIC(
 position_threshold : DISPLACEMENT := 0.5e-3);
 PORT(
 TERMINAL mech_input : TRANSLATIONAL;
 QUANTITY control_output : OUT REAL);
END ENTITY simple_microswitch;
```

Architecture Description - Simple Microswitch

The model architecture uses the 'ABOVE attribute to determine the threshold crossing and an IF-USE statement to set the control quantity output.

```
ARCHITECTURE behav OF simple_microswitch IS
 QUANTITY s_val ACROSS mech_input TO TRANSLATIONAL_REF;
 SIGNAL crossing : BOOLEAN := FALSE;
BEGIN
 crossing <= s_val'ABOVE(position_threshold);
 BREAK ON crossing;
 IF (crossing) USE
 control_output == 1.0;
 ELSE
 control_output == 0.0;
END USE;
```

```
END ARCHITECTURE behav;
```

Model: Advanced Microswitch

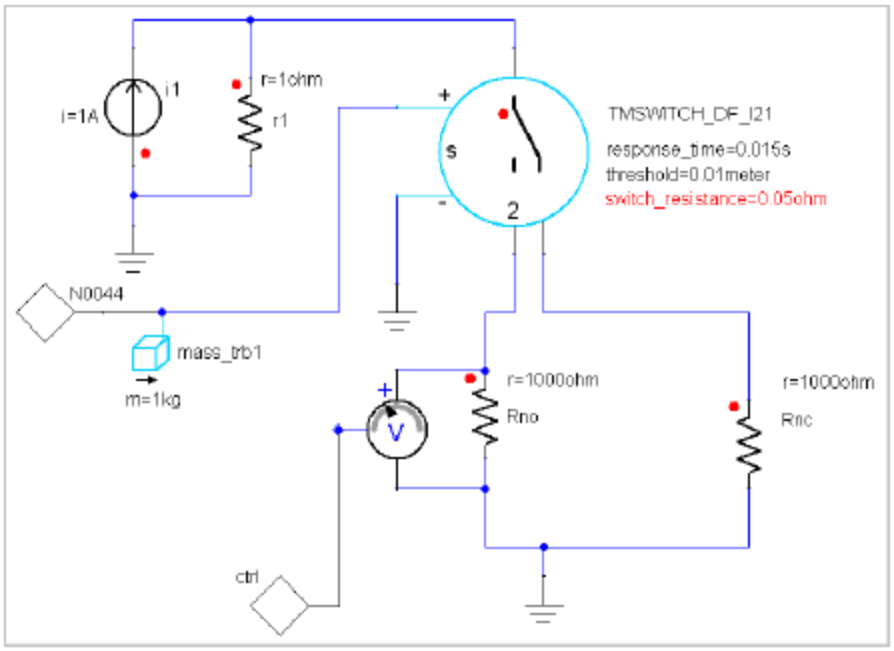
The second sensor (*advanced_microswitch*) is a slightly more complicated model developed as a graphical subsheet with a microswitch component from the Twin Builder Sensor Add-on library.

The Sensor library offers a wide variety of sensor models with a choice of using system-level models with idealized behavior, as well as device-level models with non-ideal behavior such as temperature sensitivity, nonlinearity, hysteresis, and measurement error. In general, all the models from the sensor library offer “ready parameterization,” i.e., they can be easily parameterized by using datasheet information. Additionally, the library includes a complete set of building blocks that enable users to create custom sensor models very quickly using an easy-to-use, color-coded format.

The advanced microswitch is modeled as a subcircuit and uses a Level 2 Sensor Model from the **Simplorer Elements** > **Multiphysics** > **Sensors** in the **Component Libraries** window. The model is available from **Level 2 Sensor Models** > **Linear Position** > **Displacement-Force** > **Microswitch**, and the subcircuit can be modeled as shown below. Note that except for the microswitch model, all other elements are from the *Basic Elements VHDLAMS* library.

This component accepts position input from a conservative mechanical terminal and toggles from its normally closed terminal to its normally open terminal when the specified threshold position is exceeded by some element in the displacement-force translational domain. The voltage output from this model's conservative electrical terminals provides a control quantity that is used by the digital controller. The *advanced_microswitch* model is a more detailed model when compared to the *simple_microswitch* model since it also models switch resistances, and an additional response time.

In this example, the result of an additional response time within the sensor can be simulated using the *advanced_microswitch* model. This is explained further in "Results - Alarm System" on page 4-88 .



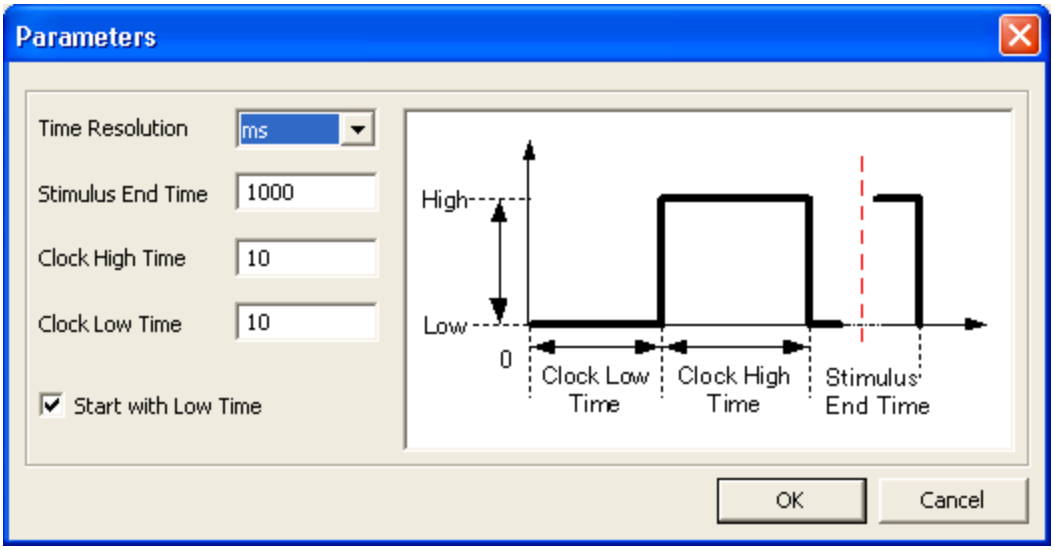
Model: Stimulus Generator

The stimulus generator is used to provide one or more digital stimuli of types BIT, STD_LOGIC, BIT_VECTOR and STD_LOGIC_VECTOR. It provides an easy user interface for specifying the characteristics of digital stimuli and automatically generates VHDL source code for the user-specified stimuli. This model is available from the **Digital Sources** folder in the **Digital Elements** library.

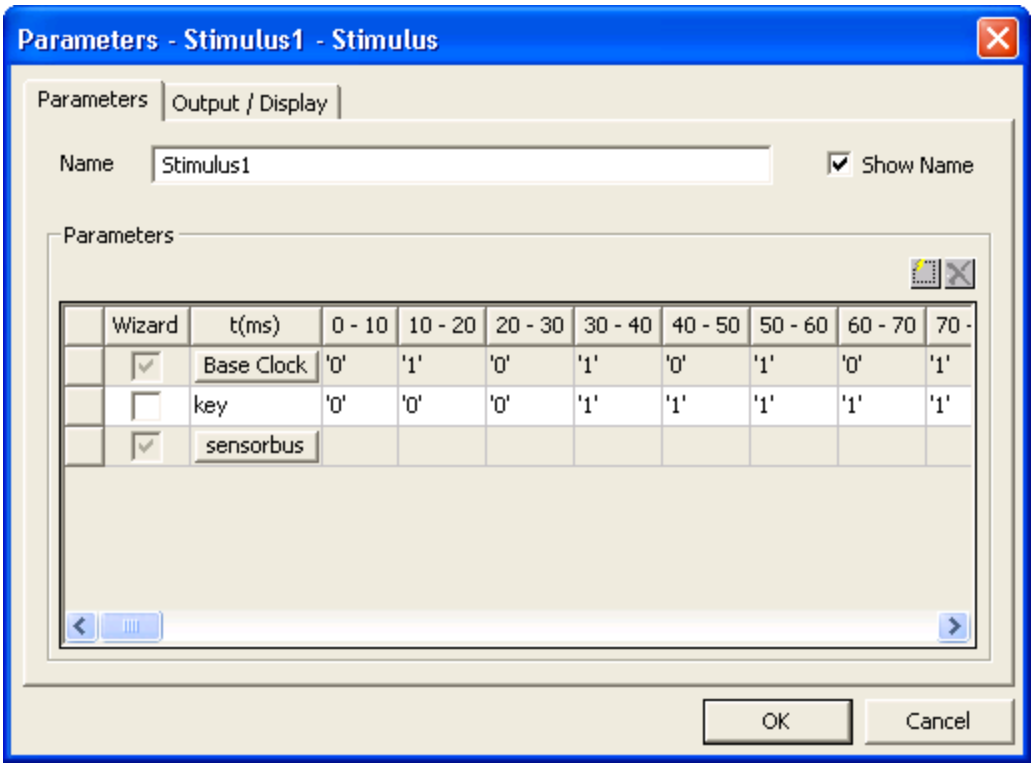
The first time the Properties dialog of the stimulus generator model is opened, a time initialization dialog is displayed. The stimulus generator uses a base clock signal as a reference for creating events for other signals. Users can specify the time resolution, stimulus end time, and base clock characteristics in this dialog box. After exiting the time initialization dialog box, users can set up one or more individual digital stimuli that are required in the Schematic.

The events on each stimulus can be specified by setting the value of the signal on a time line or by specifying a pattern for the stimulus.

In this example, the stimulus generator is used to provide the key switch input as well as vector input from four sensors to the digital controller of the alarm system. To parameterize the stimulus generator, right-click the model and select **Show Component Dialog**. The **Base Clock** dialog box is parameterized with the following values:



Click the Add button to add two additional parameters. Add a parameter of type BIT and change the name to *key*. Add a bit vector parameter and change the name to *sensorbus*.



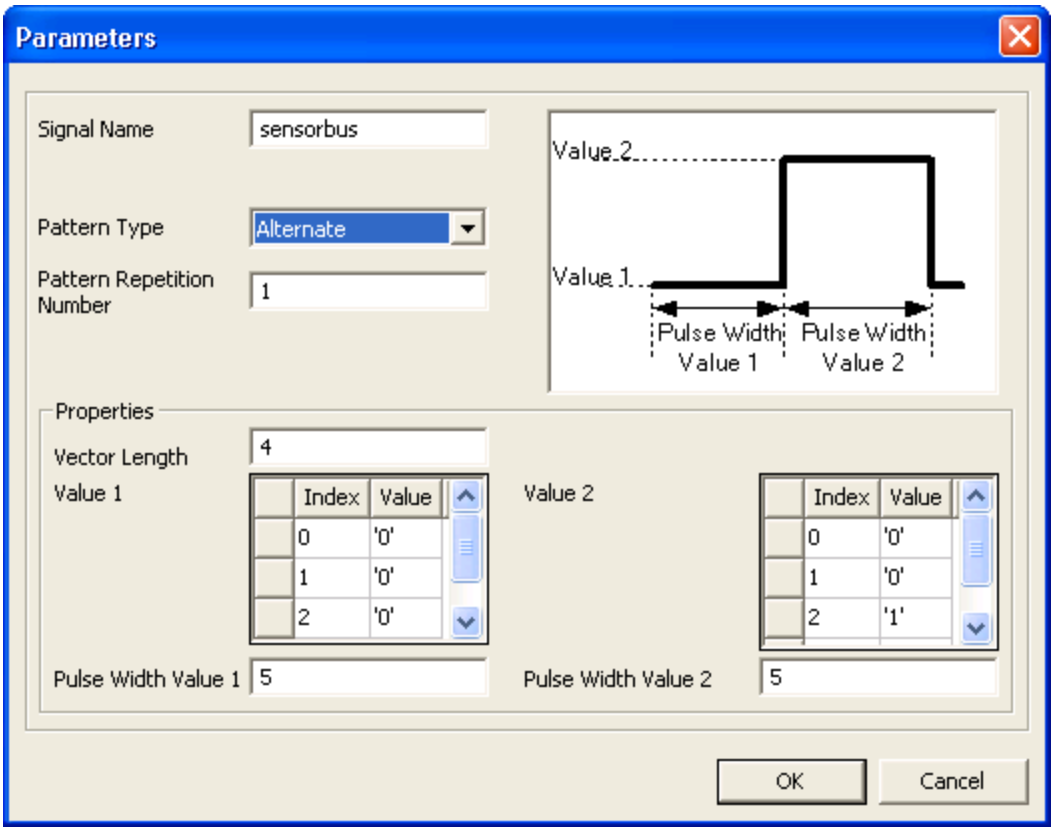
The key switch input is modeled as a BIT signal with signal events defined as follows:

Time(ms)	0-30	30-250	250-340	340-550	550-620	620-850	850-910	910-1000
Values	0	1	0	1	0	1	0	1

Set the values from the drop-down menu for each 10 ms time interval of the *key* parameter according to the table above.

The sensor bus input is modeled as a BIT_VECTOR with signal events on the vector defined by an alternate pattern.

Click on the [sensorbus] button to open the pattern wizard. An alternate pattern is used to toggle two sets of values on the bit vector signal. The pattern repetition number specifies the number of times the pattern needs to be repeated. The size of the bit vector is specified in the Vector Length edit box and actual patterns are specified in the Value 1 and Value 2 section. The period for which each pattern should be applied on the bit_vector signal is specified in terms of the number of base clock pulses, in the Pulse Width Value 1 and Value 2 sections.



Model: Digital Controller

The digital controller accepts a key switch input used to arm and disarm the vehicle, as well as inputs from the different sensors monitoring the security of the vehicle. Two position sensor inputs are available from the two microswitch models and a sensor bus input is available from the stimulus generator model. If any of the sensors detect a security infringement when the system is armed, the digital controller sends a siren output signal.

The logic for the digital controller is modeled as follows. The system is armed by providing an enable (a digital 1 bit) in the key switch input. The controller requires 30 ms to arm the system. Once the system is armed, triggering any of the sensors would cause the siren to turn on after another 30 ms. If the siren has been turned on then it can be switched off instantaneously by disabling the key switch (a digital 0 bit). Note that it is possible to arm a system even if sensors are already triggered (for example, if a door has not been closed properly). In this case, the siren will be turned on 30 ms after the key switch is enabled.

Entity Description - Digital Controller

The digital controller is modeled as a VHDL-AMS text subsheet and the entity declaration for the model is shown below. Note that the siren output is defined as an INOUT signal indicating that its value can be read from as well as written to, inside the model.

```
ENTITY auto_alarm IS
 PORT(
 SIGNAL sensor_bus : IN bit_vector (3 DOWNTO 0) := (OTHERS => '0');
 SIGNAL position_sensor1 : IN bit := '0';
 SIGNAL position_sensor2 : IN bit := '0';
 SIGNAL key_sw : IN bit := '0';
 SIGNAL siren : INOUT bit);
END ENTITY auto_alarm;
```

Architecture Description - Digital Controller

The model architecture has three process statements that are used to co-ordinate between the key inputs and the sensor inputs and provide a warning alarm to the siren output if a security infringement occurs.

The first process statement sets the locally declared BOOLEAN signal *alarm* to TRUE if any of the sensors indicate that there is a problem.

```
PROCESS(sensor_bus, position_sensor1, position_sensor2)
BEGIN
alarm <= sensor_bus(0) OR
sensor_bus(1) OR
sensor_bus(2) OR
sensor_bus(3) OR
position_sensor1 OR
position_sensor2;
END PROCESS;
```

The second process sets the locally declared BOOLEAN signal *armed* to TRUE if the key switch is turned on. Note that there is a 30 ms delay between the switching on of the key switch and the arming of the system.

```
PROCESS
Begin
WAIT ON key_sw;
WAIT FOR 30 ms;
IF (key_sw = '1') THEN
armed <= true;
ELSE
armed <= false;
END IF;
END PROCESS;
```

The third process sets the siren on if the system is armed and an alarm has been detected by the sensors. It incorporates a delay of 30 ms from the time of detection of an alarm in an armed system to the time that a siren is sounded.

```
PROCESS
```

```
BEGIN
WAIT ON armed, alarm;
IF armed AND (alarm = '1') THEN
WAIT FOR 30 ms;
siren <= '1';
END IF;
END PROCESS;
```

The fourth process shuts off the siren if the key switch is turned off. Since the system requires 30 ms for key switch inputs to have an effect on the system (refer to the second process), it takes 30 ms for the siren to turn off after the key switch has been turned off.

```
PROCESS(armed, siren)
BEGIN
IF siren = '1' AND (NOT armed) THEN
siren <= '0';
END IF;
END PROCESS;
```

Note that even though the statements within a process are sequentially executed, the four process statements themselves are concurrently executed.

Results - Alarm System

In this example scenario, the system behavior is described with the repeated enabling and disabling of the alarm system (using the key switch input pattern) and repeated security infringements (triggering the sensor inputs). The triggering of the sensor inputs may be caused by ajar doors, ajar trunks, breaking of glass, open windows, etc.

The system behavior is simulated for 1 sec, and the effect of the trigger inputs from the different sensors on the digital controller is described below.

Sensor Bus Trigger

Initially, the system is armed (at 30ms), and then one of the sensor bus signals (sensor_bus[2]) is triggered (at 100 ms). This causes the siren to sound (at 130 ms). The siren is turned off when the key switch signal is disabled (at 250 ms).

Sensor 1 Input

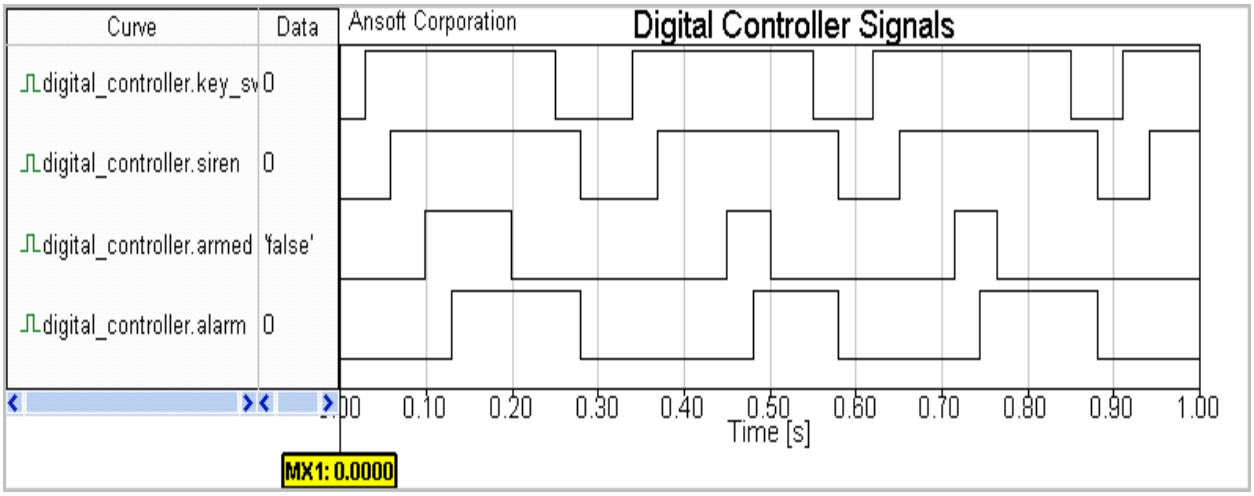
After the system is armed again (at 340 ms), the *position_sensor_1* input to the digital controller input from the *simple_microswitch* sensor model is triggered (at 450 ms). This causes the siren to sound for 100 ms even though the sensor trigger is reset in 50 ms. The siren is turned off when the key switch signal is disabled (at 550 ms).

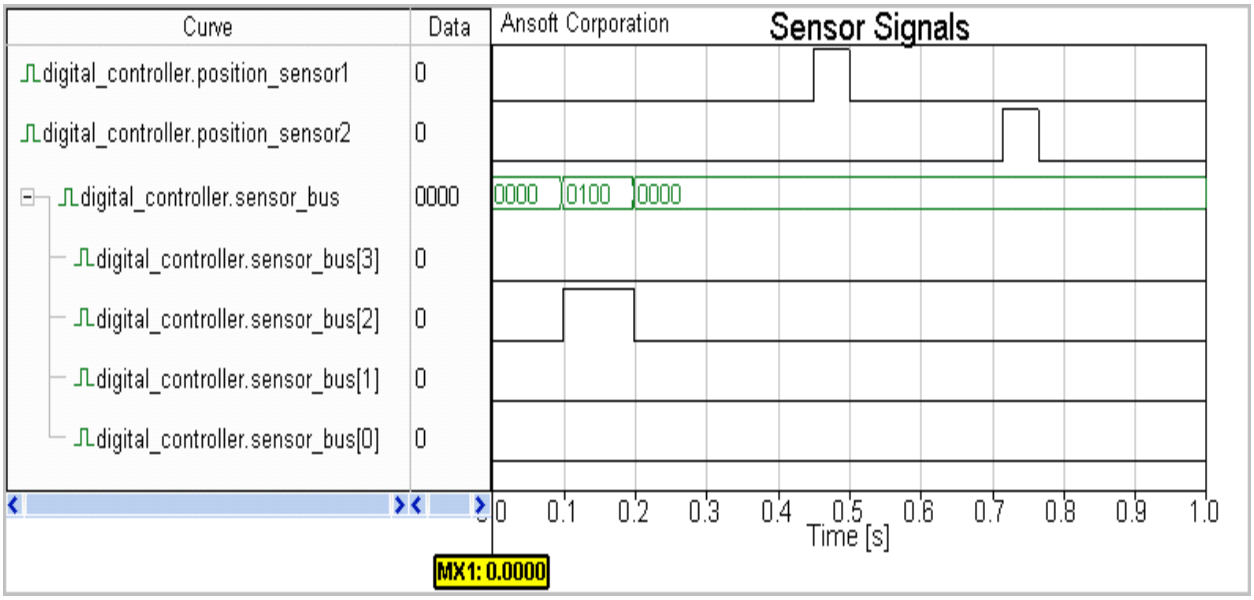
Sensor 2 Input

The system is armed again (at 620 ms) and the *position_sensor_2* input to the digital controller input from the *advanced_microswitch* sensor model is triggered (at 700 ms). However, the internal response time of the sensor model delays the sensor output (to 715 ms). The siren sounds (at 745 ms) till the key switch is disabled at 850 ms.

If the *simple_microswitch* sensor model is used instead of the *advanced_microswitch* sensor for the Sensor 2 Input, then the siren sounds earlier (at 730 ms) due to the absence of an internal delay.

.





VHDL-AMS Export

The standardization of the VHDL-AMS language ensures that models developed in VHDL-AMS are exchangeable among simulators from different vendors. Models created in Twin Builder using VHDL-AMS can be exported to ASCII files for simulation in other VHDL-AMS tools. It is not only possible to export one or more VHDL-AMS models fromSimplorer, but to also export entire sheets with VHDL-AMS models. The export of models is discussed in"Export a VHDL-AMS Component to a Personal Library" on page 5-38 . Exporting sheets to VHDL-AMS netlists is demonstrated in this case study example.

Sheets can be built with VHDL-AMS models from model libraries and subcircuits. These schematics can be exported to “pure” VHDL-AMS netlists that can be used in other simulation tools. Sheets that not only contain VHDL-AMS models but also internal/SML/C (non-VHDL-AMS) models also can be exported to a VHDL-AMS netlist. In this case, the non-VHDL-AMS models are exported as foreign models, compliant with the IEEE 1076.1 standard.

> **Note:**
>
> Netlists containing foreign models can only be simulated in Twin Builder and in most cases cannot be used with other VHDL-AMS simulation tools. To use these netlists in another simulator, the foreign models must be manually replaced in the code by equivalent foreign models that the target simulator can use.

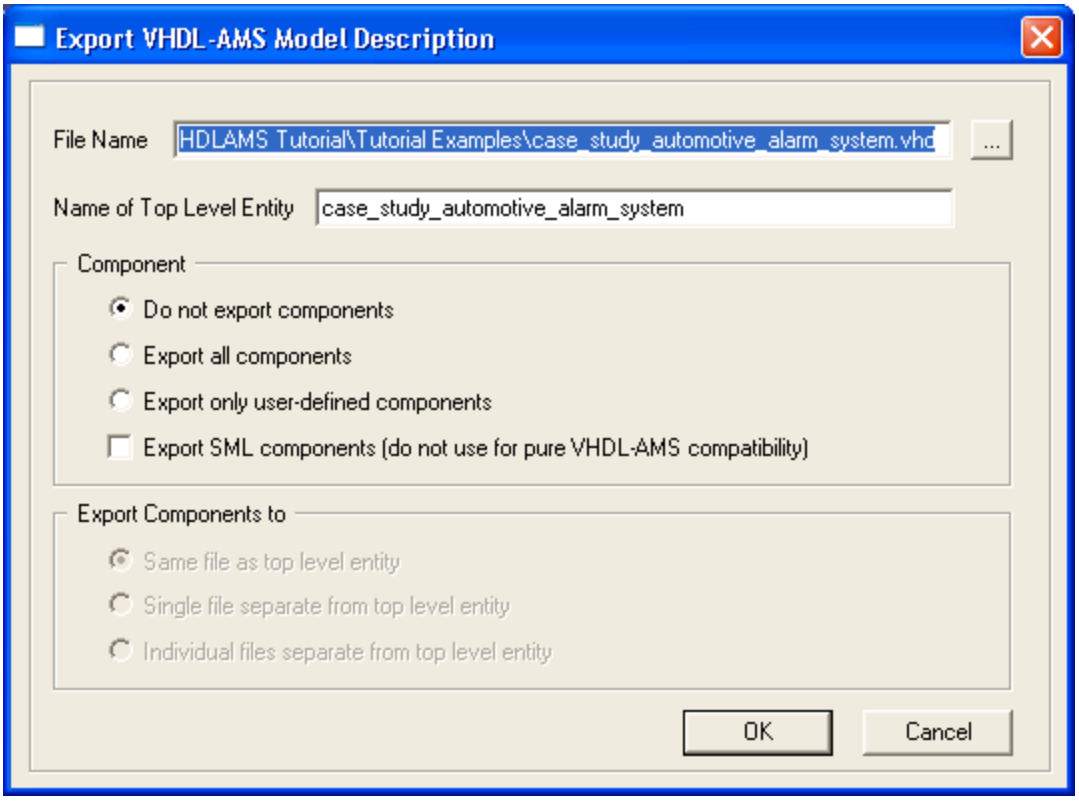
The Automobile Alarm System case study sheet has the following types of models:

- VHDL-AMS models from multiple libraries
- VHDL-AMS text subsheet models
- Subcircuit model with SML sensor model and VHDL-AMS models
- Transformation models for signal-quantity conversions
- Stimulus generator model that generates VHDL code

Exporting a VHDL-AMS Model

To initiate the export of a Schematic sheet to a VHDL-AMS netlist:

1. Select **Tools** > **Design Tools** > **Export to VHDL-AMS**. The **Export VHDL-AMS Model Description** dialog box appears:



2. Enter the name of the VHDL-AMS file that will contain the exported code of the top-level entity. Click on ... to browse to a new location if necessary. This file will be given the **.vhd** extension.
3. A suggested name is shown in **Name of Top Level Entity**, which specifies the top level entity name in the exported VHDL-AMS netlist. The top-level entity instantiates the components on the sheet and provides the interconnection information between the components. The suggested name can be modified if necessary, but the name should

conform to VHDL-AMS identifier syntax requirements (see "VHDL-AMS Language Fundamentals" on page 6-1 for more details). If the name does not conform to the VHDL-AMS identifier syntax requirements, a dialog box appears with the message “Please enter a valid name for the top level entity.”

Apart from the top-level entity, it is also possible to export the VHDL-AMS code of the individual library models that are used in the sheet. For example, in this case study, the VHDL-AMS code for the step function block, summation block, position source models, and so on can be exported. To export individual models, choose either one of the following options from the **Components** panel:

Option	Description
Export all components	Exports the source code of all the VHDL-AMS models on the Schematic.
Export only user-defined components	Exports the source code of only those models that are not available in the pre-installed Twin Builder libraries.

When either of these options is chosen, the **Export Components to** panel titled is enabled. This panel allows the user to specify the target location of the source code belonging to the individual models on the sheet.

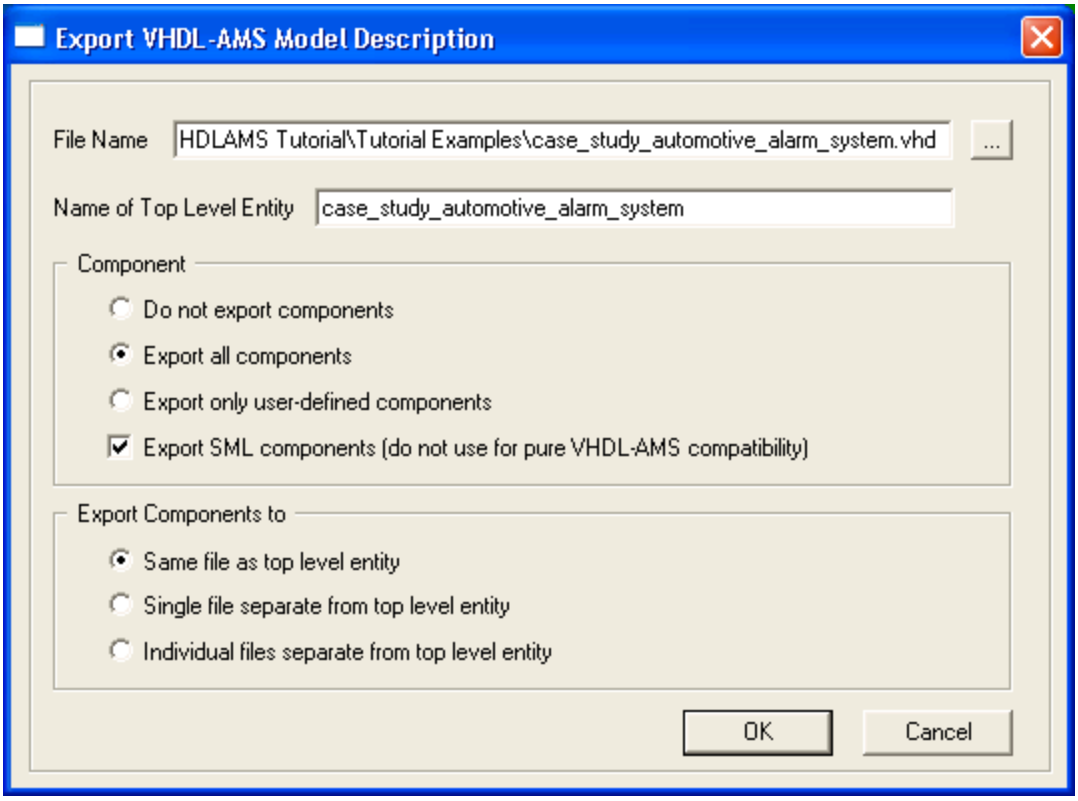
Option	Description
Same file as top-level entity	Generates one VHDL-AMS file which contains the source code of all the models as well as the description of the top-level entity.
Single file separate from top-level entity	Generates the source code of the models in a file called file_name_ Components.vhd where file_name refers to the name of the file provided earlier.
Individual files separate from top-level entity	Creates several VHDL-AMS files separate from the file containing the top-level netlist. Each of these files contains the VHDL-AMS description of a single model used in the sheet. The file name is derived from the name of the entity that the file contains.

A sheet can contain models that are not developed in VHDL-AMS, but are instead developed as internal/SML/C models. These models can also be exported to a netlist as foreign models with a VHDL-AMS wrapper.

> **Note:**
>
> Foreign models generated by Twin Builder can only be simulated in Twin Builder.

To allow VHDL-AMS wrappers to be created around non-VHDL-AMS models, select the **Export SML components** check box. If the sheet contains non-VHDL-AMS models and this check box is not selected, then the export operation will not be completed.

For this example, choose to export all components. Since the advanced microswitch model contains an SML component, check the box to export SML components. Choose to export all components to the same file as the top-level entity.



The Message Manager shows messages from the export operation. The following code is generated for the top-level entity:

```
LIBRARY IEEE;

USE IEEE.std_logic_1164.all;

LIBRARY transformations;
```

```
USE transformations.omnicaster_package.all;
USE IEEE.mechanical_systems.all;
USE IEEE.math_real.all;
USE IEEE.fundamental_constants.all;
USE IEEE.electrical_systems.all;
LIBRARY STD;
LIBRARY basic_vhdlams;

ENTITY case_study_automotive_alarm_system IS
BEGIN
END ENTITY case_study_automotive_alarm_system;
```

```
ARCHITECTURE struct OF case_study_automotive_alarm_system IS
 TERMINAL N0035 : TRANSLATIONAL;
 QUANTITY Q_1 : VELOCITY := 0.0;
 TERMINAL N0039 : TRANSLATIONAL;
 QUANTITY Q_2 : VELOCITY := 0.0;
 QUANTITY Q_3 : REAL := 0.0;
 QUANTITY Q_4 : VOLTAGE := 0.0;
 SIGNAL S_1 : bit := bit'LEFT;
 SIGNAL S_2 : bit := bit'LEFT;
 SIGNAL S_3 : bit := bit'LEFT;
 SIGNAL S_4 : bit := bit'LEFT;
 SIGNAL S_5 : bit := bit'LEFT;
 SIGNAL S_6 : bit := bit'LEFT;
 SIGNAL S_7 : bit := bit'LEFT;
 SIGNAL S_8 : bit := bit'LEFT;
BEGIN
```

```
 Stimulus1: ENTITY WORK.stimulus1

 GENERIC MAP ( bvrsig1_val2_period => 5 , bvrsig1_val1_period => 5 ,
bvrsig1_len => 4 , bvrsig1_num => 1 , stim_end => 1.000000e+000 ,
bc_low => 1.000000e-002 , bc_high => 1.000000e-002 )

 PORT MAP ( key => S_1 , bvrsig1(3) => S_4 , bvrsig1(2) => S_5 ,
bvrsig1(1) => S_6 , bvrsig1(0) => S_7 , clk => S_8 );

 OmniCaster1 : ENTITY transformations.realqty_bitsig(behav)

 GENERIC MAP ( thres => 5.000000e-001 )

 PORT MAP ( val => S_3 , inp => Q_3 );

 OmniCaster2 : ENTITY transformations.realqty_bitsig(behav)

 GENERIC MAP ( thres => 5.000000e-001 )

 PORT MAP ( val => S_2 , inp => Q_4 );

 digital_controller: ENTITY WORK.auto_alarm

 PORT MAP ( key_sw => S_1 , position_sensor2 => S_2 , position_
sensor1 => S_3 , sensor_bus(3) => S_4 , sensor_bus(2) => S_5 ,
sensor_bus(1) => S_6 , sensor_bus(0) => S_7 );

 advanced_microswitch: ENTITY WORK.Macro1

 PORT MAP ( N0044 => N0039, ctrl => Q_4 );

 step2 : ENTITY basic_vhdlams.step(behav)

 GENERIC MAP ( y0 => 0.000000e+000 , ts => 0.000000e+000 , t0 =>
5.000000e-001 )

 PORT MAP ( val => Q_2 , ac_phase => 0.000000e+000 , ac_mag =>
1.000000e-003 , ampl => 1.000000e+000 );

 simple_microswitch1 : ENTITY WORK.simple_microswitch(behav)

 GENERIC MAP ( position_threshold => 5.000000e-004 )

 PORT MAP ( mech_input => N0035, control_output => Q_3 );

 step1 : ENTITY basic_vhdlams.step(behav)
```

```
 GENERIC MAP ( y0 => 0.000000e+000 , ts => 0.000000e+000 , t0 =>
4.500000e-001 )

 PORT MAP ( val => Q_1 , ac_phase => 0.000000e+000 , ac_mag =>
1.000000e-003 , ampl => 1.000000e+000 );


 vel_src2 : ENTITY basic_vhdlams.v_trb(behav)

 GENERIC MAP ( s0 => 0.000000e+000 )

 PORT MAP ( trb1 => N0039, ac_phase => 0.000000e+000 , ac_mag =>
1.000000e-003 , value => Q_2 );


 vel_src1 : ENTITY basic_vhdlams.v_trb(behav)

 GENERIC MAP ( s0 => 0.000000e+000 )

 PORT MAP ( trb1 => N0035, ac_phase => 0.000000e+000 , ac_mag =>
1.000000e-003 , value => Q_1 );

END ARCHITECTURE struct;
```

The exported netlist includes all the libraries and associated packages required for this example. The top-level entity has an empty interface since it represents the top-level sheet. The architecture for this entity declares the terminals, quantities and signals used for interconnections in this example. The architecture body instantiates the components used in the example.

Foreign Models

The SML sensor model used in the advanced microswitch subsheet can be exported as a foreign model. Foreign models contain instantiations of non-VHDL-AMS models within VHDL-AMS wrappers, and cannot be exchanged among simulators. The following figure shows the code snippet of the wrapper that is created around the sensor model.

```
-- ********** Foreign Model Wrapper Start **********

LIBRARY basic_vhdlams, digital_elements, ieee, transformations;
USE basic_vhdlams.all, digital_elements.all, ieee.all, transformations.all;

ENTITY ent_tmswitch_df_l21 IS
      PORT (
            TERMINAL np : ELECTRICAL;
            TERMINAL nmno : ELECTRICAL;
            TERMINAL nmnc : ELECTRICAL;
            TERMINAL ref : TRANSLATIONAL;
            TERMINAL position : TRANSLATIONAL
      );
BEGIN
END ENTITY ent_tmswitch_df_l21;

ARCHITECTURE struct OF ent_tmswitch_df_l21 IS
      COMPONENT comp_tmswitch_df_l21 IS
      PORT (
            TERMINAL np : ELECTRICAL;
            TERMINAL nmno : ELECTRICAL;
            TERMINAL nmnc : ELECTRICAL;
            TERMINAL ref : TRANSLATIONAL;
            TERMINAL position : TRANSLATIONAL
      );
      END COMPONENT;

      ATTRIBUTE sml_model : STRING;
      ATTRIBUTE sml_model OF comp_tmswitch_df_l21 : COMPONENT IS
            "MODEL TMSWITCH_DF_l2 TMSWITCH_DF_l21 POSITION  := position , REF  := ref , nmr
BEGIN
      comp_tmswitch_df_l21_inst : comp_tmswitch_df_l21 PORT MAP ( nmno => nmno, np => np, p
END ARCHITECTURE struct;

-- ********** Foreign Model Wrapper End **********
```

The foreign model wrapper specifies the interface of the sensor model in an entity description. The architecture description of the foreign model specifies a component declaration and instantiation for the sensor model. It also associates an attribute called *sml_model* to the component that specifies the SML call line of the sensor model.

Another example of the use of foreign models can be seen in the source code of the continuous memory block in the *basic_vhdlams* library (Blocks>Continuous>Memory). This model has two architectures called *behav* and *sml_behav*. The second architecture instantiates an SML model as a component and associates two attributes to the component. The attribute *sml_model*

specifies the SML call line and the second attribute *sml_output* specifies output connections for the model.

5 - Model Development

This chapter discusses the use, development, export, and import of VHDL-AMS models in Twin Builder. It is important to first understand basic Library concepts.

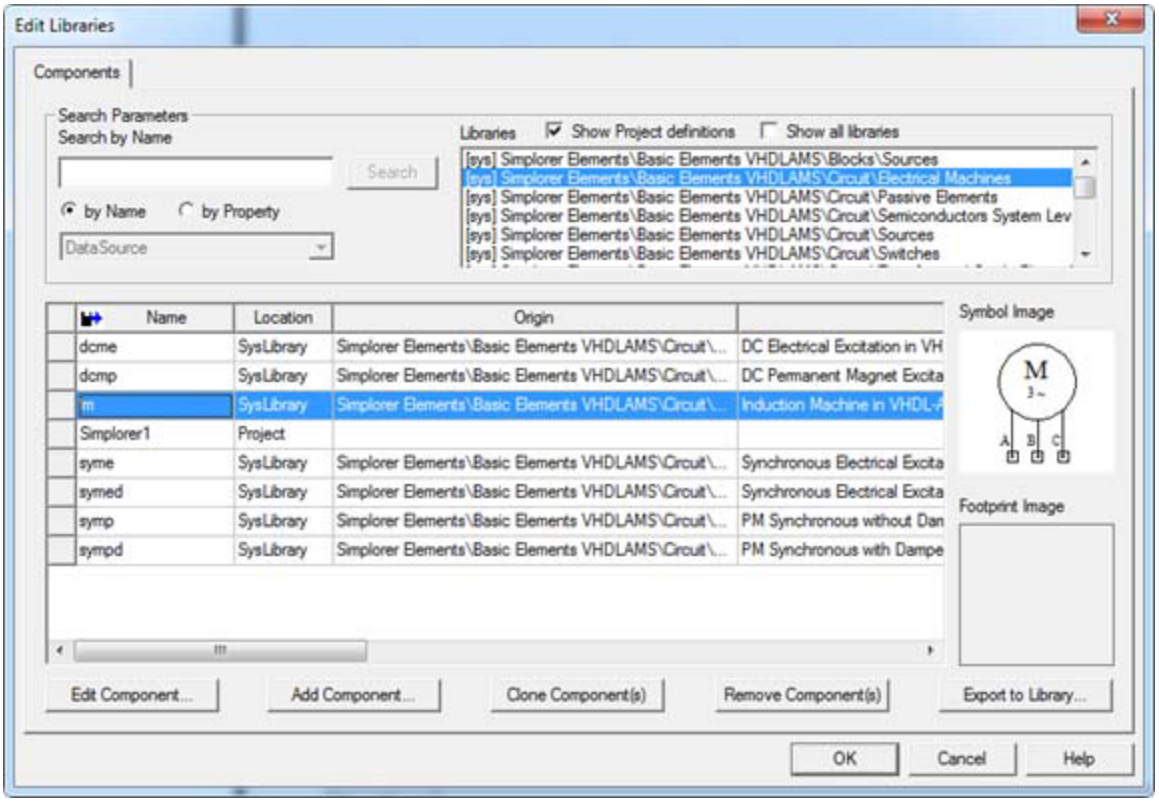
Library Concepts

A component (for example, a resistor, transistor, motor) has several definitions associated with it. It needs a “Symbol” definition to describe the appearance of its symbol, a “Model” definition to define the characteristic behavior of the component (that is, the VHDL-AMS code), and a “Component” definition as a kind of high-level wrapper. There are also other types of definitions depending on the component’s need such as “Material” definitions, or if using VHDL-AMS components, sometimes a “Package” definition is needed.

In Twin Builder projects, library component definitions are accessed once per component (when you place the first instance of a component onto a schematic). Once a component is placed, all the definitions for that component transfer from the library to the project file.

Editing component definitions and updating instances are then controlled from the project definitions, which are listed in the **Definitions** folder in the **Project** tab.

In other words, you do not edit component definitions in the *library*, but in the *project*. If you want your edits to be reflected in the library definition for use in another design, you must export the edited component definitions to the library by using the menu **Tools** > **Edit Libraries** > **Components** to open the **Edit Libraries** dialog box, selecting the component definition that was changed, and clicking the **Export to Library** button to export it.



Library Directories

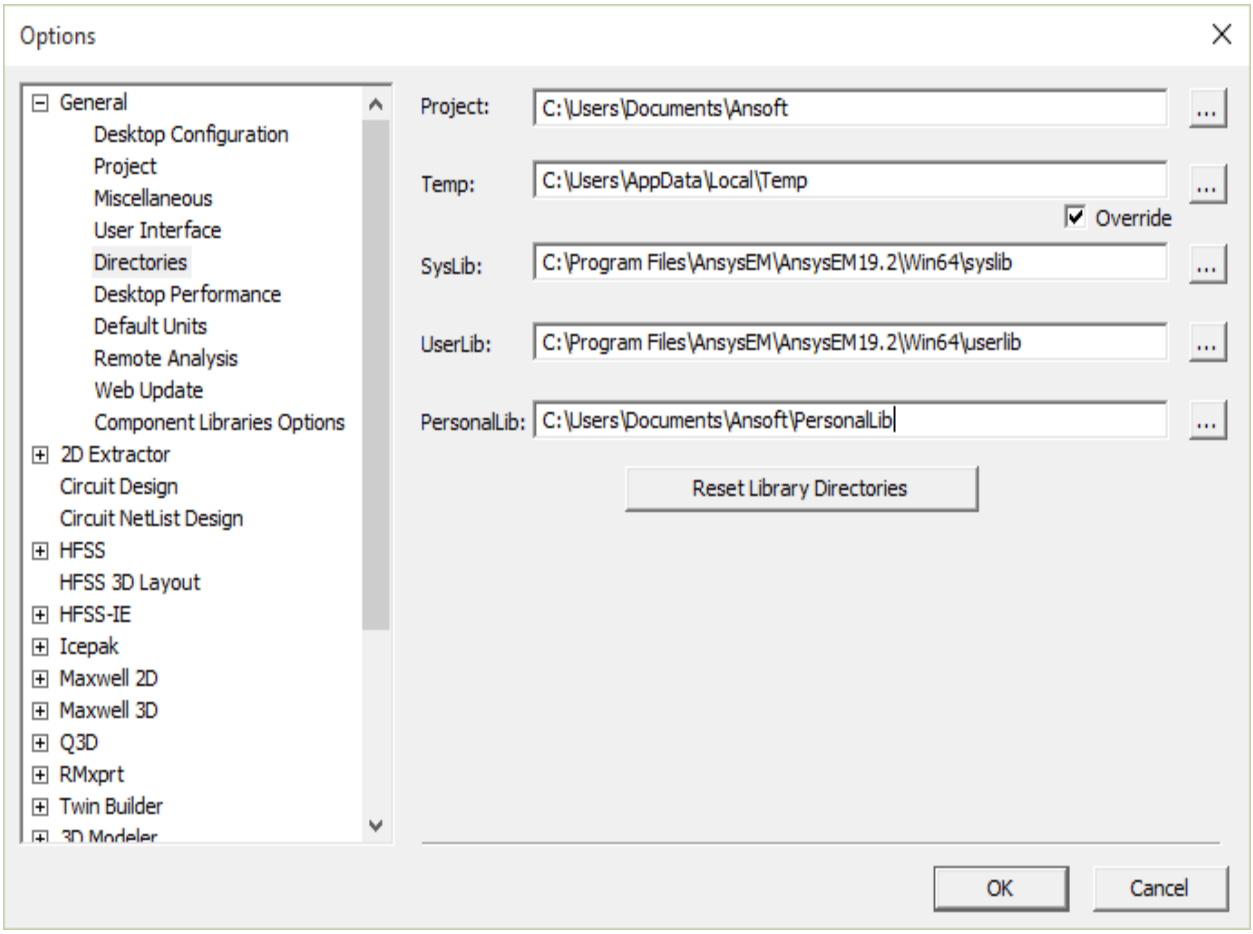
The standard library files that ship with Twin Builder are stored by default in the **/Twin Builder/syslib** directory. These libraries are intended to be read-only. They also include many VHDL-AMS models that can be used in simulation, or saved to a user library where they can be edited.

In addition to the system libraries, Twin Builder recognizes two user-configurable libraries, **userlib** and **PersonalLib**. Customarily, **userlib** is a network repository for proprietary or corporate definitions available to all Twin Builder seats in a company, and **PersonalLib** contains specific libraries as needed by individual users. A root library directory is set up at installation. If none is specified, the default is the root Twin Builder directory.

Specifying the Location of Library Files

You can relocate the library directories to a new location. To specify a new library directory:

1. Select **Tools** > **Options** > **General Options**.
2. Select **General** > **Directories** and type the new folder location in the appropriate box, or use the browse button to specify it.



3. Click **OK**.

Components and Library Search Precedence

When you place a new component in a design, Twin Builder searches in the current project for a definition with the corresponding name. If an appropriate definition exists in the project, that definition is used to satisfy the placement *whether additional instances, even more recent ones, exist in external libraries.*

If Twin Builder cannot locate the required definitions in the project, it searches in the personal (**PersonalLib**), user (**userlib**), and system (**syslib**) library directories, *in that order*. When a definition of the correct name has been located, the search ends, and that definition is used to satisfy the placement request.

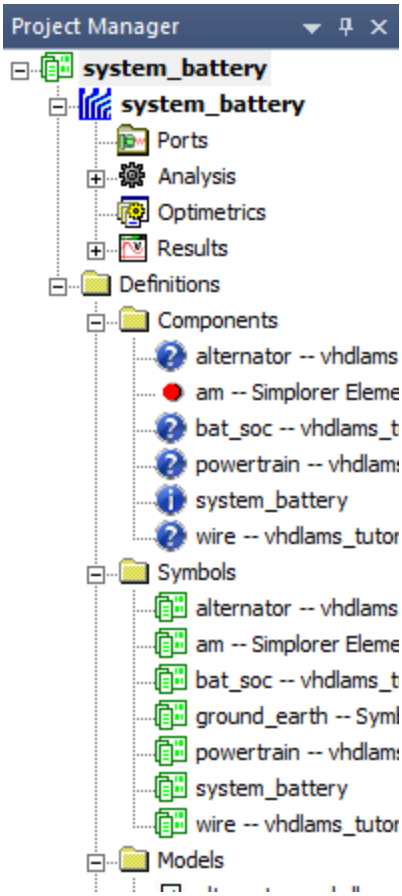
To be included in a search path, an external library (including libraries in **userlib** and **PersonalLib**) must be “configured” to each design within the project that must access it.

> **Note:**
>
> Each design within a project can have a different library configuration.

Definitions

When a definition is used, it is transferred to the current project, and remains in the project unless it is explicitly removed.

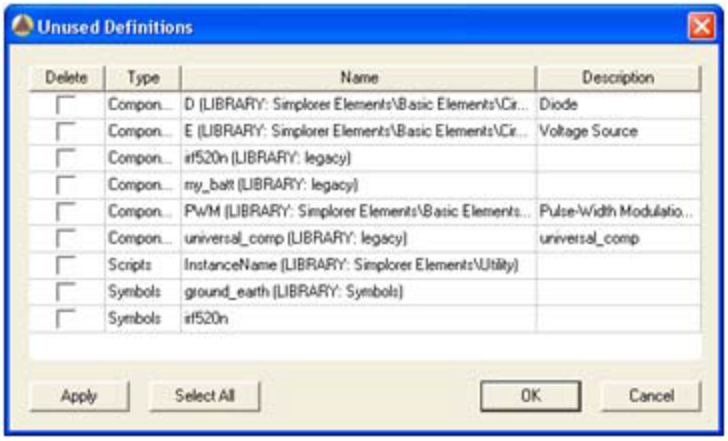
1. To see the definitions included in a project, expand its **Definitions** folders and subfolders in the **Project Manager** pane:



2. Because the project version of a given definition takes precedence over all other instances of that definition – unused definitions still present in a project can lead to unintended results if you do not delete them.
3. If you have removed every usage instance of a definition from a project (that is, that component is no longer used in the schematic), you can remove its **Definitions** entry by

clicking the main menu **Tools** > **Project Tools** > **Remove Unused Definitions** command.

The **Unused Definitions** dialog box opens.



4. Use the dialog box to select definitions to delete. You can select individual definitions using the check boxes next to each, or you can click **Select All** to choose all definitions in the list.
5. When finished selecting definitions to delete, do one of the following:
   a. Click **Apply** to remove the selected items while remaining in the dialog box, allowing you to continue removing any other associated definitions.
   b. Click **OK** to remove the selected items and exit the dialog box.

> **Note:**
>
> Modifying component and dependency definitions in libraries, or installing libraries with modified component and dependency definitions, does not automatically update those definitions in projects that contain them. See ***Updating Project Definitions from Library Definitions***.

Updating Project Definitions from Library Definitions

When a library component is placed into a design, the parent component definition resides in the project, and the link to the library of origin is broken. To update a project component definition with a definition from a library file:

1. Select the project you want to update in the project tree, then select **Tools** > **Update Definitions** from the main menu.

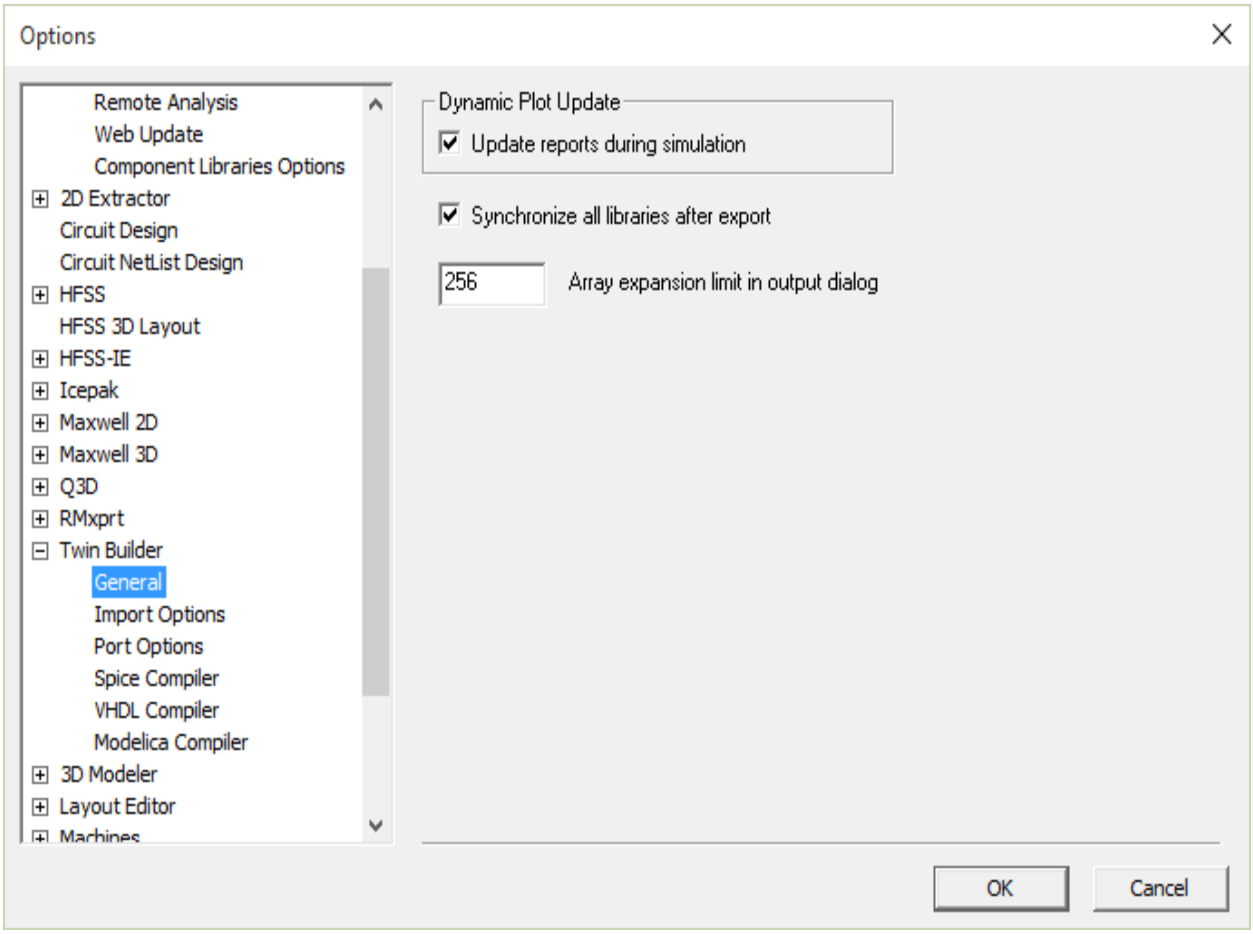
The **Update Definitions** dialog box opens, listing any library definitions that have changed since they were added to the project, and their original library locations.

2. Use the dialog box to select the items to update, and click **Update** to finish.

Model and Library Synchronization

In Twin Builder, model definitions also may be used by other models in their definitions. Consequently, changes made to a model definition affect all models that depend upon (use) that definition. If you update models that are interdependent after making changes to them, there needs to be some process to keep them in sync. Synchronization is the process by which Twin Builder reconciles these changes. You can synchronize project definitions by using **Tools** > **Project Tools** > **Synchronize Definitions**. You can synchronize all libraries on demand by using **Tools** > **Library Tools** > **Synchronize All**.

> **Note:**
>
> Automatic library synchronization after model “export” into a library in Twin Builder is enabled by default in the **Synchronize all libraries after export** check box in **General** in the **Twin Builder** panel in **Tools** > **Options** > **General Options**. This setting allows library models that are dependent on exported models to be synchronized automatically.



Translating Legacy Libraries (Libraries created prior to v8)

Starting with Simplorer 2016 (Ansys Electromagnetics Suite 17.0), Simplorer/Twin Builder no longer supports Simplorer 7.0 schematics. If you want to port a design from Simplorer 7.0, you must first use Simplorer Release 16.2 or earlier to translate it and then bring it into the current release.

Example #1

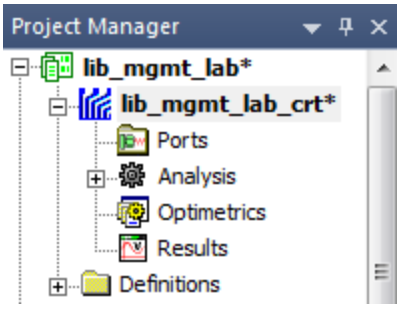
In this example we will:

- Create a new project and design in Twin Builder.
- Create a path for a personal library.
- Create a new VHDL-AMS model and import the new model into Twin Builder.
- Create a symbol for the model.
- Export the VHDL-AMS component to a personal library.

- Configure this new library to be used in the design.
- Create a circuit using the VHDL-AMS component and simulate.

Create a new project and design

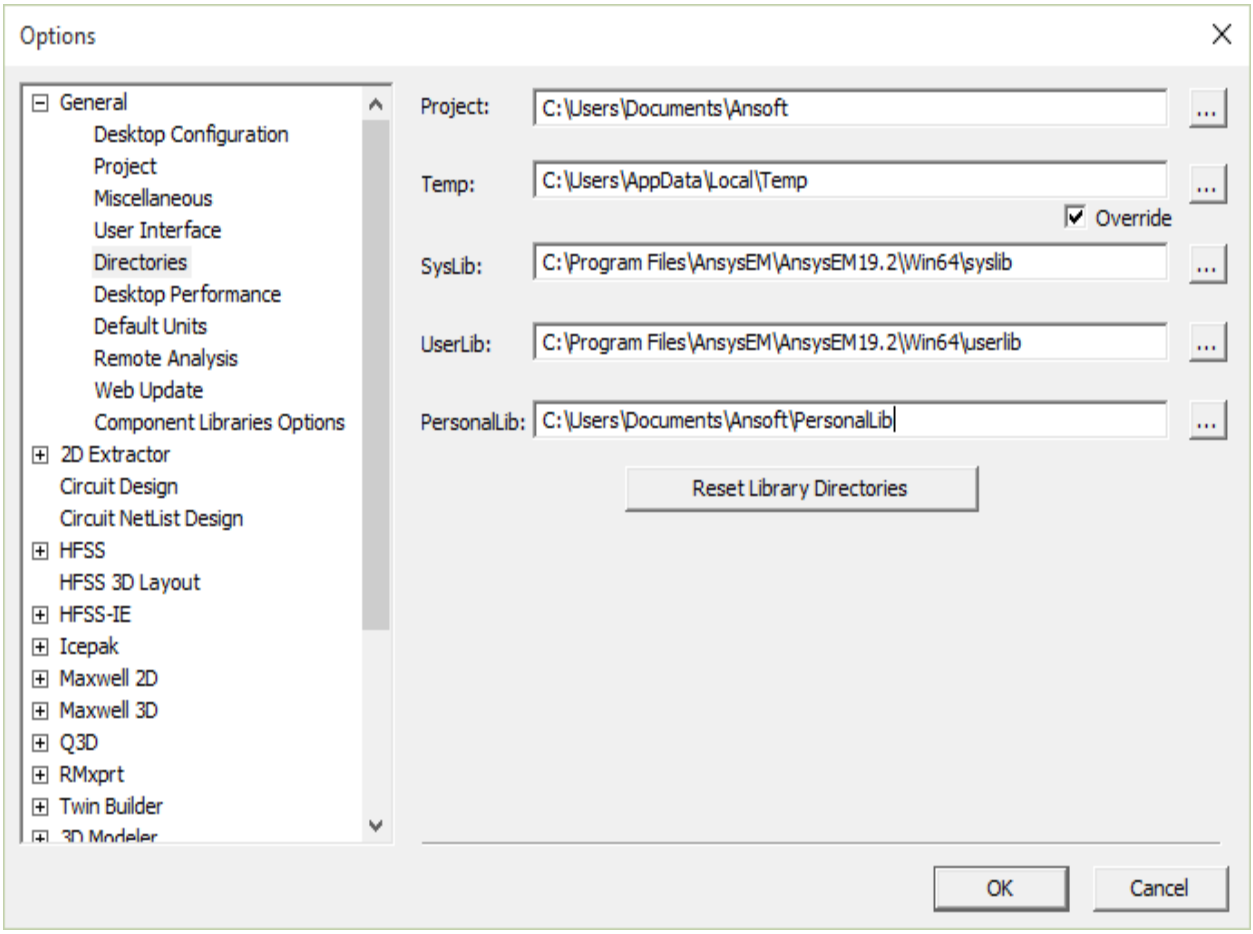
1. Start Twin Builder.
2. Change the project and design name to the following.



3. Select **File** > **Save As** to save the project to your desired location.

Create a new path for your personal library

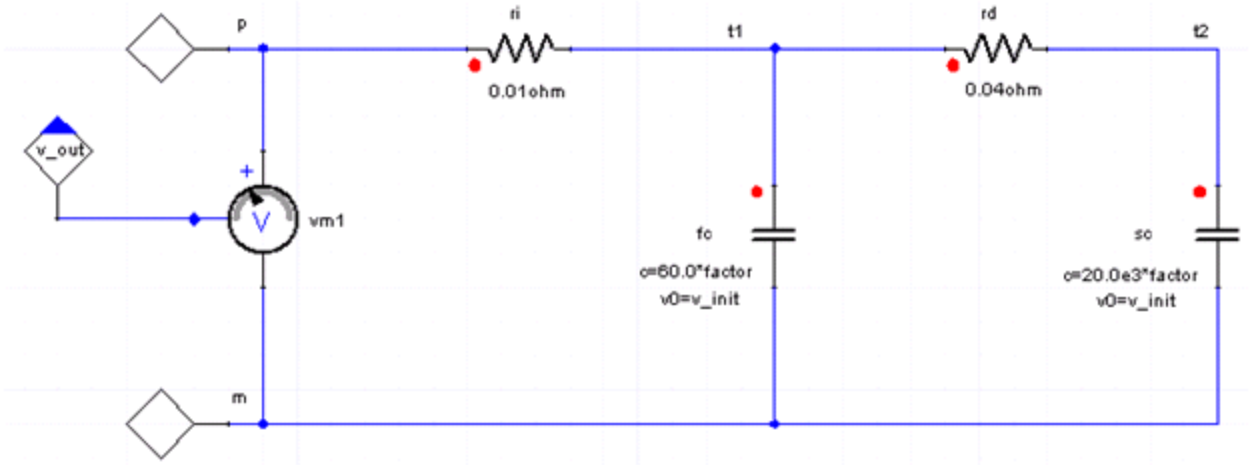
1. Select **Tools** > **Options** > **General Options** on the main menu.
2. Select **General** > **Directories**.
3. Here you can assign paths for the different categories. Do not change the path for the **SysLib** directory. For this example use the default paths already set up. Change the location of the **PersonalLib** to a location you want.



3. Select **OK**.

Create a VHDL-AMS Model using the VHDL-AMS Editor

Create a simple VHDL-AMS model for a battery and name it **my_batt**. This battery model is based on the following:



- Model Inputs: **factor**, and **v_init** (initial voltage charge on the capacitors).
- Model Outputs (non-conservative): **v_out** which represents the output terminal voltage (**p**,**m**).
- Conservative nodes: the **p** and **m** nodes of the battery itself.
- Equations were developed based on circuit theory for the voltages.

The finished code should look similar to the following:

```
---------- VHDLAMS MODEL my_batt ----------
---------- ENTITY DECLARATION my_batt ----------
LIBRARY Ieee;
use Ieee.electrical_systems.all;

ENTITY  my_batt IS
    generic (
        factor : real := 1.0;
        v_init : voltage := 12.0

    );

    port (
        QUANTITY v_out : OUT voltage := 0.0;
        TERMINAL p :  electrical;    --positive node
        TERMINAL m :  electrical     --negative node

    );

END ENTITY my_batt;
```

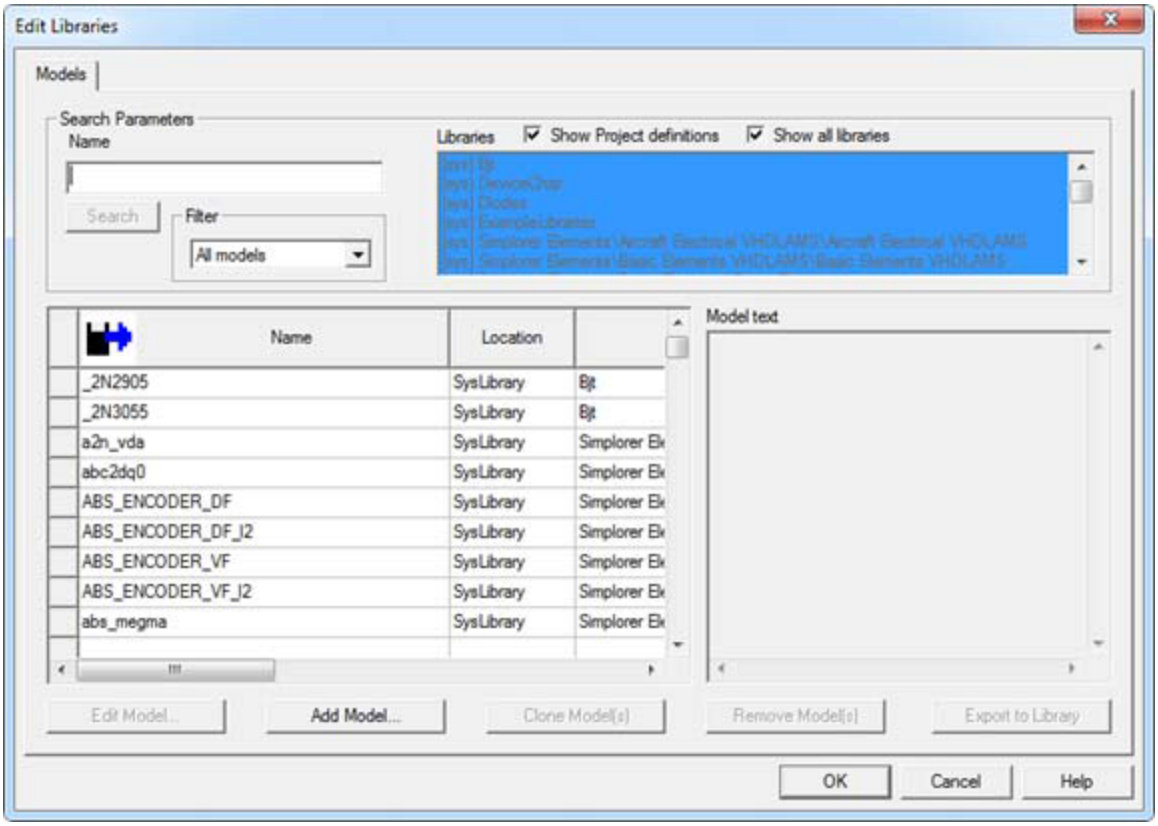
```
---------- ARCHITECTURE DECLARATION arch_my_batt --------
ARCHITECTURE arch_my_batt OF my_batt IS

    TERMINAL t1 :  electrical;
    TERMINAL t2 :  electrical;

    QUANTITY v_ri across p  TO t1;
    QUANTITY v_fc across t1 TO m;
    QUANTITY v_rd across t1 TO t2;
    QUANTITY v_sc across t2 TO m;
    QUANTITY v    across p  TO m;

    QUANTITY i_ri through p  TO t1;
    QUANTITY i_fc through t1 TO m;
    QUANTITY i_rd through t1 TO t2;
    QUANTITY i_sc through t2 TO m;

  CONSTANT ri: RESISTANCE := 1.0e-2;
  CONSTANT fc: CAPACITANCE := 60.0;
  CONSTANT rd: RESISTANCE := 4.0e-2;
  CONSTANT sc: CAPACITANCE := 2.0e4;

BEGIN
  BREAK v_fc => v_init, v_sc => v_init;

  v_ri == i_ri * ri;
  v_fc'dot == 1.0/(fc*factor) * i_fc;
  v_rd == i_rd * rd;
  v_sc'dot == 1.0/(sc*factor) * i_sc;
  v_out == v;

END ARCHITECTURE arch_my_batt;
```

Add a VHDL-AMS Model

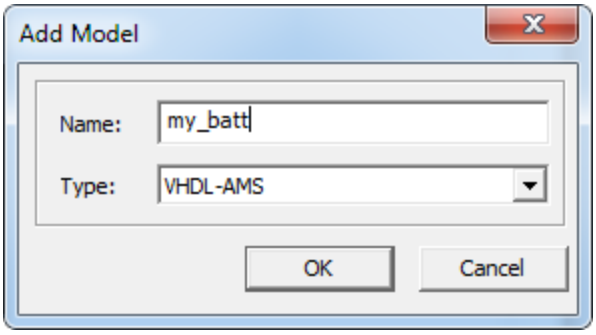
The following procedure outlines the steps for adding a VHDL-AMS model to a project using Twin Builder's VHDL-AMS Model Editor.

1. On Twin Builder's menu bar, select **Tools** > **Edit Libraries** > **Models**.

   The **Edit Libraries** dialog box appears.



2. Click **Add Model** to open the **Add Model** dialog box.
3. Select **VHDL-AMS** from the **Type** pull-downlist.
4. Type the new VHDL-AMS model name (**my_batt**) in the **Name** field, and click **OK**.

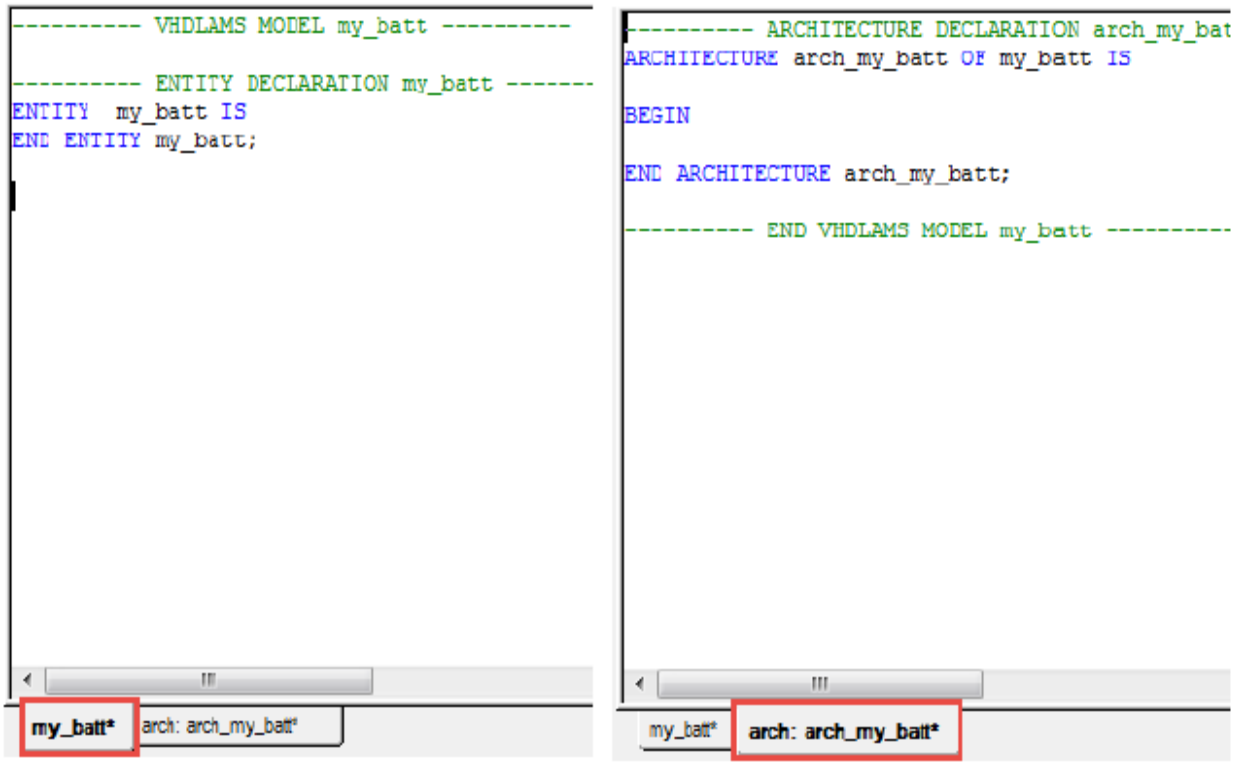


> **Note:**
>
> Twin Builder also gives you the choice of adding the new model directly to the current project (for later editing, perhaps).

Twin Builder creates a new VHDL-AMS model entity and architecture declarations bearing the chosen model name, and adds the model to the Project Manager **Definitions>Models** folder.

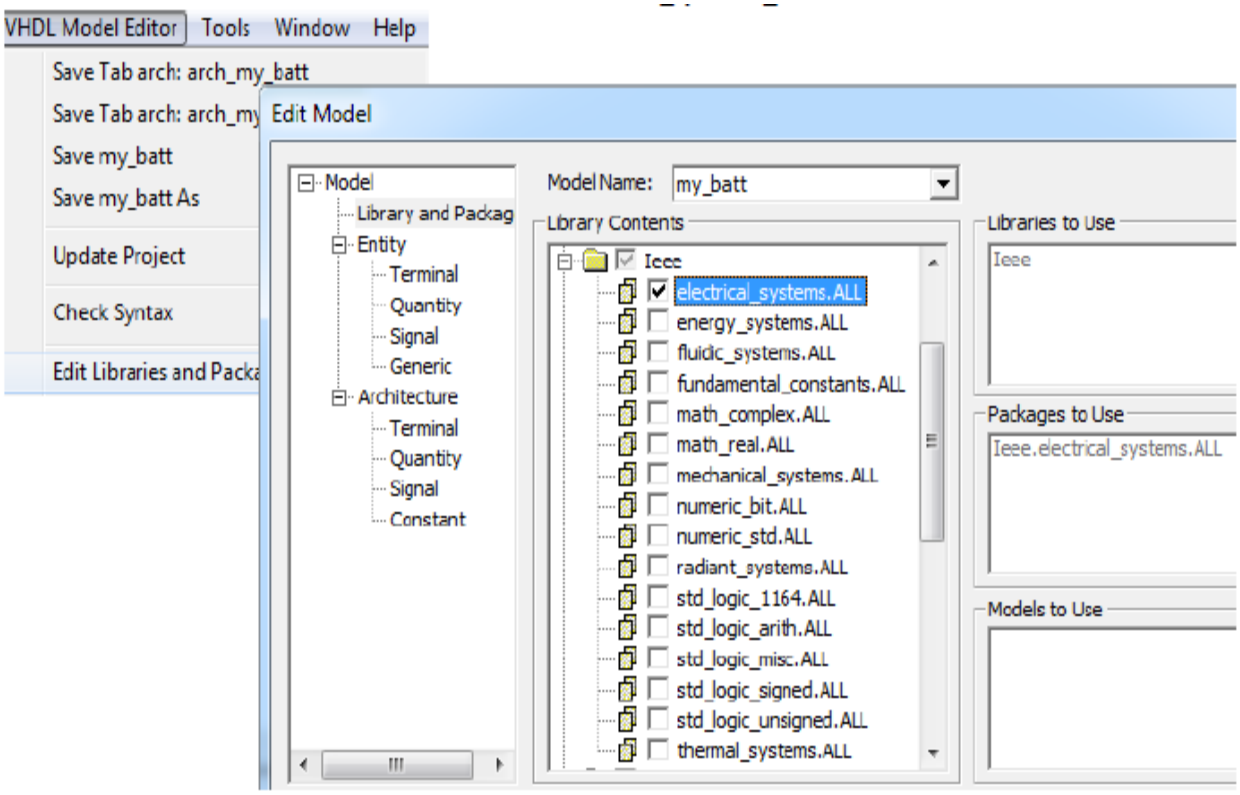
5. You can open the model for editing from the Project Manager by double-clicking the model, or by right-clicking the model and selecting **Edit Model**.

   When the VHDL-AMS Model Editor opens, it displays the new VHDL-AMS model entity and architecture declarations in separate tabs bearing the model name (**my_batt**) you chose.

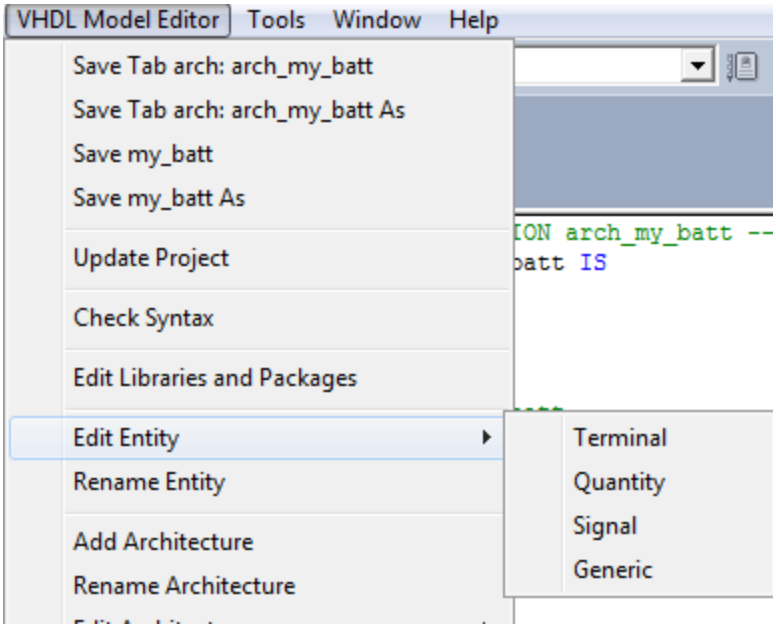


6. Edit the VHDL-AMS model source code as follows:
   a. First, add the libraries to be used by the model. When the libraries are selected, VHDL-AMS code is automatically generated.

      Select the **leee** box and the **electrical_systems_all** sub box; then select **OK**.

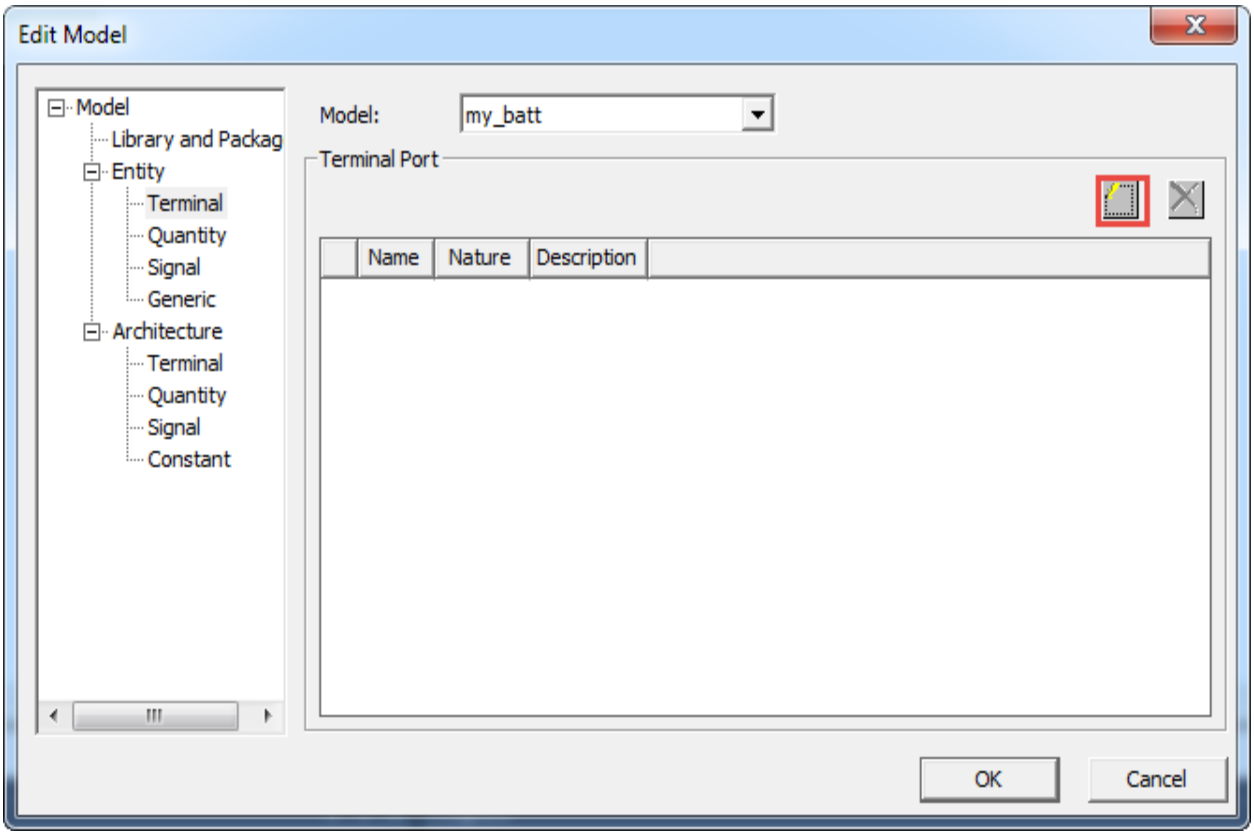


b. Next, edit the **Entity** using the menu choices in **VHDL Model Editor** > **Edit entity** (view the finished code as we go through this process).

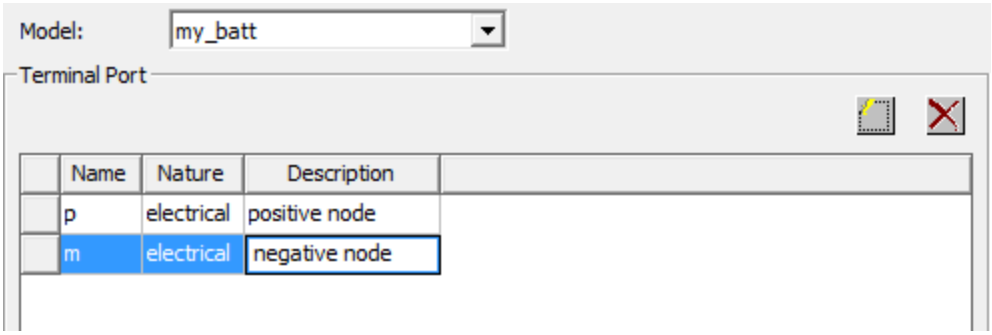


c. Select **VHDL Model Editor** > **Edit Entity** > **Terminal**.

d. Select the input button.



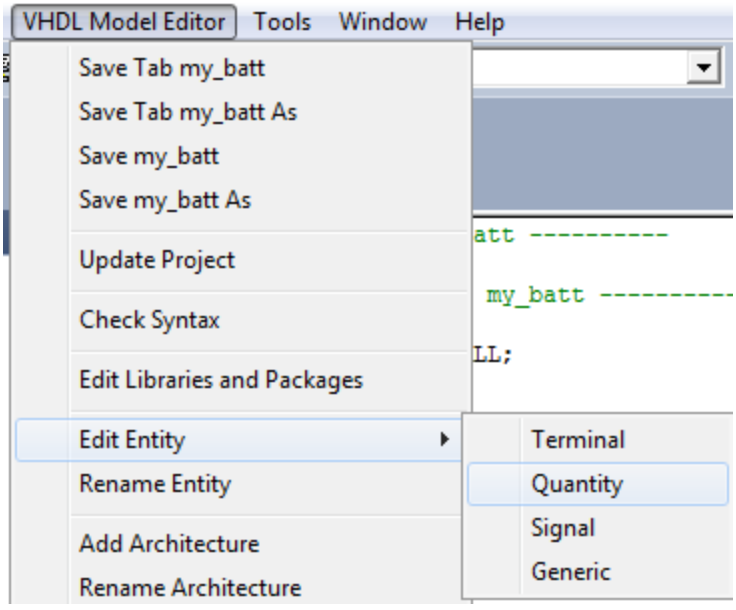
e. Name the two ports (“**p**” and “**m**”) and define them as **electrical** via the drop-down menu (the port options – **electrical** or **magnetic** - are defined by the library that was inserted in the first step). You can also define descriptions if desired (for example, positive node, negative node).



```
---------- VHDLAMS MODEL my_batt ----------

---------- ENTITY DECLARATION my_batt ----------
LIBRARY Ieee;
use Ieee.electrical_systems.all;
ENTITY  my_batt IS
    port (
        TERMINAL p :  electrical;   --positive node
        TERMINAL m :  electrical    --negative node

    );

END ENTITY my_batt;
```

f. Select **OK** to generate the code.

g. Add a **Quantity** to the **Entity** for the non-conservative output **v_out** (to represent the output voltage of the battery).

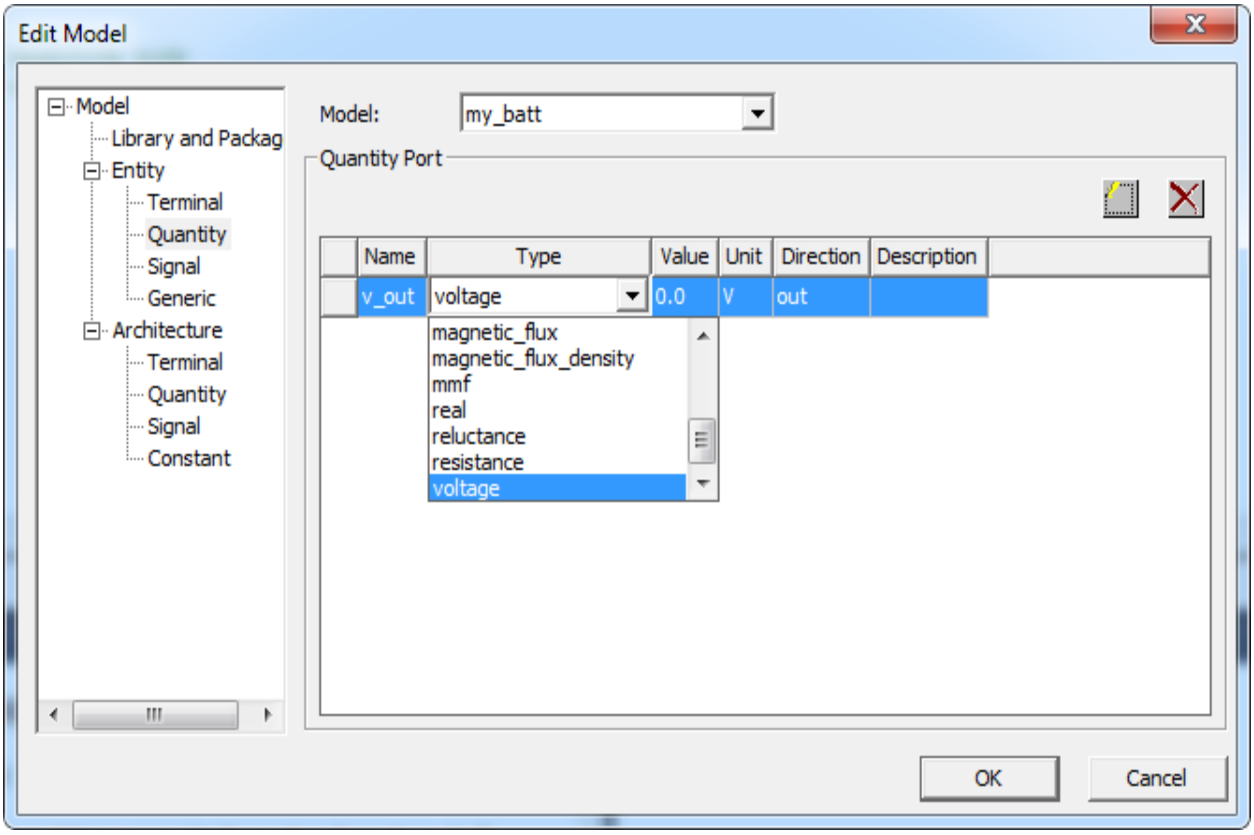


h. Select the input button  to add a Quantity.

i. Name it **v_out**.

j. Select the **Type** via the drop-down menu to be **voltage**.

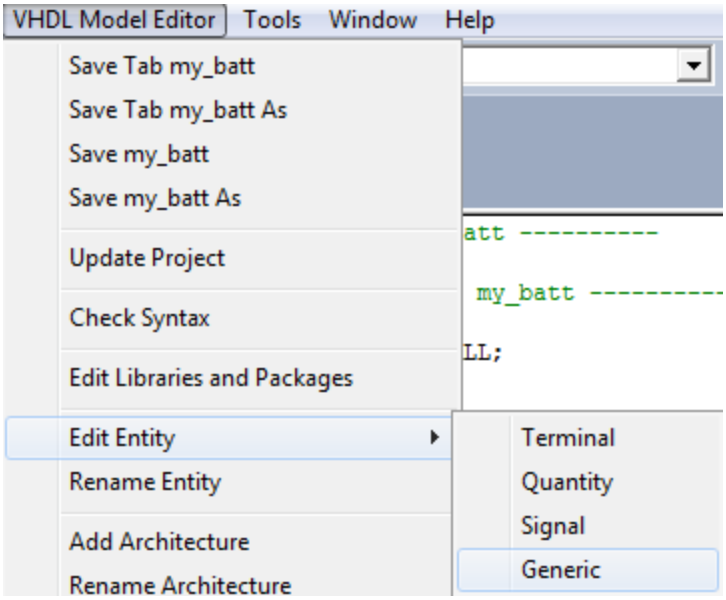
k. Give it a default value of **0.0**, make the direction **out**,vthen click **OK**.



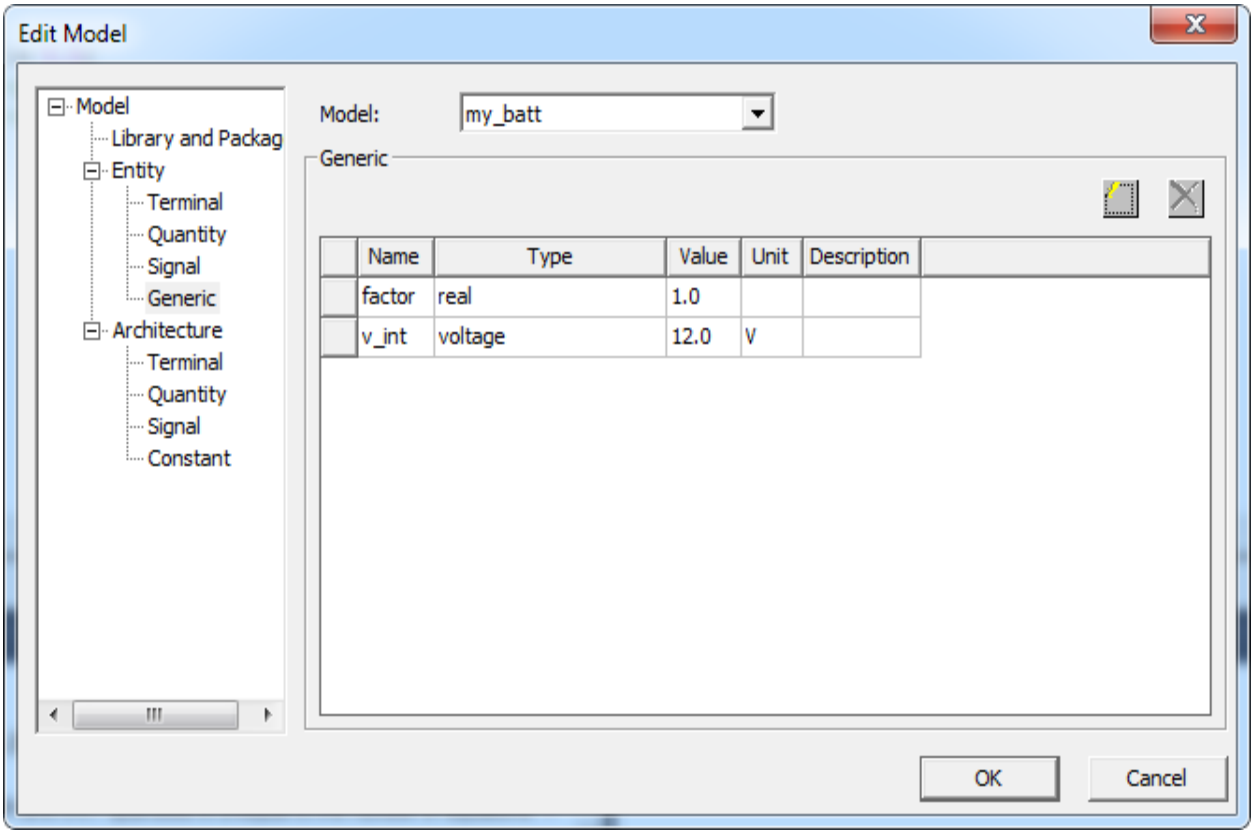
This generates the following addition to the code:

```
---------- VHDLAMS MODEL my_batt ----------

---------- ENTITY DECLARATION my_batt ----------
LIBRARY Ieee;
use Ieee.electrical_systems.all;
ENTITY  my_batt IS
    port (
        QUANTITY v_out : OUT voltage := 0.0;
        TERMINAL p :  electrical;    --positive node
        TERMINAL m :  electrical     --negative node

    );

END ENTITY my_batt;
```

7. Select **Generic** to add the inputs **factor** and **v_init**:



8. Select the input button to insert two items.
9. Create the following. Note that **v_init** is the initial voltage charge on the battery with a default value of **12.0v**.

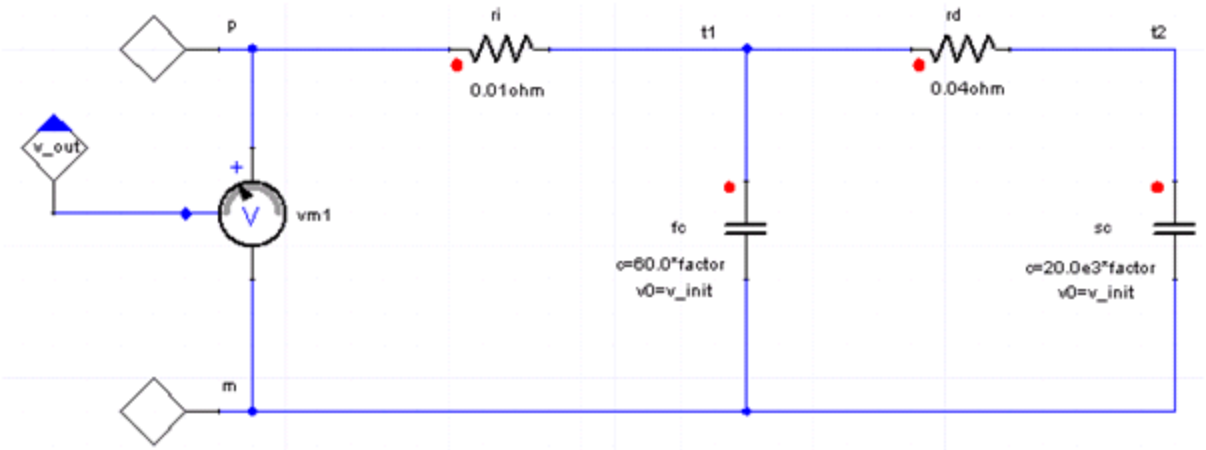


10. Click **OK**.

    This generates the final addition to the “Entity” part of the model.

```
---------- VHDLAMS MODEL my_batt ----------

---------- ENTITY DECLARATION my_batt ----------
LIBRARY Ieee;
use Ieee.electrical_systems.all;
ENTITY my_batt IS
    generic (
        factor : real := 1.0;
        v_init : voltage := 12.0

    );

    port (
        QUANTITY v_out : OUT voltage := 0.0;
        TERMINAL p :  electrical;    --positive node
        TERMINAL n :  electrical     --negative node

    );

END ENTITY my_batt;
```

11. Now add the **Architecture** information to the VHDL-AMS model that defines the behavior. Note this model is based on the following circuit description:

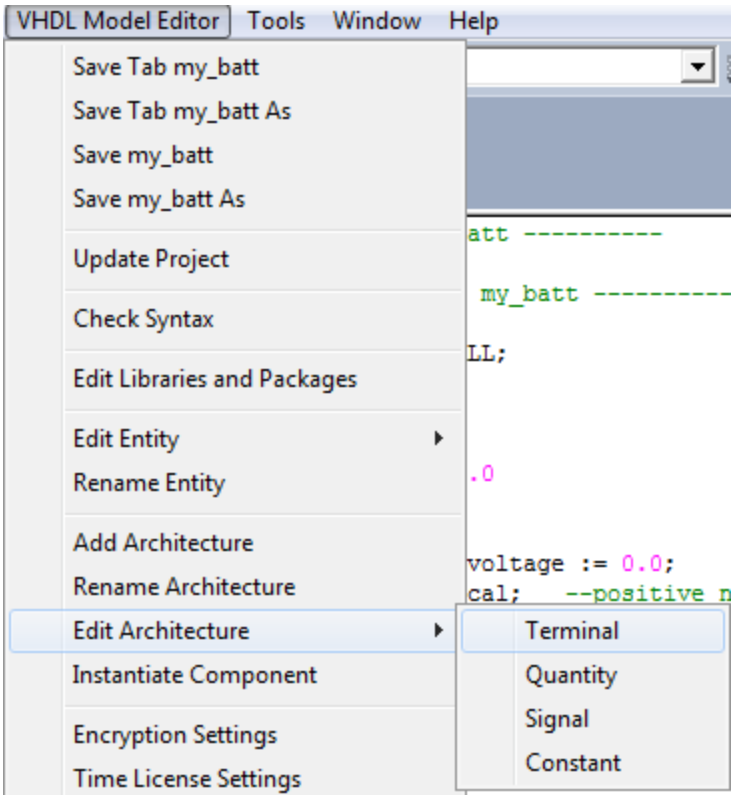


Internal nodes (terminals) **t1** and **t2** will be created and equations derived based on circuit analysis for the voltages across each component. The final code should look similar to the following:

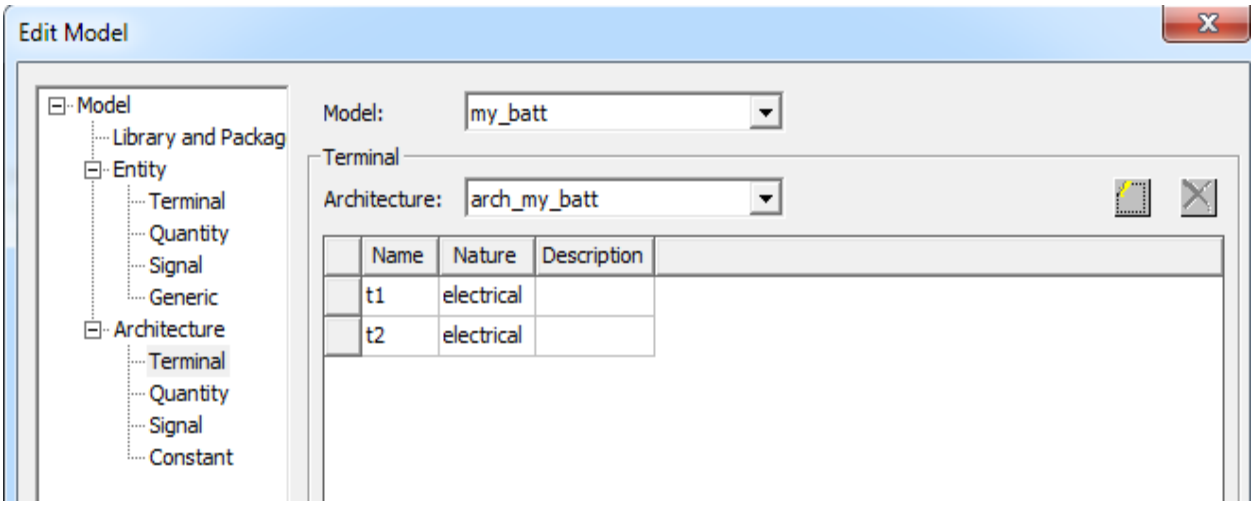
```
---------- ARCHITECTURE DECLARATION arch_my_batt --------
ARCHITECTURE arch_my_batt OF my_batt IS

    TERMINAL t1 :  electrical;
    TERMINAL t2 :  electrical;

    QUANTITY v_ri across p  TO t1;
    QUANTITY v_fc across t1 TO m;
    QUANTITY v_rd across t1 TO t2;
    QUANTITY v_sc across t2 TO m;
    QUANTITY v    across p  TO m;

    QUANTITY i_ri through p  TO t1;
    QUANTITY i_fc through t1 TO m;
    QUANTITY i_rd through t1 TO t2;
    QUANTITY i_sc through t2 TO m;

  CONSTANT ri: RESISTANCE := 1.0e-2;
  CONSTANT fc: CAPACITANCE := 60.0;
  CONSTANT rd: RESISTANCE := 4.0e-2;
  CONSTANT sc: CAPACITANCE := 2.0e4;

BEGIN
  BREAK v_fc => v_init, v_sc => v_init;

  v_ri == i_ri * ri;
  v_fc'dot == 1.0/(fc*factor) * i_fc;
  v_rd == i_rd * rd;
  v_sc'dot == 1.0/(sc*factor) * i_sc;
  v_out == v;

END ARCHITECTURE arch_my_batt;
```

a. Add the two internal terminals.

12. Select **VHDL Model Editor** > **Edit Architecture** > **Terminal**.



13. Use the insert button to add two items. Name the two terminals **t1** and **t2** and make them both **electrical**.

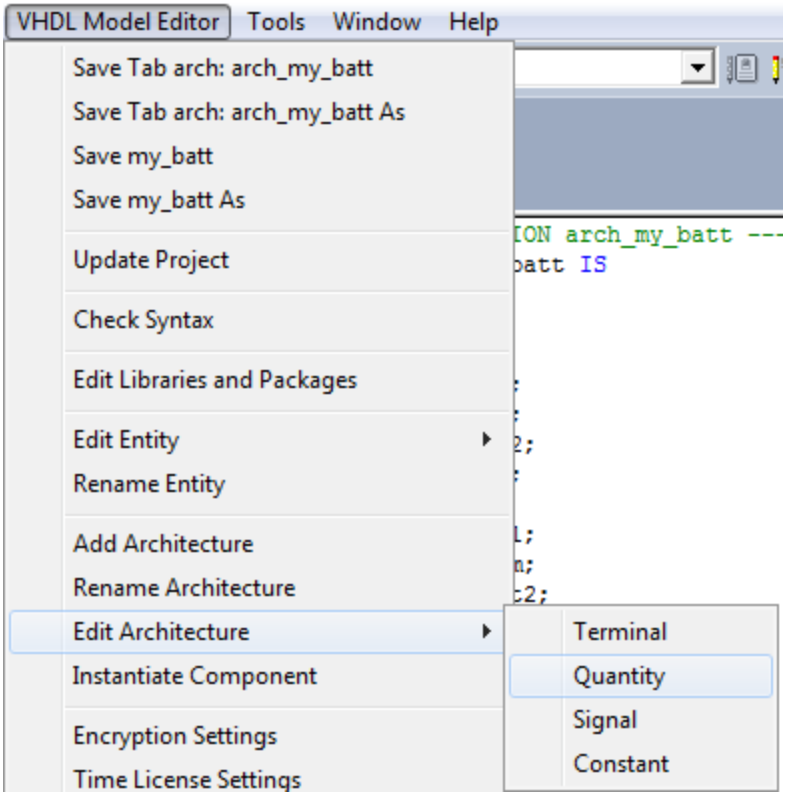


14. Click **OK**.

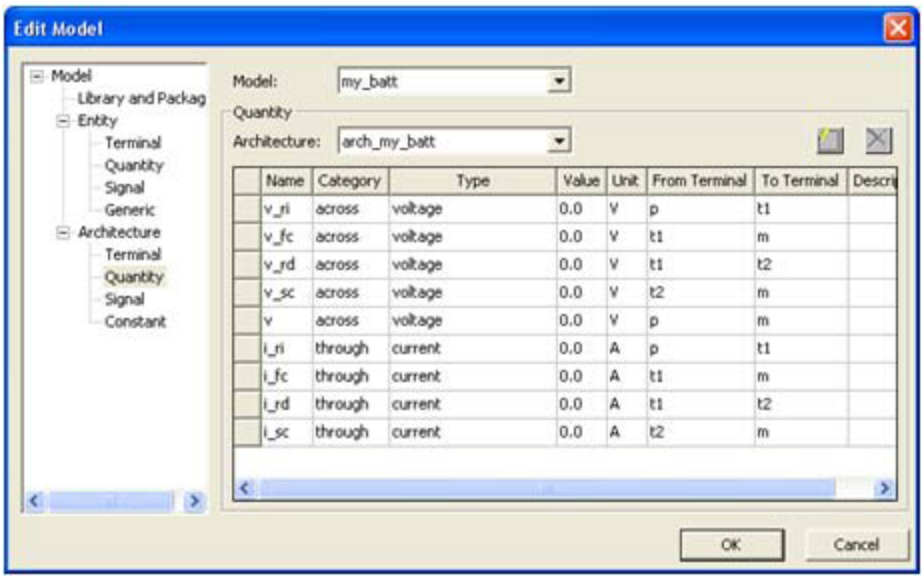
    This automatically adds the code that defines the two internal terminals.

```
---------- ARCHITECTURE DECLARATION arch_my_batt ----------
ARCHITECTURE arch_my_batt OF my_batt IS

    TERMINAL t1 :  electrical;
    TERMINAL t2 :  electrical;
BEGIN

END ARCHITECTURE arch_my_batt;
---------- END VHDLAMS MODEL my_batt ----------
```

15. Insert the quantities that defined the “through” and “across” variables with **VHDL Model Editor** > **Edit Architecture** > **Quantity**.



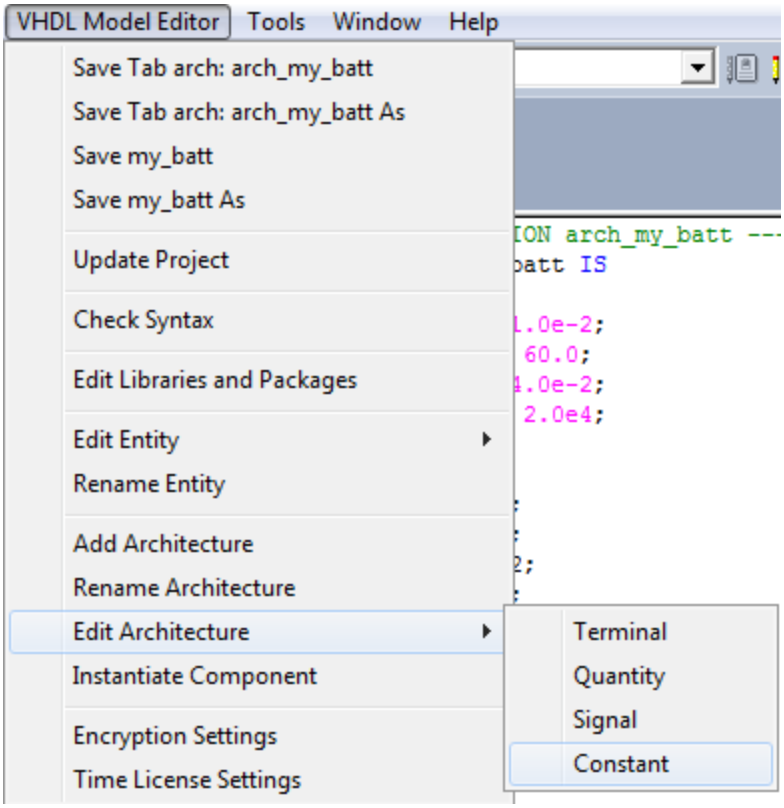
16. Use the insert button to add the following nine items. Add the **Name**, **Type**, **Category**, then **from/to** terminal; then click **OK**.



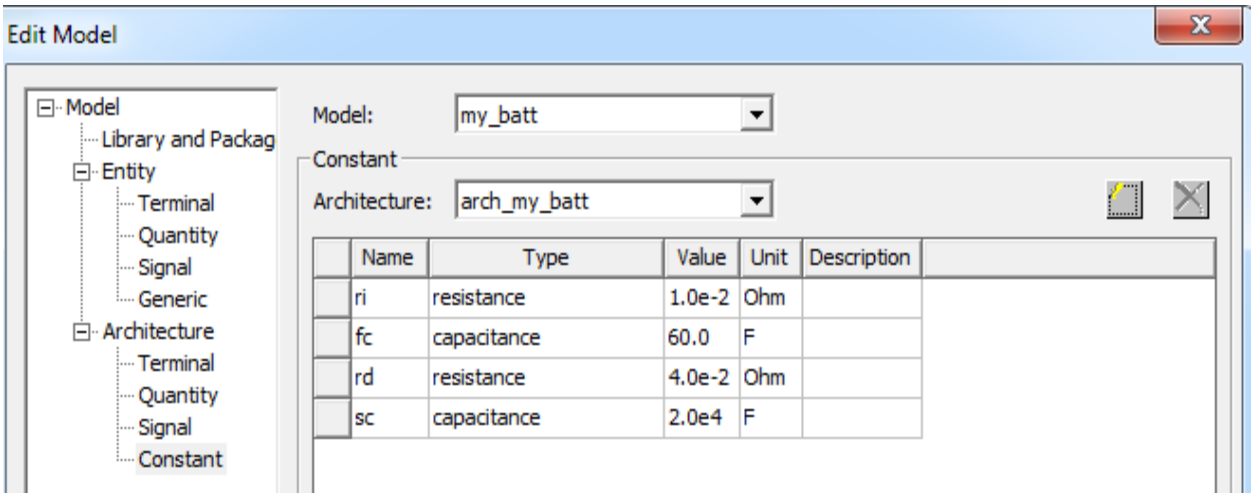
The generated code should appear as follows for the **Quantities**.

```
---------- ARCHITECTURE DECLARATION arch_my_batt ---
ARCHITECTURE arch_my_batt OF my_batt IS

    TERMINAL t1 :  electrical;
    TERMINAL t2 :  electrical;
    QUANTITY v_ri across p TO t1;
    QUANTITY v_fc across t1 TO m;
    QUANTITY v_rd across t1 TO t2;
    QUANTITY v_sc across t2 TO m;
    QUANTITY v across p TO m;
    QUANTITY i_ri through p TO t1;
    QUANTITY i_fc through t1 TO m;
    QUANTITY i_rd through t1 TO t2;
    QUANTITY i_sc through t2 TO m;
BEGIN

END ARCHITECTURE arch_my_batt;
---------- END VHDLAMS MODEL my_batt ----------
```

17. Add the constants using the menu **VHDL Model Editor** > **Edit Architecture** > **Constant**.



18. Using the **Insert** button, add four items and define the constants as shown below:



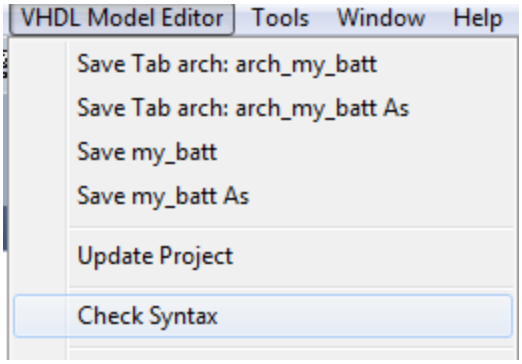
The generated code should appear as shown below. If code doesn't appear, you may need to add the text directly.

```
  CONSTANT ri: RESISTANCE  := 1.0e-2;
  CONSTANT fc: CAPACITANCE := 60.0;
  CONSTANT rd: RESISTANCE  := 4.0e-2;
  CONSTANT sc: CAPACITANCE := 2.0e4;

BEGIN
```

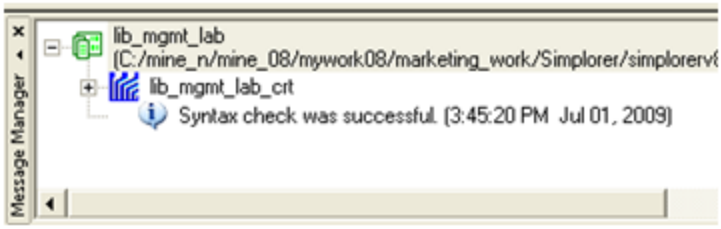
19. Type the equations into the body of the architecture, and add the **Break** statement for initialization, arranging the text as desired for clarity. The final **Architecture** code should appear as shown below:

```
---------- ARCHITECTURE DECLARATION arch_my_batt -----
ARCHITECTURE arch_my_batt OF my_batt IS

    TERMINAL t1 :  electrical;
    TERMINAL t2 :  electrical;

    QUANTITY v_ri across p  TO t1;
    QUANTITY v_fc across t1 TO m;
    QUANTITY v_rd across t1 TO t2;
    QUANTITY v_sc across t2 TO m;
    QUANTITY v    across p  TO m;

    QUANTITY i_ri through p  TO t1;
    QUANTITY i_fc through t1 TO m;
    QUANTITY i_rd through t1 TO t2;
    QUANTITY i_sc through t2 TO m;

  CONSTANT ri: RESISTANCE  := 1.0e-2;
  CONSTANT fc: CAPACITANCE := 60.0;
  CONSTANT rd: RESISTANCE  := 4.0e-2;
  CONSTANT sc: CAPACITANCE := 2.0e4;

BEGIN
  BREAK v_fc => v_init, v_sc => v_init;

  v_ri == i_ri * ri;
  v_fc'dot == 1.0/(fc*factor) * i_fc;
  v_rd == i_rd * rd;
  v_sc'dot == 1.0/(sc*factor) * i_sc;
  v_out == v;

END ARCHITECTURE arch_my_batt;
```

20. Select **VHDL Model Editor** > **Check Syntax** to check the syntax.

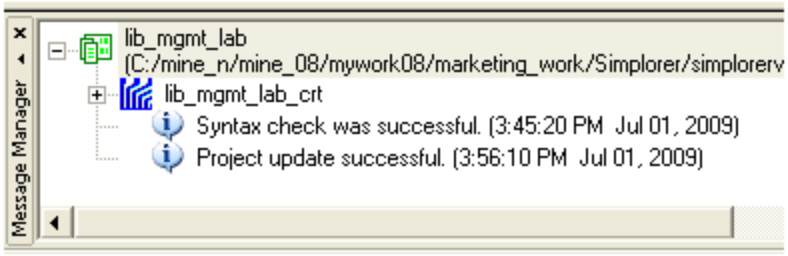


The status of the syntax check will be displayed in the **Message Manager** pane.

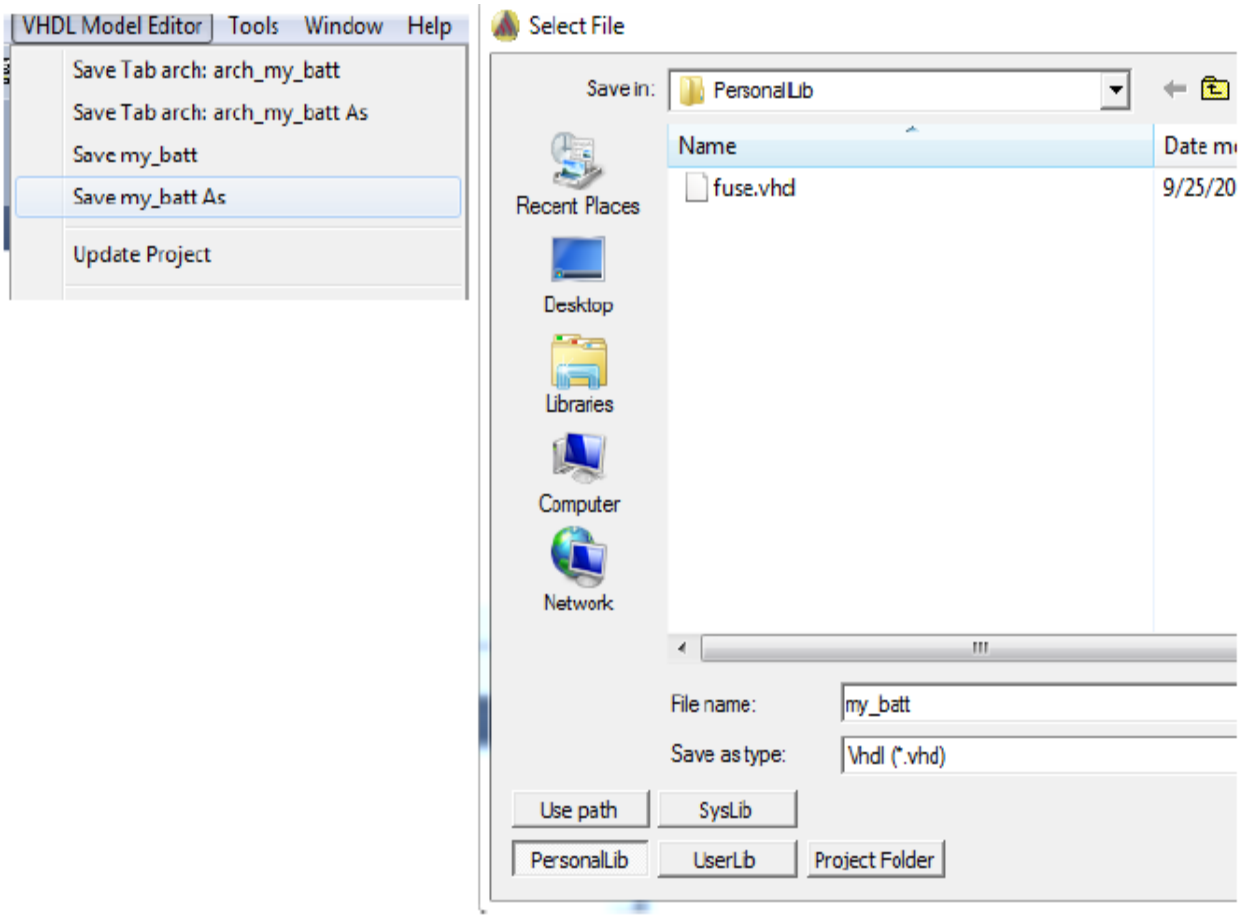


21. Check to make sure the VHDL-AMS code will compile correctly by selecting **VHDL Model Editor** > **Update Project**. Click **OK** for the pop-up window.

    The status of the compile is displayed in the **Message Manager** pane. If the message "Project update successful" appears, then it compiled with no errors.



22. Save the VHDL-AMS model to a text file in your personal library folder (or anywhere you desire) using the **VHDL Model Editor** > **Save** *my_batt***As**. Keep the name *my_batt*. A VHDL-AMS model file named **my_batt.vhd** has been created.

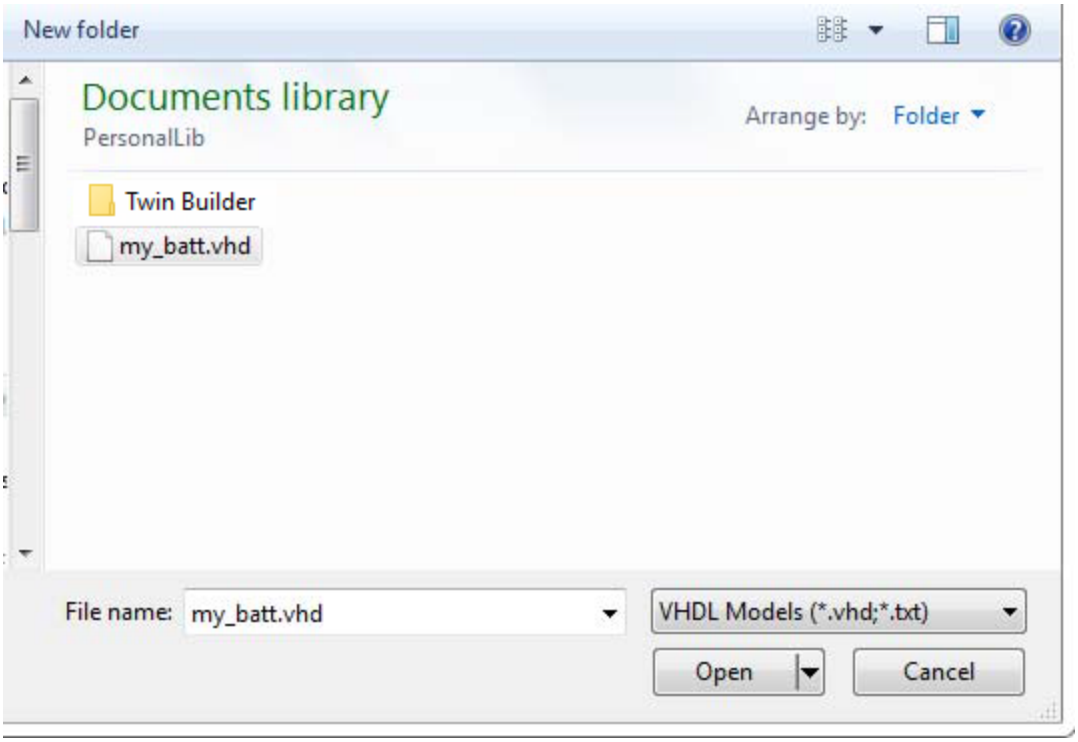


23. Close the VHDL Editor window.
24. Clean up the project definitions on the Project Manager Project tab by selecting **Tools** > **Project Tools** > **Remove Unused Definitions** on the menu bar, then choosing **Select All** on the popup window and clicking **Apply**. Repeat until no unused definitions remain in the list.

Import VHDL-AMS model into Twin Builder

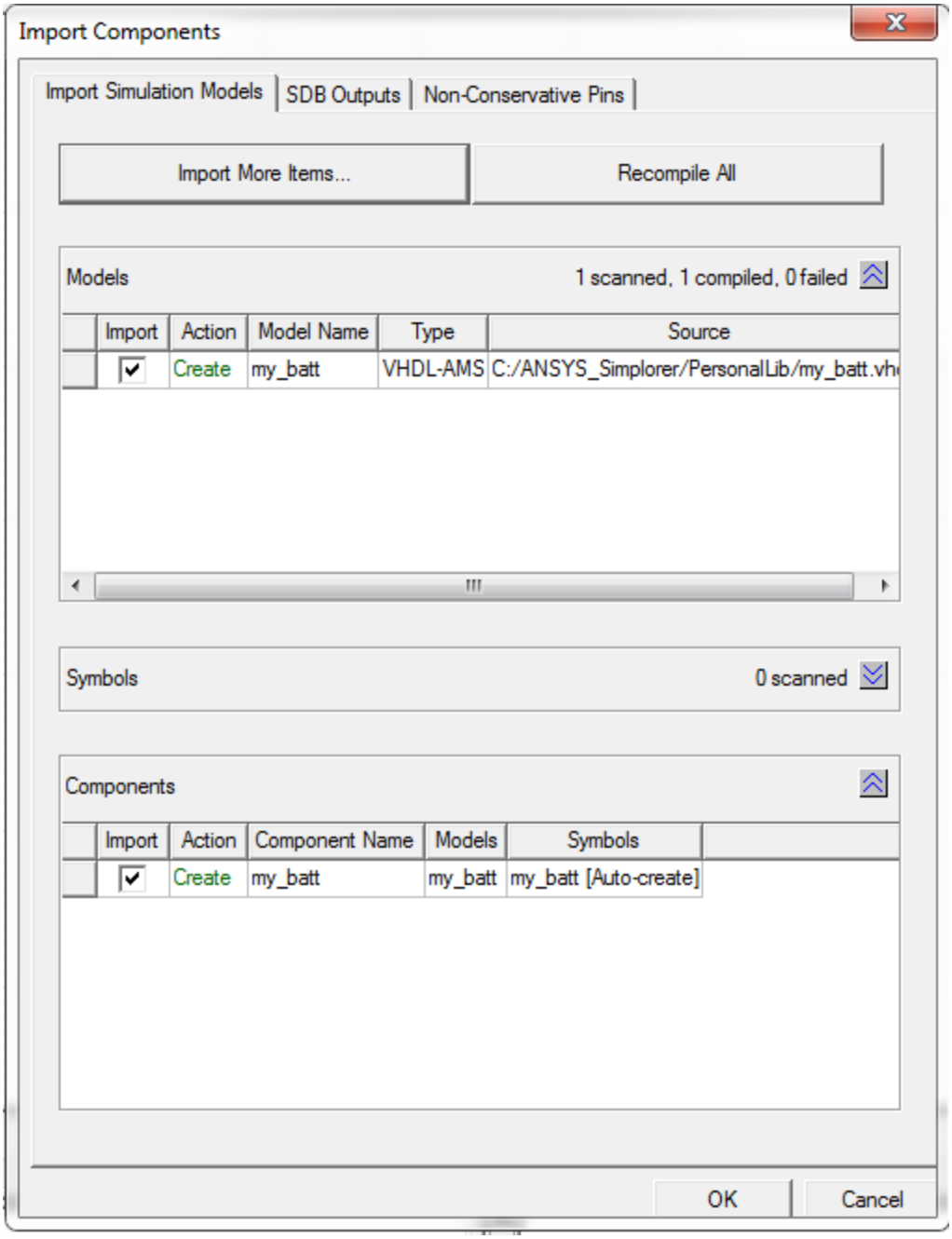
To import the VHDL-AMS battery model (**my_batt.vhd**) that you created in the previous section:

1. On the Twin Builder main menu, select **Tools** > **Project Tools** > **Import Simulation Models**.
2. Select **VHDL Models (*.vhd, *.txt)** as the **Files of type**, navigate to where **my_batt.vhd** is located, and open it.

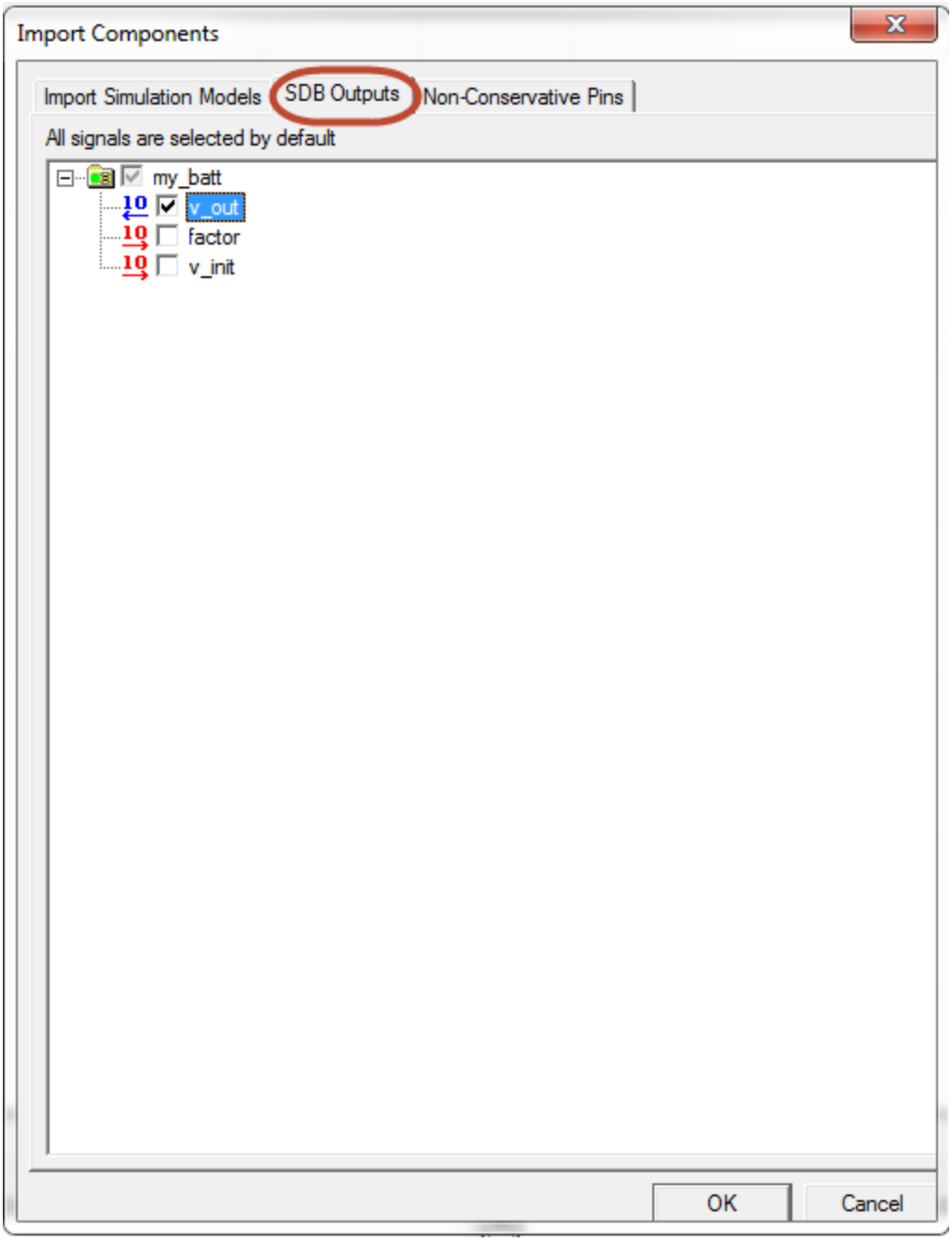


3. Select the following settings for each tab in the **Import Components** dialog box, and then click **OK**.

   This shows the **Import Simulation Models** tab.



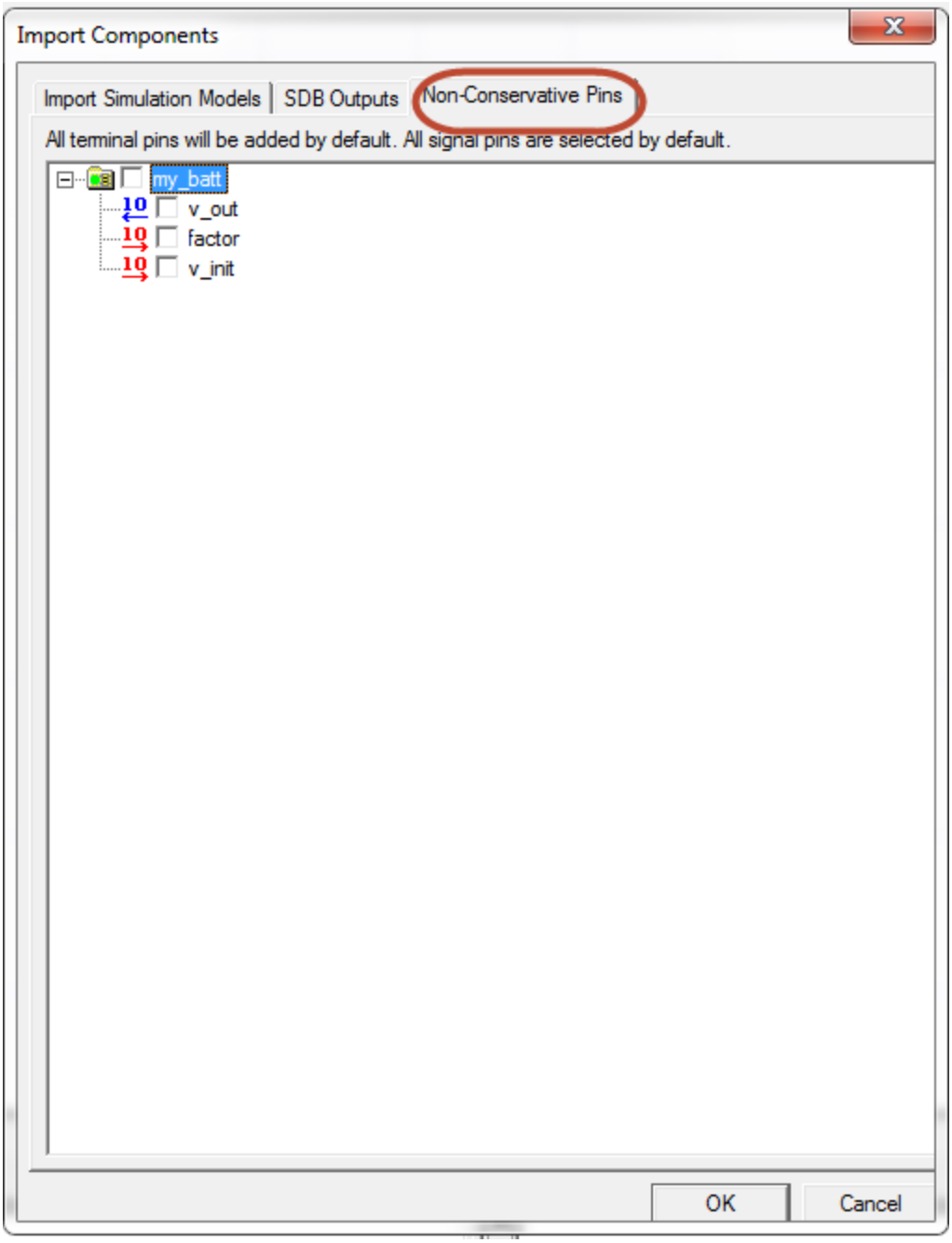
This shows the **SDB Outputs** tab.



Note:

The SDB tab defines what is available in the data base file (for example, **v_out** is the battery's terminal voltage and this will make it available for plotting).

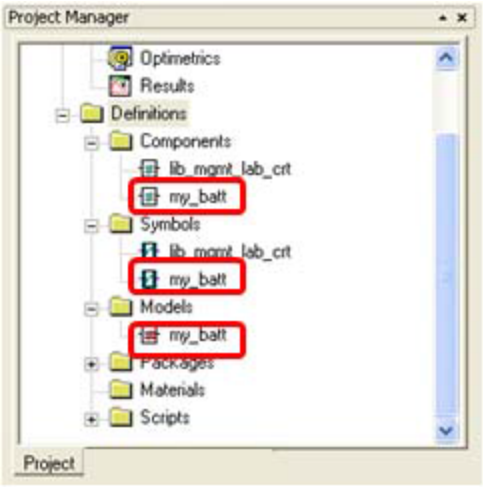
This shows the **Non-Conservative Pins** tab.



Note:

The **Non-Conservative Pins** tab allows you to define pins on the model to represent parameter inputs or outputs if desired.

In the **Project Manager** pane under the Definitions folder, observe that a Component, Symbol, and Model definition have now been created for **my_batt**.



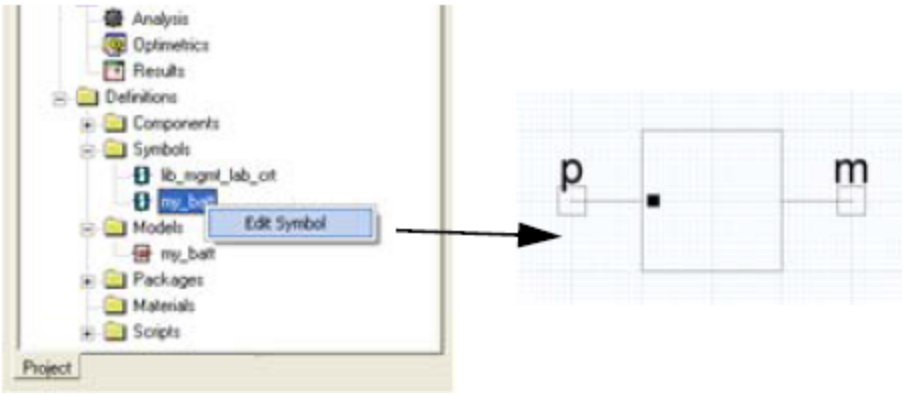
4. If you want to edit/view the VHDL-AMS model, select **my_batt** in the Models folder, and then right-click and select **Edit Model**, to open the VHDL-AMS model code in the **VHDL-AMS Editor**.

5. To exit the **VHDL-AMS Editor**, select the lower “x” in the upper right corner of the editor window.

Create a user-defined symbol for *my_batt*

When the VHDL-AMS model is imported, Twin Builder automatically creates a generic symbol for it.

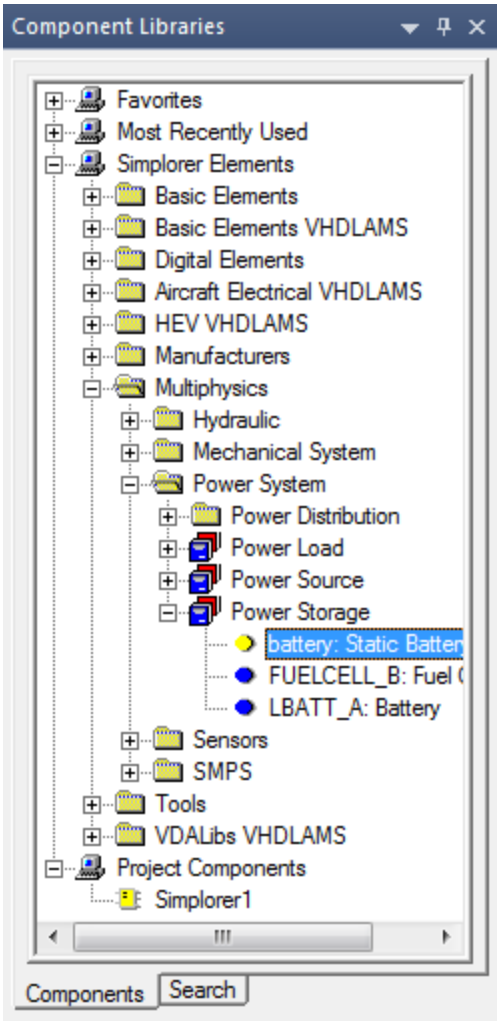
1. To edit the symbol, select **my_batt** in the **Definitions/Symbols** folder in the **Project Manager** pane **Project** tab and right-click and select **Edit Symbol**. This opens the **Symbol Editor** window.



> **Note:**
>
> When creating a new symbol, you can draw it from scratch, or copy existing symbols to use.

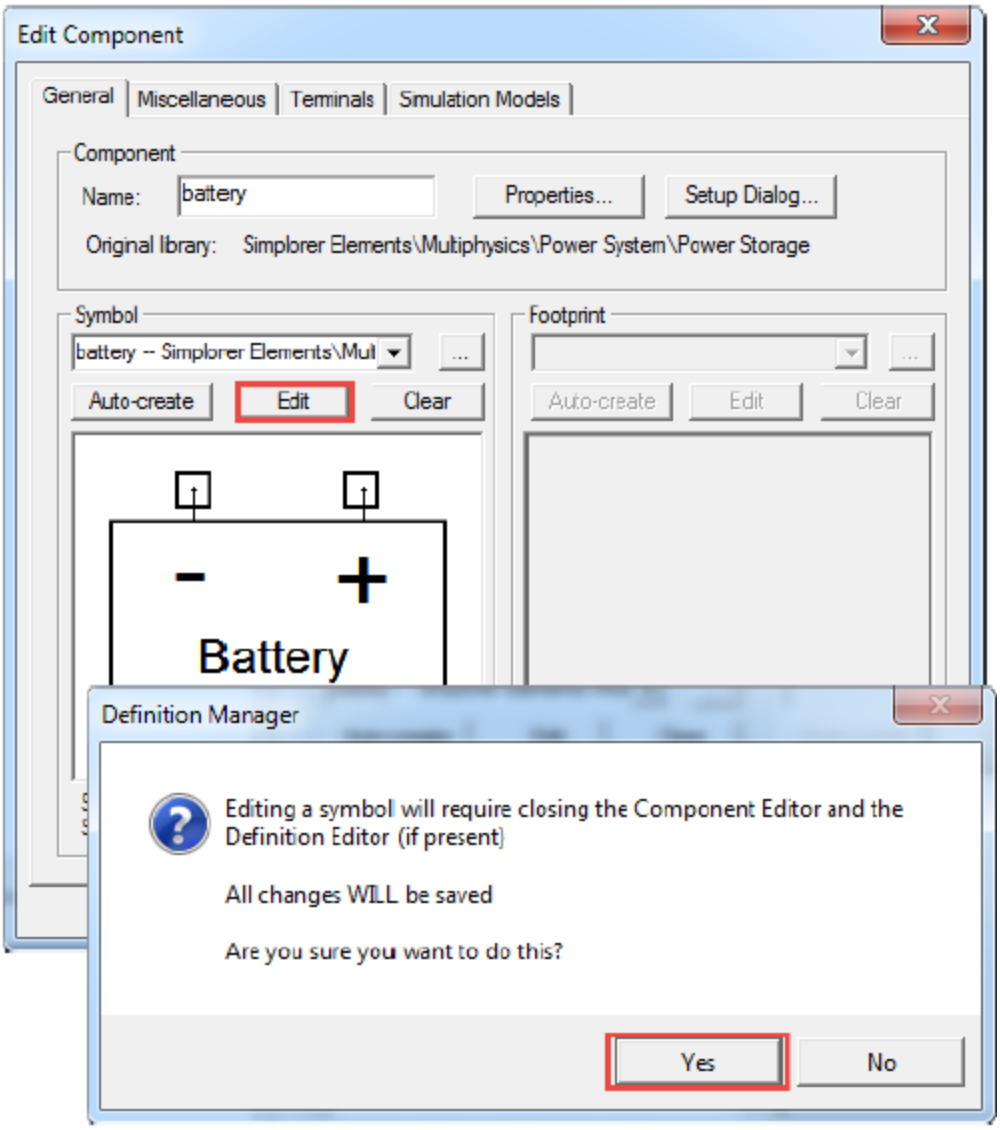
This example uses an existing battery symbol from which you copy the graphics to use for the **my_batt** symbol.

2. While still in the symbol editor, go to the **Component Libraries** window. Under **Simplorer Elements/Multiphysics/Power System/Power Storage**, select the **battery: Static Battery Model**, then right-click and select **Edit Component**.

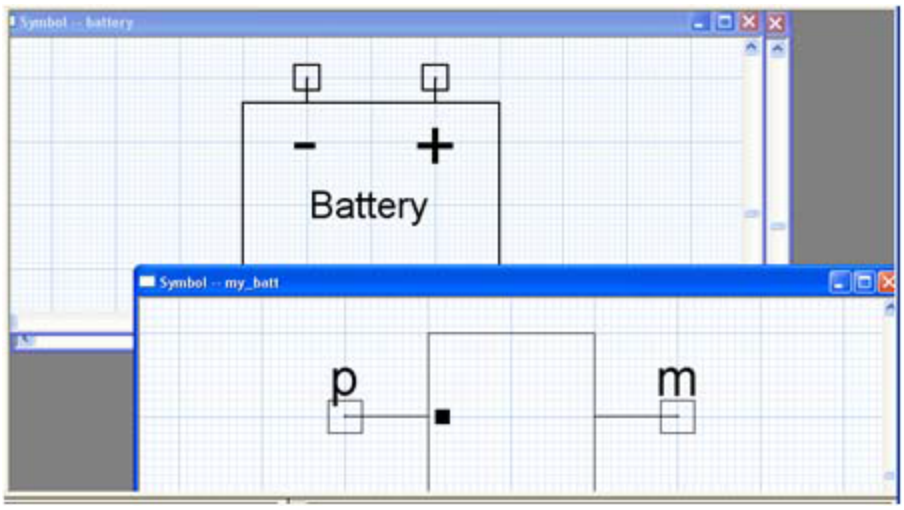


The **Edit Component** dialog box opens.

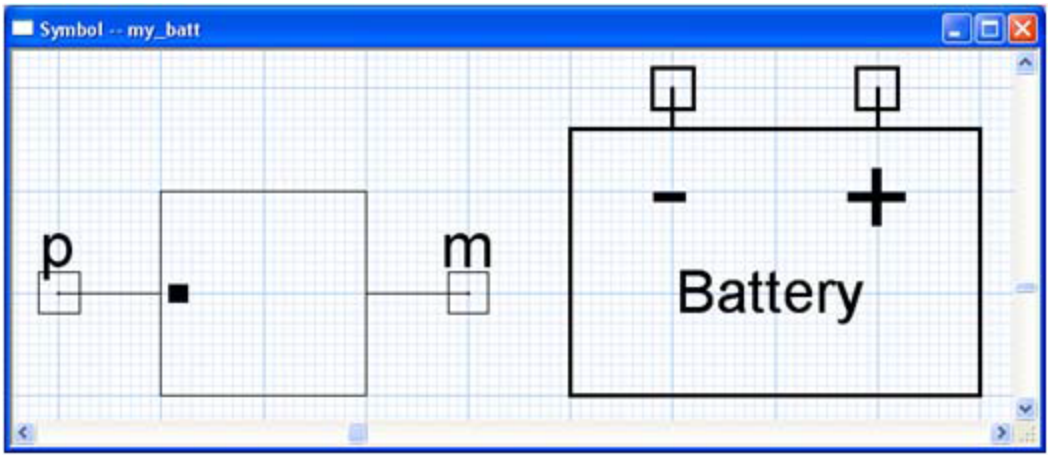
3. Click the **Edit** button in the symbol area and select **Yes** to close the Component Editor.



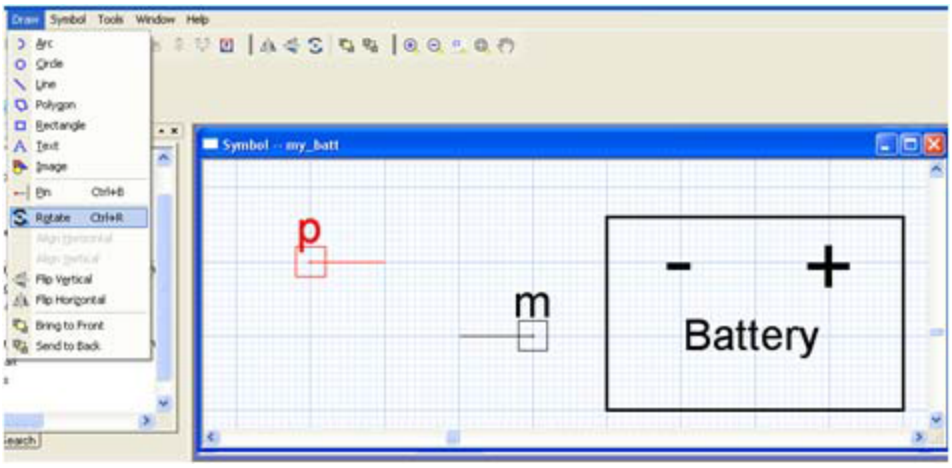
4. After the battery symbol has also been opened for editing, there are now two symbol windows available. Click on the cascade selection so that both symbol windows are shown at the same time, or use **Window** > **Cascade** (close all other windows).



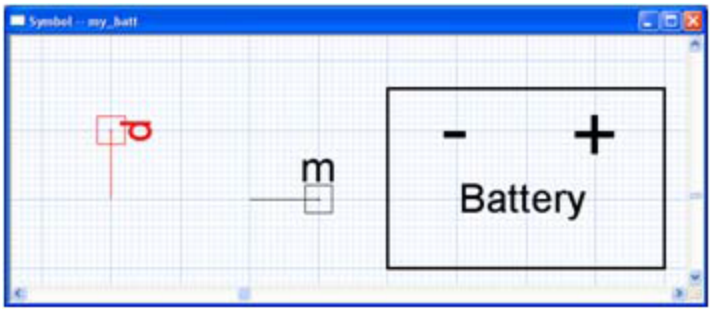
5. Select the graphics from the **battery** symbol window, and paste them into the **my_batt** symbol window (using standard windows functions: Ctrl-C for copy, Ctrl-V for paste). You should now have the following in the **my_batt** symbol edit window. (Close the other battery symbol edit window.)



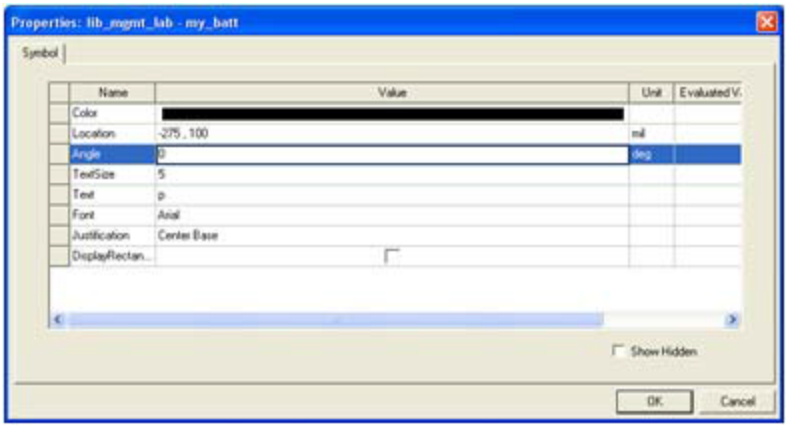
6. Using the symbol editor, replace the pin graphics on the battery symbol that was copied with the actual pins from the **my_batt** default symbol. To do this, select unwanted graphics (such as the battery symbol pins) and press Del.



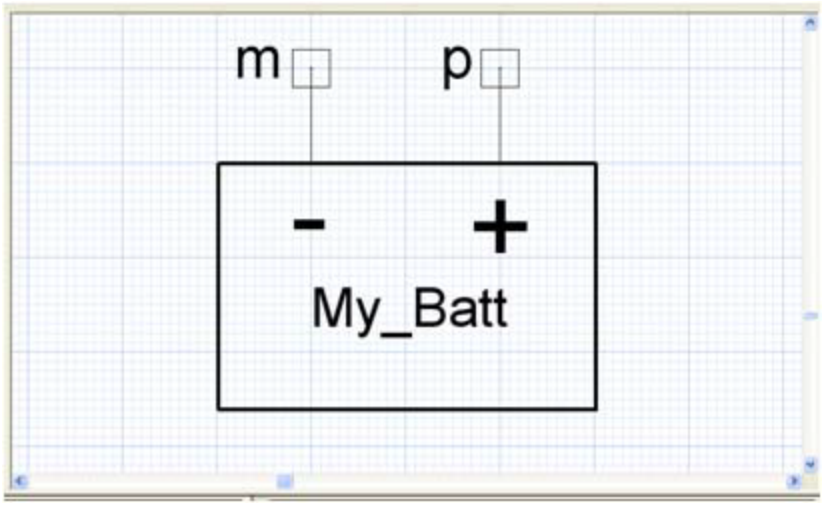
7. Use the various **Draw** menu commands such as **Rotate** and **Flip...** to orient the “**p**” and “**m**” pins.

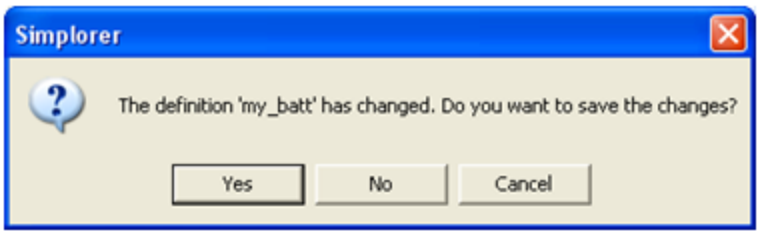


To rotate a pin name such as “**p**”, double-click on the name, and change the **Angle** to “**0**”.



8. Use the **Draw** menu commands until the symbol for **my_batt** appears as shown below. Save it with **File** > **Save**, and click **Yes** to save the new symbol definition for the **my_batt** component.





9. When finished, close the symbol edit window by clicking the “x” in the upper right corner.

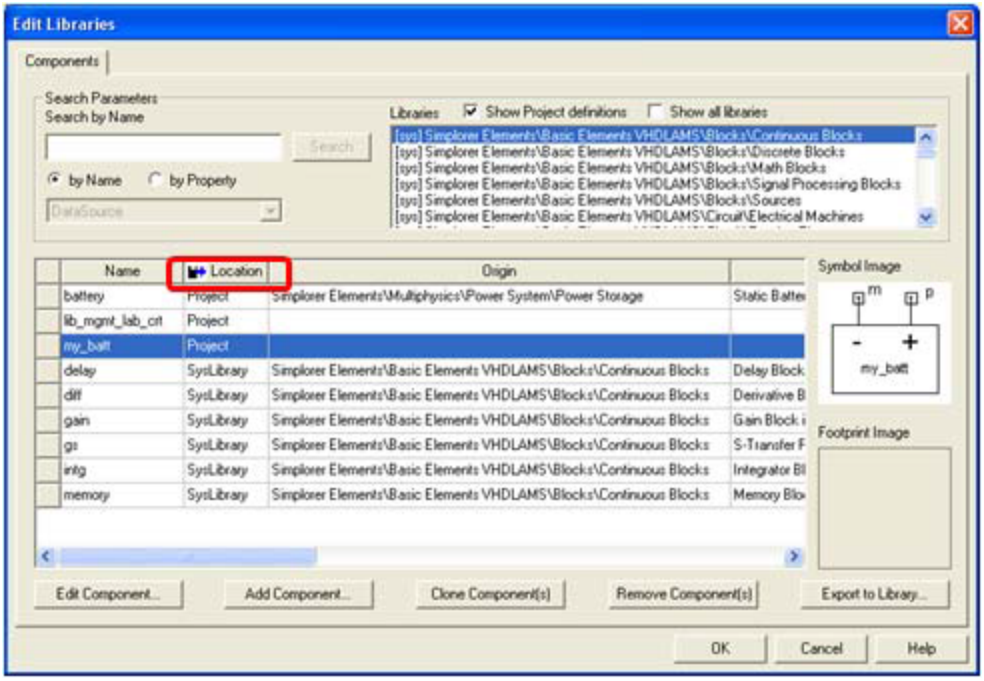
Export a VHDL-AMS Component to a Personal Library

Now that the VHDL_AMS model definition has been imported, and a symbol and component definition defined for it, you can export it into a **PersonalLib** library.

> **Note:**
>
> Use the **Tools** > **Edit Libraries** submenus to display the contents of selected library files, and to copy, modify, and export components to user/personal libraries.

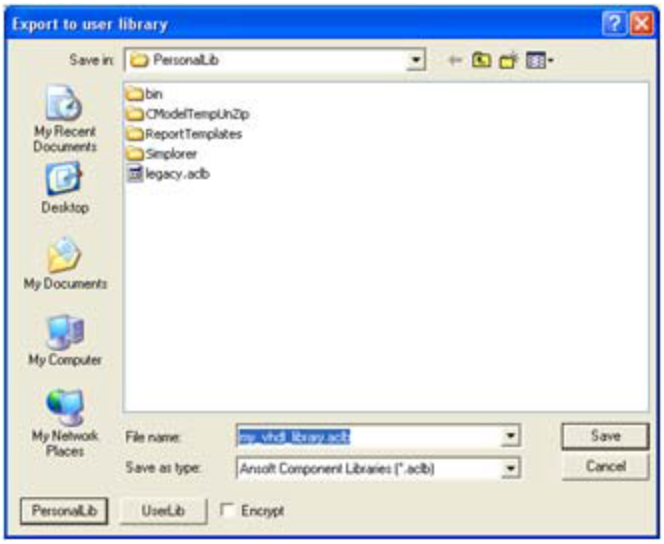
1. On the Project tab in the Project Manager window, double-click the **lib_mgmt_lab_crt** design to open it.
2. Select **Tools** > **Edit Libraries** > **Components**.

This shows the Component Definitions for the selected library at the top and for the Project definitions as seen below. You can sort the order by selecting the “location” icon title. You should see the **my_batt** component listed under the project location.



3. To export the component to a personal library, select **my_batt** from the project definitions, and then click **Export to library**. The following window appears. It should be pointing to your personal library location. Name the library where **my_batt** will be placed. In this

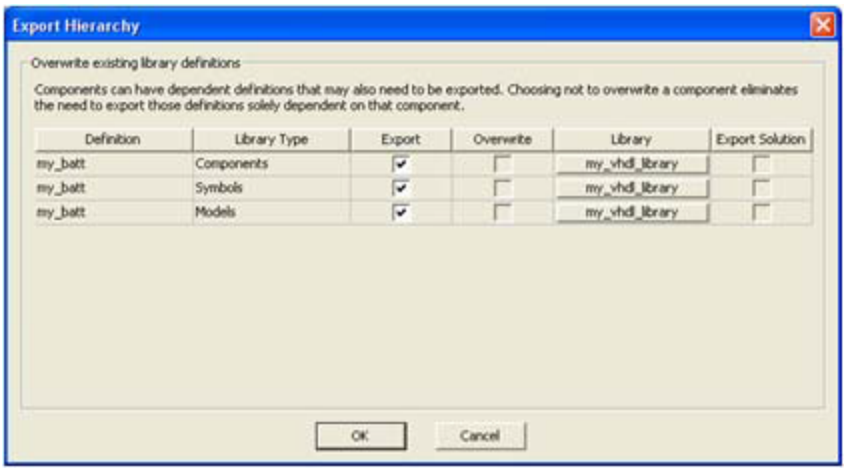
example the library name is **my_vhdl_library.aclb**.



The following window appears:

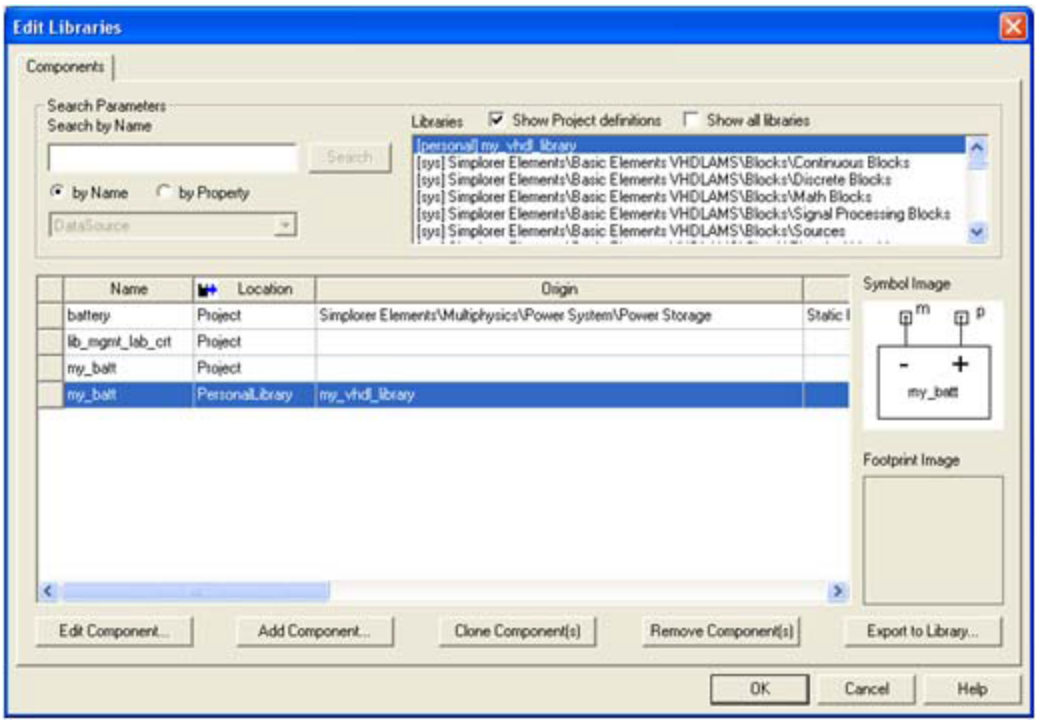
4. Click **Yes**.

   The following window pops up showing that this component has three definitions associated with it (Component, Symbol, and Model).



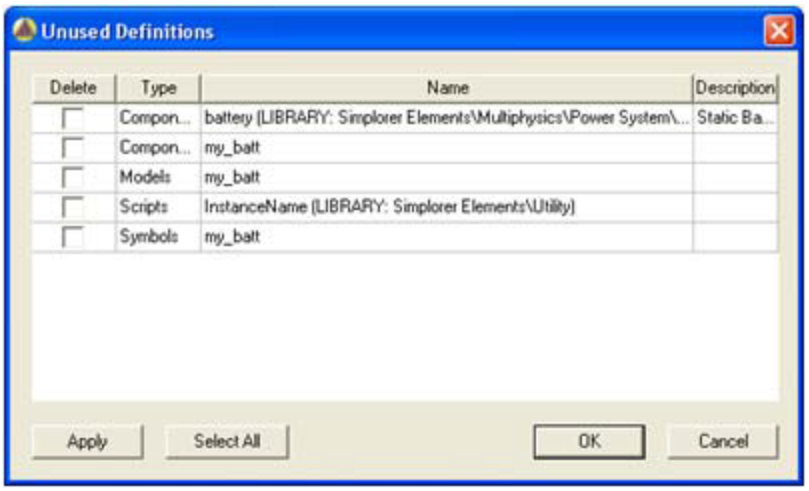
5. Make sure all three are selected and click **OK**.

   The "**my_batt**" component you just exported should show up in the (Personal) **my_vhdl_library** library as shown below. Note that the other Project definitions for **my_batt** still exist at this point.



6. Select **OK** to close.
7. It is a good idea at this point to clean up the Project definitions. To do so, select the **Project** tab in the **Project Manager** pane, then select **Tools** > **Project Tools** > **Remove Unused Definitions** on the main menu bar.

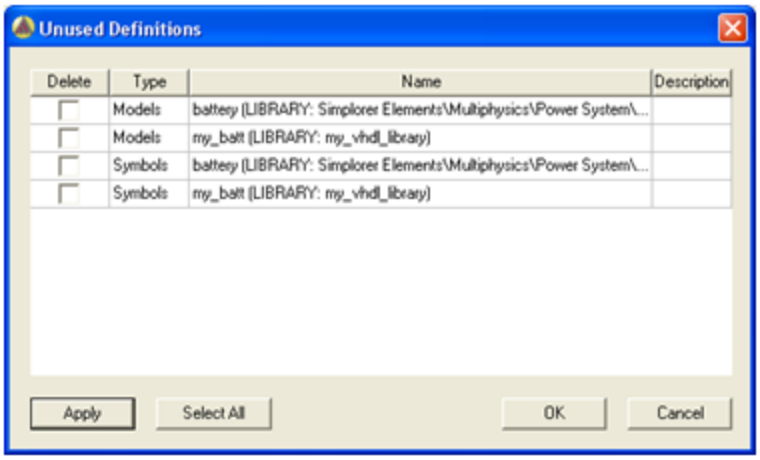
   The window that appears should look similar to the following:



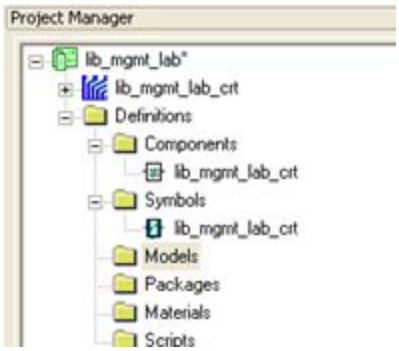
8. Click **Select All**, and then **Apply** to remove the selected definitions. Click **OK** in the following window:

9. Another window with additional unused definitions could pop up. If it does, once again click **Select All** and **Apply** to remove unused definitions. Repeat this process until no

more unused definitions remain.



The project manager should now appear as shown below.



10. Save the project.

Create a Twin Builder Design Using the New Components

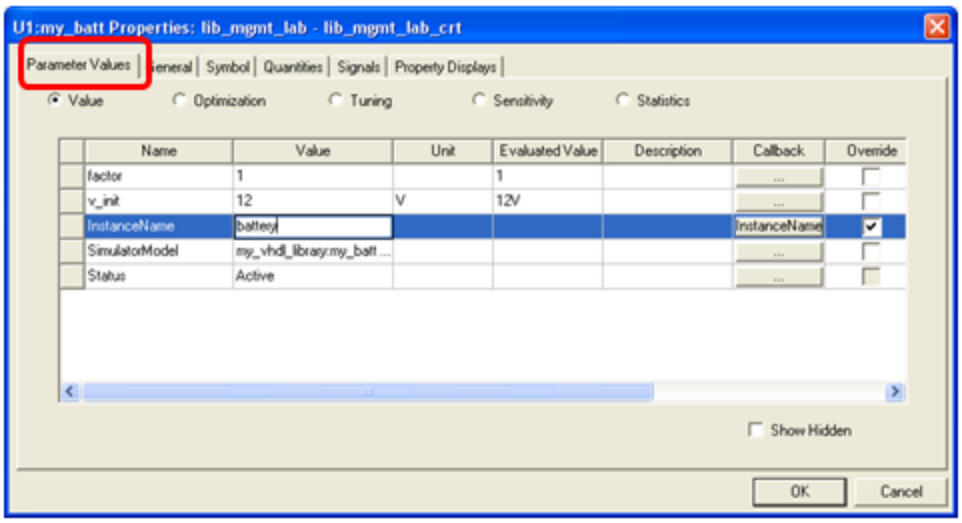
1. In the Component Libraries window, view the contents of the **my_vhdl_library** personal library.

   The VHDL-AMS model created in the previous sections should appear.

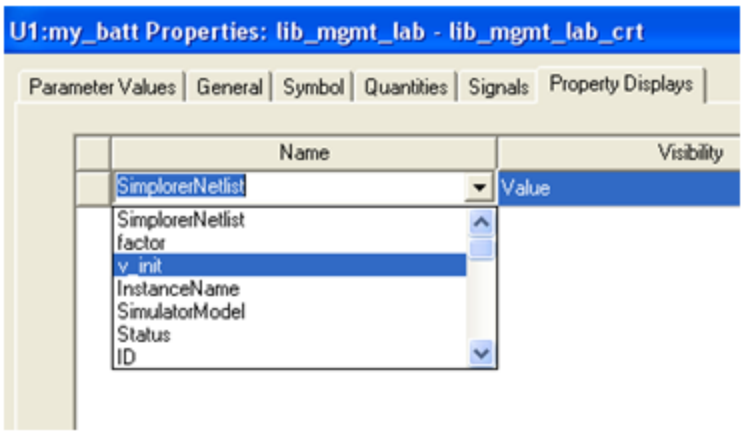
2. Drag the **my_batt** VHDL-AMS model into the **lib_mgmt_lab_crt** Twin Builder design that you created previously.

> **Note:**
>
> In Twin Builder, you can mix VHDL-AMS models with all other available types of models (SML, C-Models, and so on).

3. Place a resistor from the **Simplorer Elements/Basic Elements/Circuit/Passive Elements** library.
4. Select **Draw** > **Ground** to add a ground node. You can also press Ctrl-G to add a ground.
5. Connect the resistor in parallel with the battery model, and connect the ground to the battery's negative terminal. Double-click the **my_batt** symbol and select the **Parameter Values** tab to edit the parameters for the model. Leave the values for **factor** and **v_init** at their default values as defined in the VHDL-AMS code. Rename the **InstanceName** to **battery**.

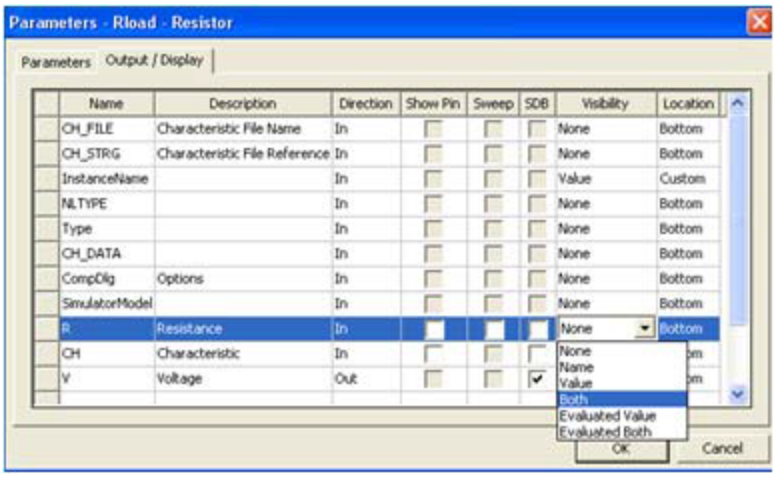


6. Select the **Properties Display** tab, and then click **Add**. Add a display to show the value of **v_init** (which represents the initial charge voltage on the battery model) on the schematic.

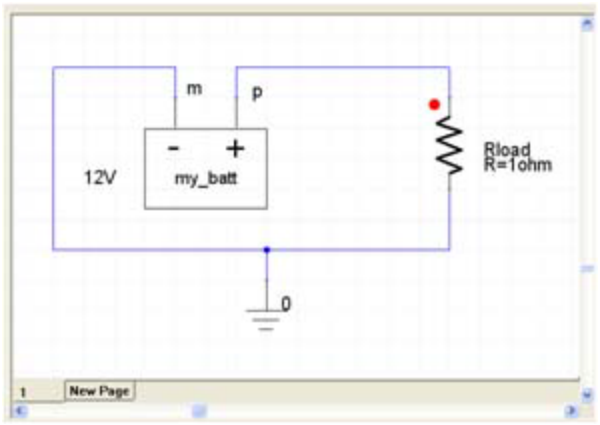


7. Double-click the resistor to open its **Parameters** dialog box and change its name to **Rload** and value to be **1** ohm.

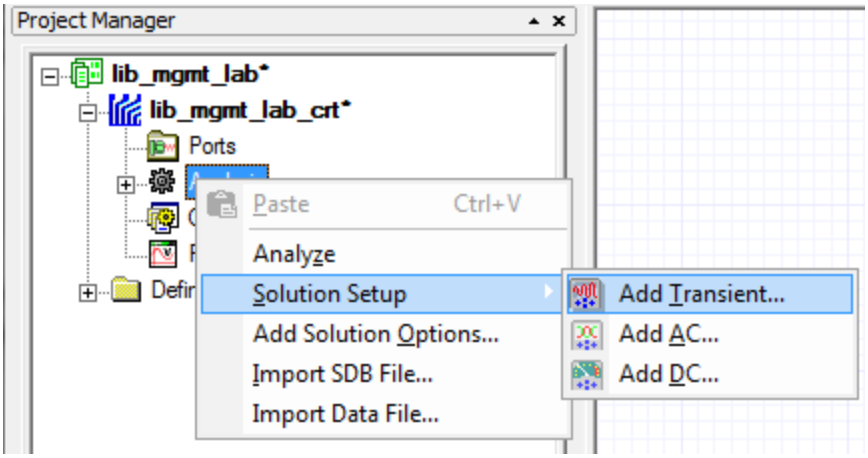
8. Select the **Output / Display** tab and choose the **Visibility** to show **Both** the parameter name and resistance value.



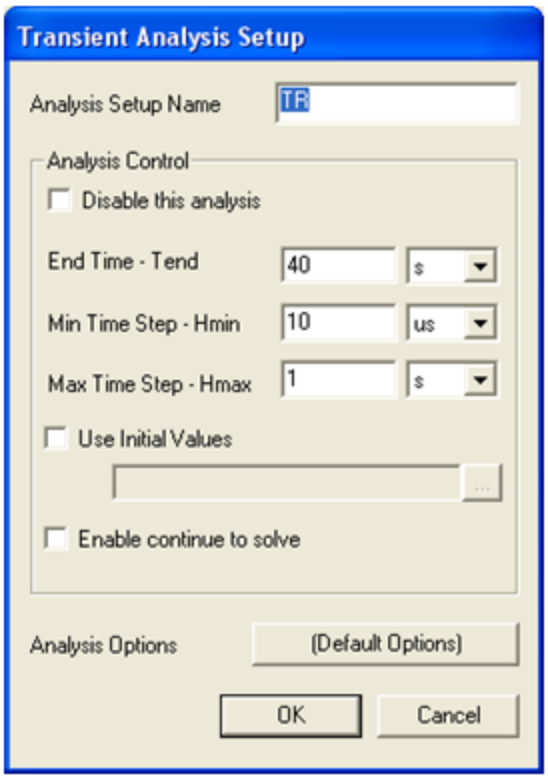
The circuit should appear as shown below. You can improve the text placement capability by choosing not to snap text to grid via the **Schematic** > **Grid Setup** menu item.



9. Add a Transient analysis setup by moving the cursor over the **Analysis** selection in the **Project Manager** pane, and then right-clicking and selecting **Solution Setup** > **Add Transient**.



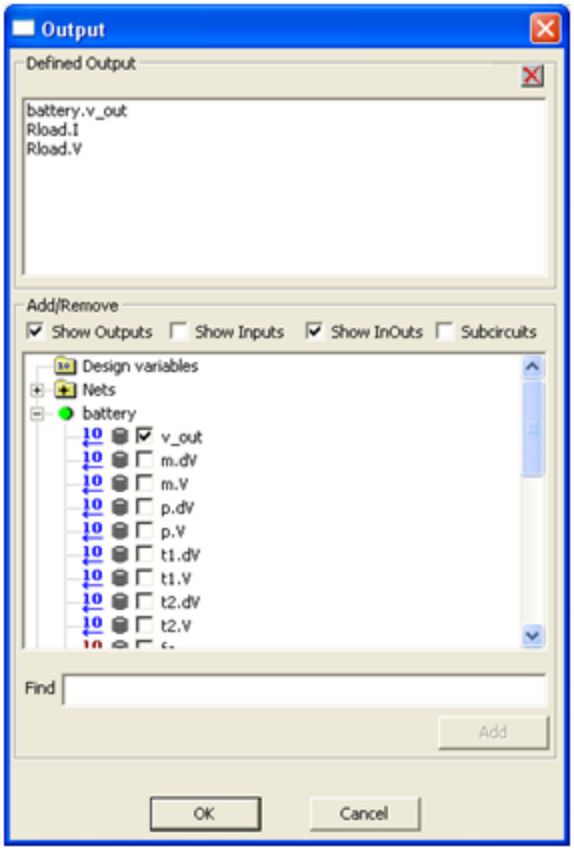
The **Transient Analysis Setup** dialog box appears.



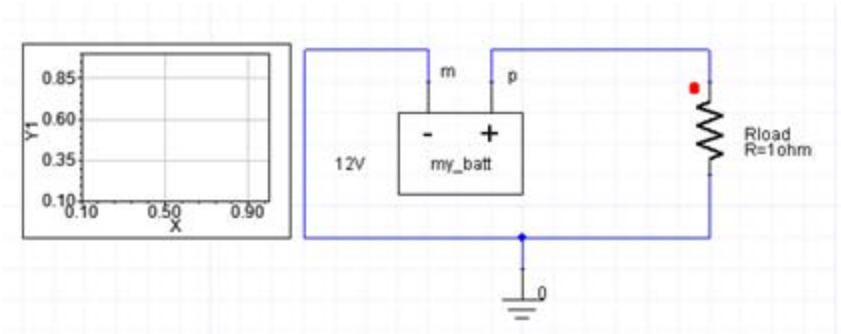
Change the End Time - Tend to be **40s**, and the Max Time Step - Hmax to be **1s**. Leave all other settings as-is and click **OK** to close the dialog box.

10. To view and define the parameters that are available for plotting select **Twin Builder Circuit** > **Output Dialog**.

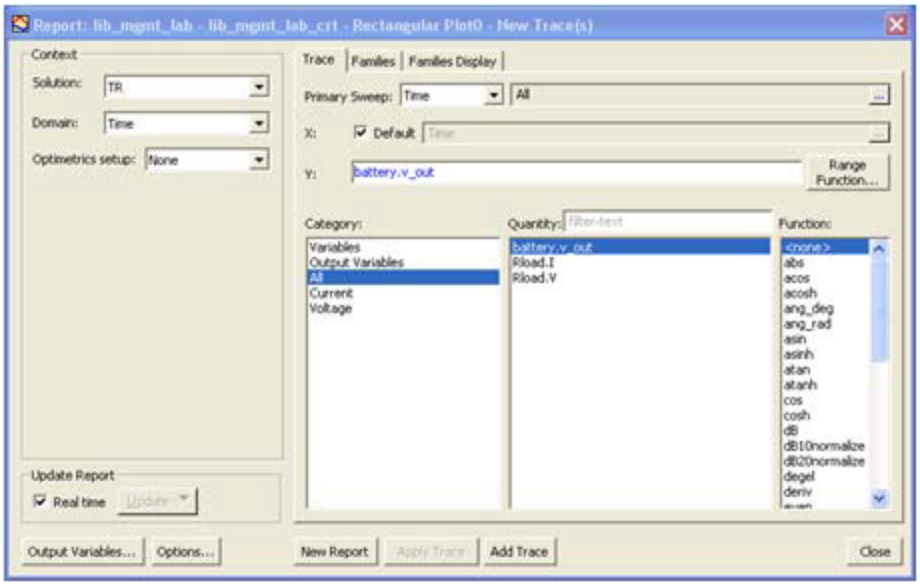
The **Output** dialog box appears.



11. Note that **v_out** had already been declared to be available when the VHDL-AMS model was imported to Twin Builder. Click **OK**.
12. Zoom out (Ctrl+F) on the schematic to make room for a plot. Add a plot to the schematic by selecting **Draw** > **Report** > **Rectangular Plot**.
13. Drag the plot box to the desired size on the schematic.

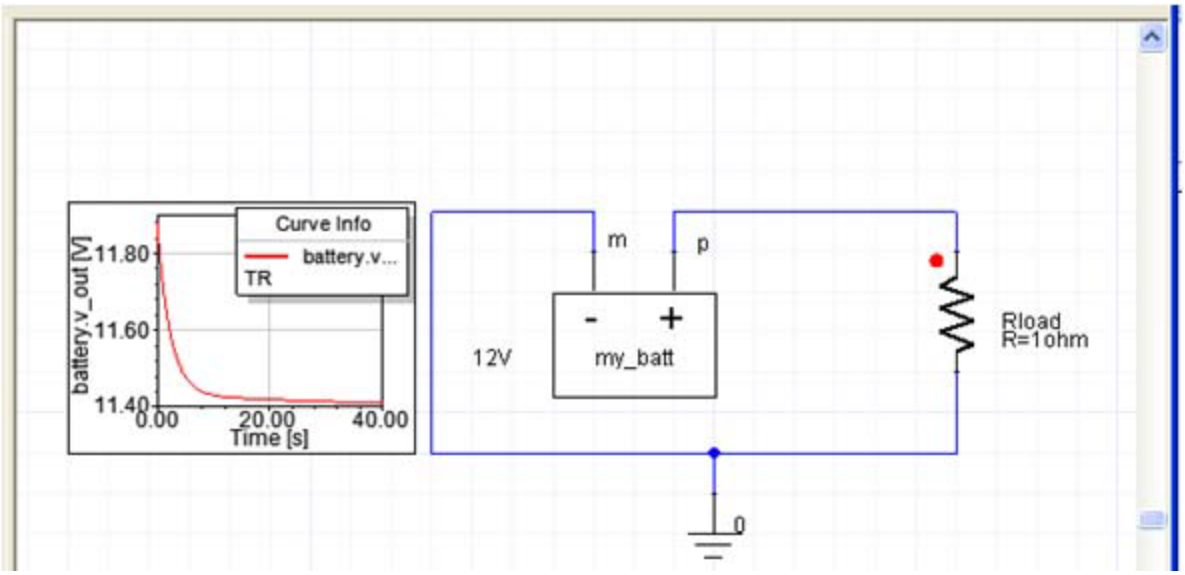


14. Double-click the plot box to open the report dialog box, select the **battery.v_out** Quantity, then click **Add Trace**. Click **Close** to dismiss the dialog box.



15. Analyze the design by moving the mouse over the **TR** analysis in the **Project Manager** pane, and right-click and select **Analyze**.

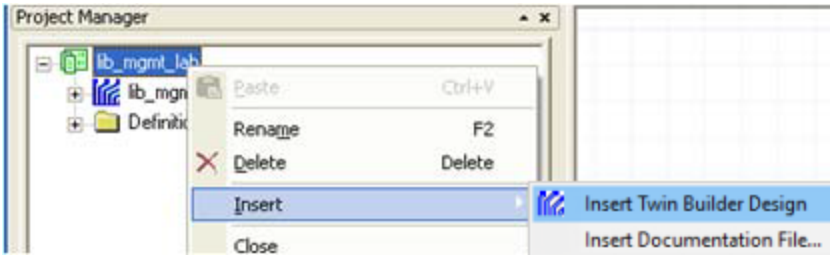
16. The results should now be displayed on the schematic for the battery voltage as it discharges into the load resistance as shown below. **File** > **Save** the project.



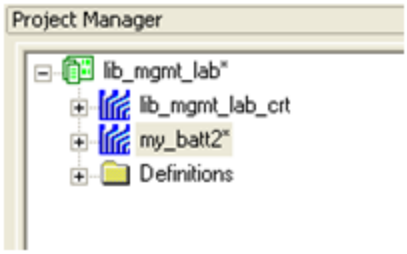
Example #2

This example creates a VHDL-AMS “netlist” model directly from a hierarchical circuit in Twin Builder. It is based on the previous simple battery model.

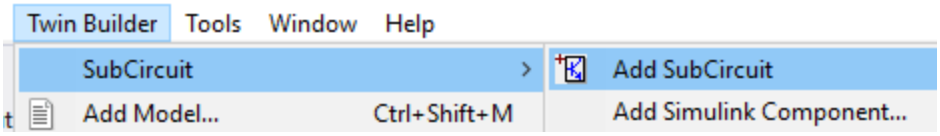
1. Create a new design file in the previous **lib_mgmt_lab** project by selecting the project, right-clicking and selecting **Insert** > **Insert Twin Builder Design**.



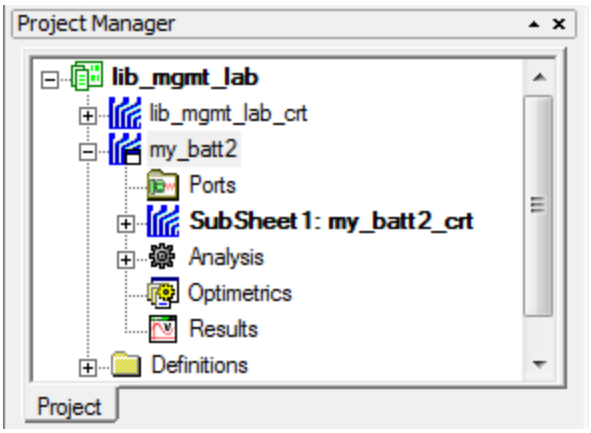
2. Rename the design **my_batt2** and save the project.



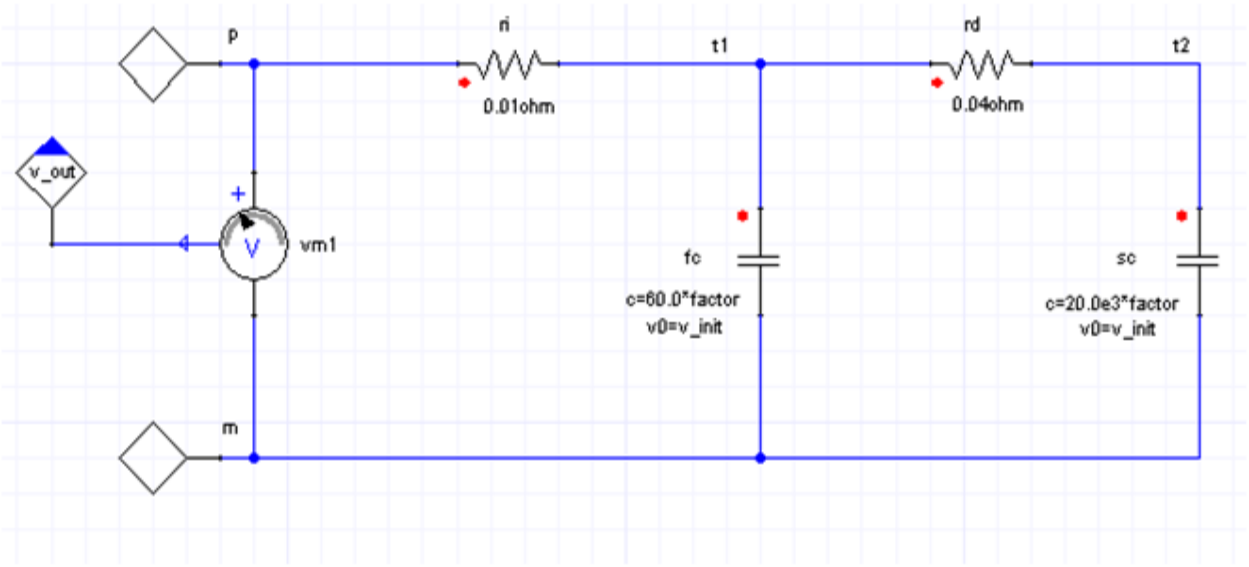
3. Within the **my_batt2** design, add a **SubCircuit** for the battery model.



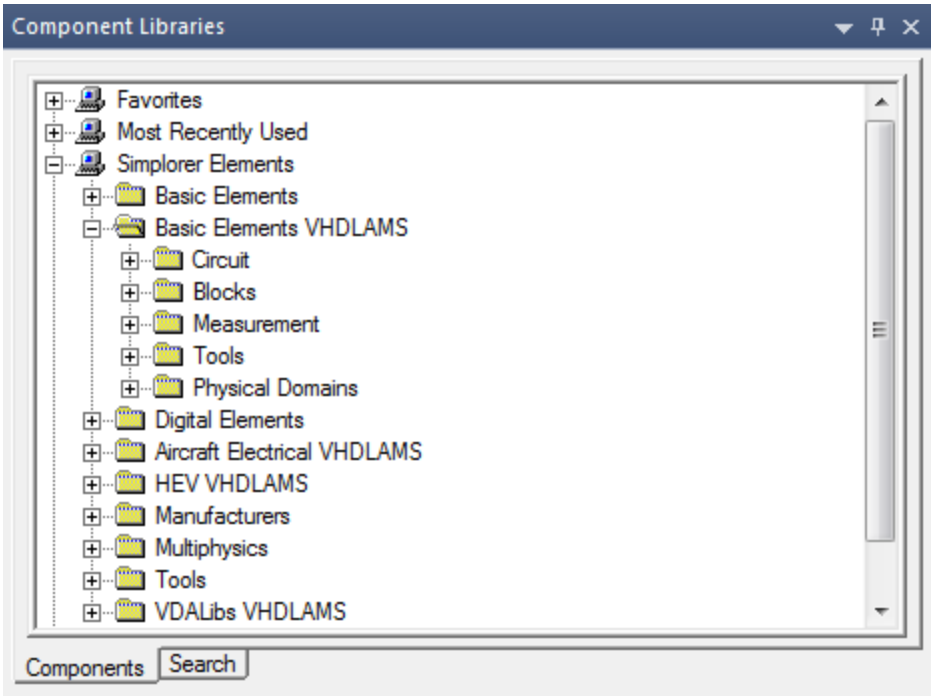
4. This adds a SubCircuit design under the **my_batt2** design. Rename this subcircuit design to **my_batt2_crt**, and save the project.



The following procedure creates a circuit implementation of the battery model in the "**SubSheet1:my_batt2_crt**" subcircuit design using only VHDL-AMS components.

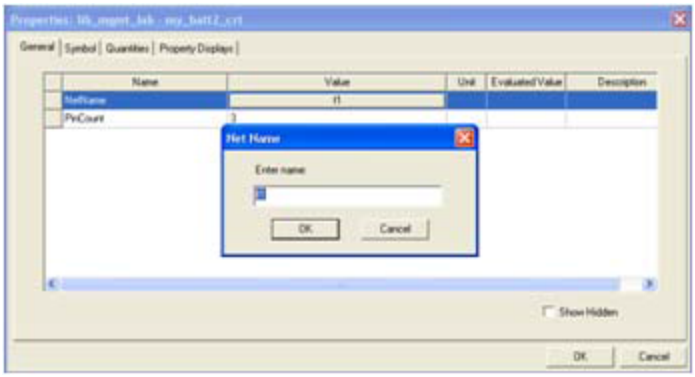


5. In the **Component Libraries** window, expand the **Basic Elements VHDLAMS** folder containing the VHDL-AMS components used in this example.

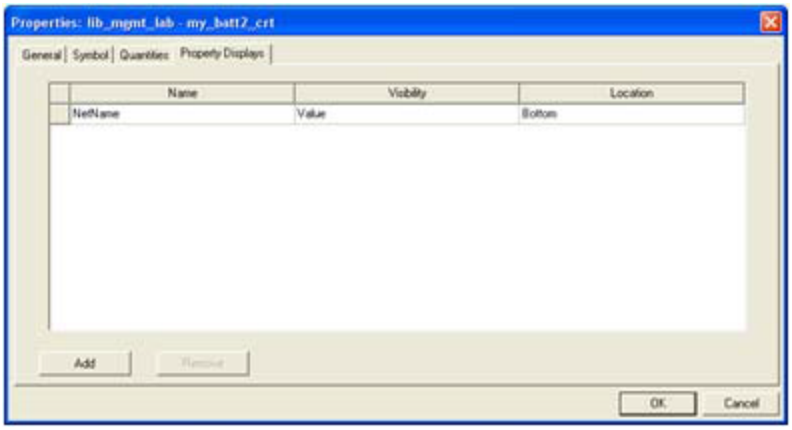


Passive elements (resistors and capacitors) appear under **Circuit/Passive Elements**. The voltage measurement component (**vm**) is found under **Measurement/Electrical**.

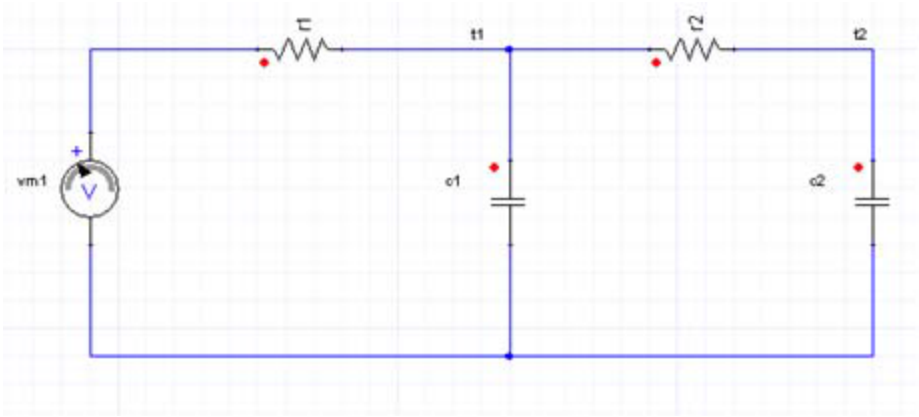
6. Place two resistors, two capacitors, and the voltage measurement component on the schematic for the **SubSheet1:mybatt2_crt** design. Place the cursor over the resistors and then right-click and choose **Rotate** to orient them as shown above. Connect them as shown in the circuit above.
7. Double-click the wire to name the **t1** node.



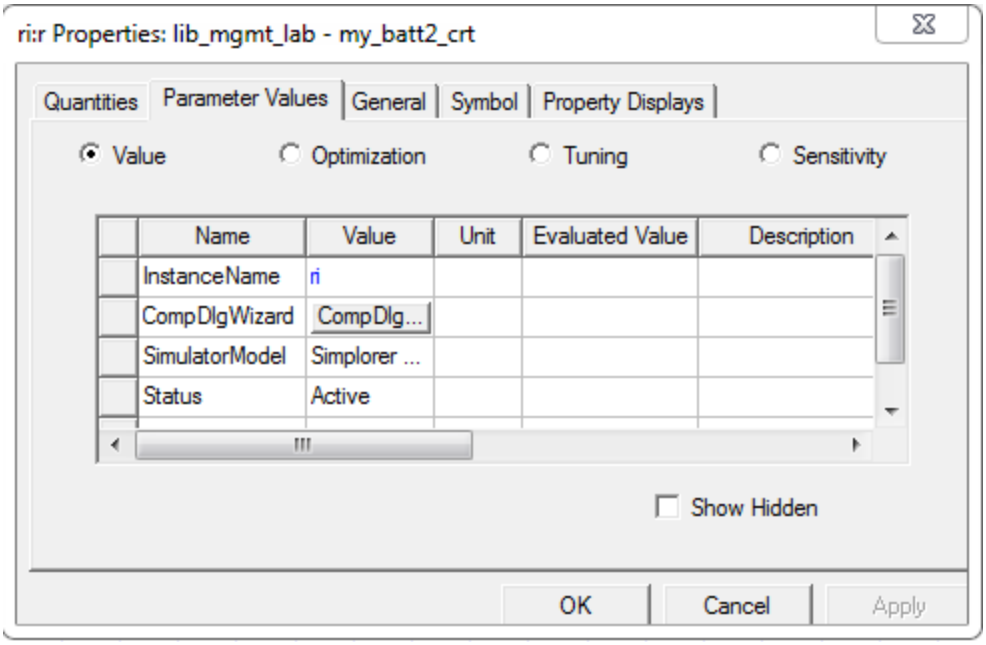
8. Select the **Property Displays** tab on the wire's **Properties** dialog box and click **Add** to display the **NetName** on the schematic.



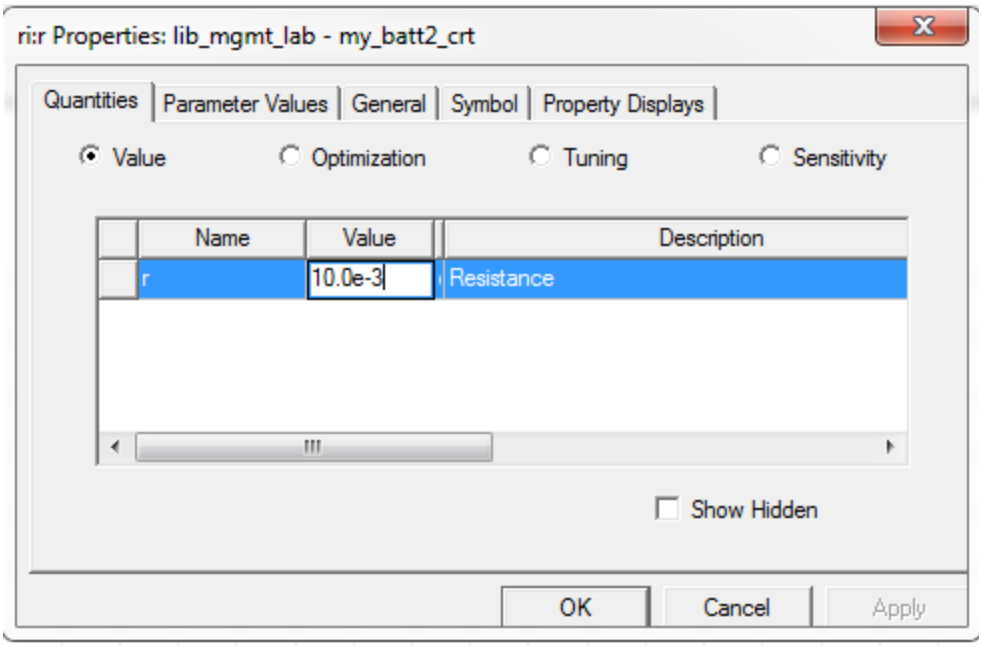
9. Repeat for the node **t2**. Move the names on the schematic (by dragging them) so the circuit appears as follows.



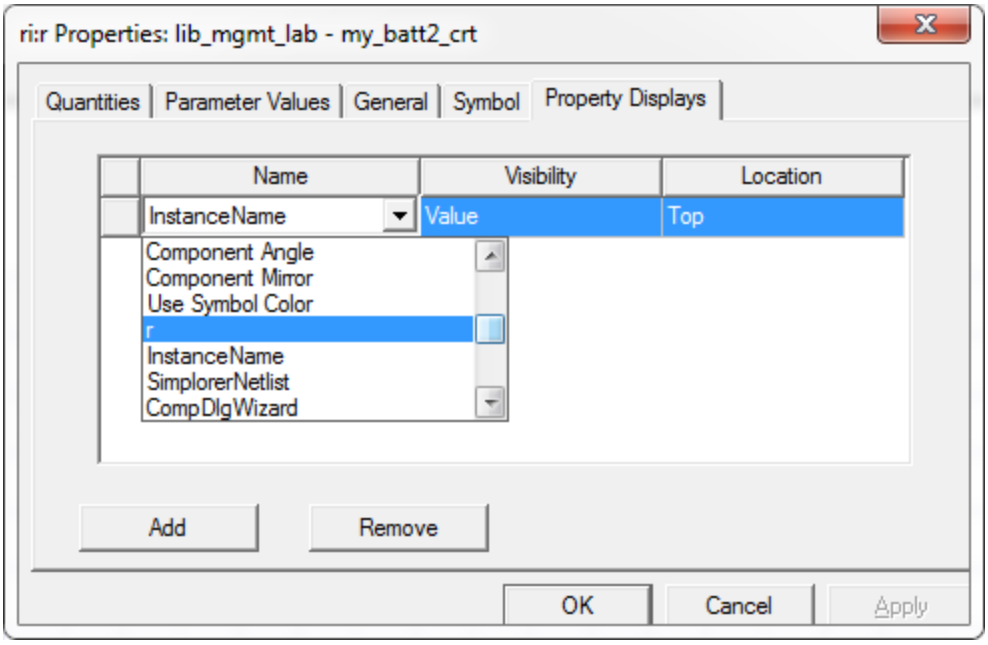
10. Right-click the first resistor and define its properties. Select the **Parameter Values** tab and define the **InstanceName** to be **ri**.



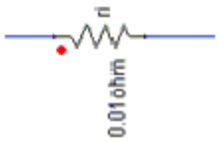
Select the **Quantities** tab and define the resistance value to be **10.0e-3**.



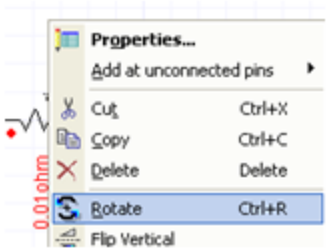
Select the **Property Display** tab and click **Add** to add another display for the "**r**" value.



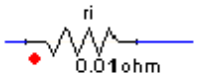
The results should look like the following on the schematic.



11. Rotate the text on the schematic by selecting each text item, right-clicking and choosing **Rotate**.

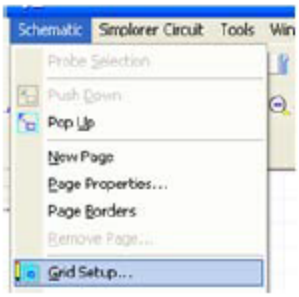


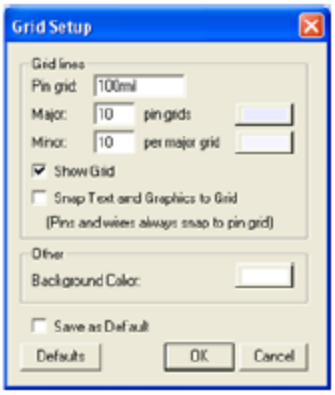
12. The display of the resistor properties should now appear as shown.



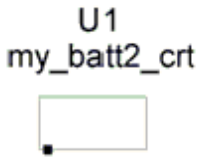
Repeat this process for the other resistance (**rd = 40.0e-3**) and save the project.

> **Note:**
>
> Text placement can be enhanced by adjusting the grid, or by choosing not to have the text and graphics snap to the grid.

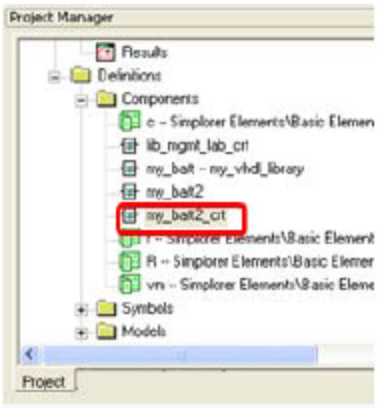




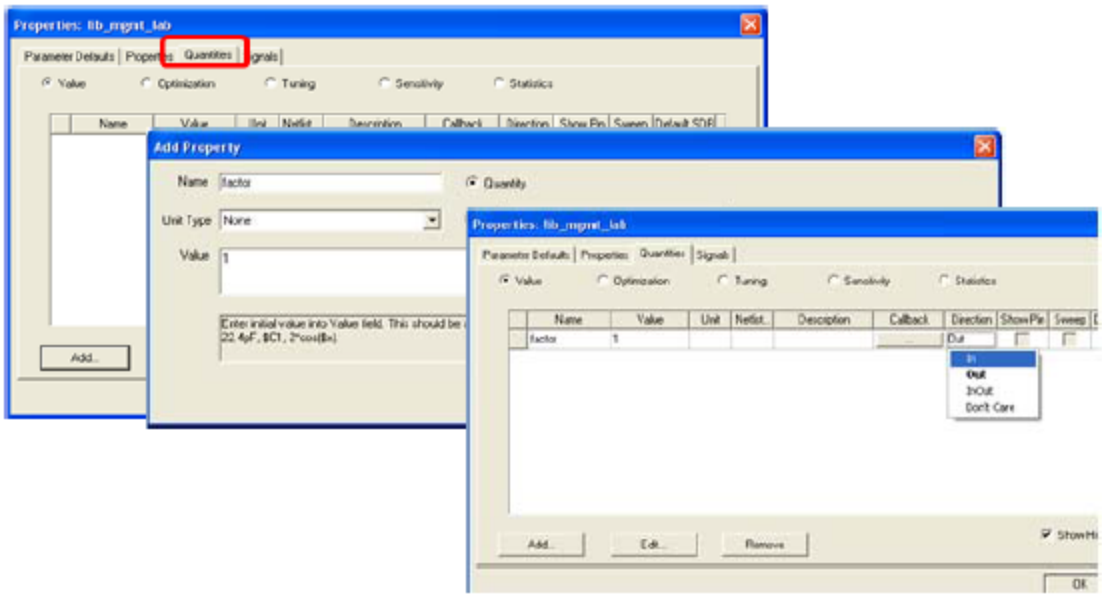
There are two variables (**factor**, and **v_init**) used by the capacitors. These must be declared as quantities that get passed to the model. Select the **Project** tab in the **Project Manager** pane, then double-click the **my_batt2** design. The hierarchical symbol for the **my_batt2_crt** should be seen at this level.



13. Expand the **Definitions/Components** folder in the **Project Manager** pane, and double-click **my_batt2_crt** to open the **Edit Component** dialog box. Click **Properties**.

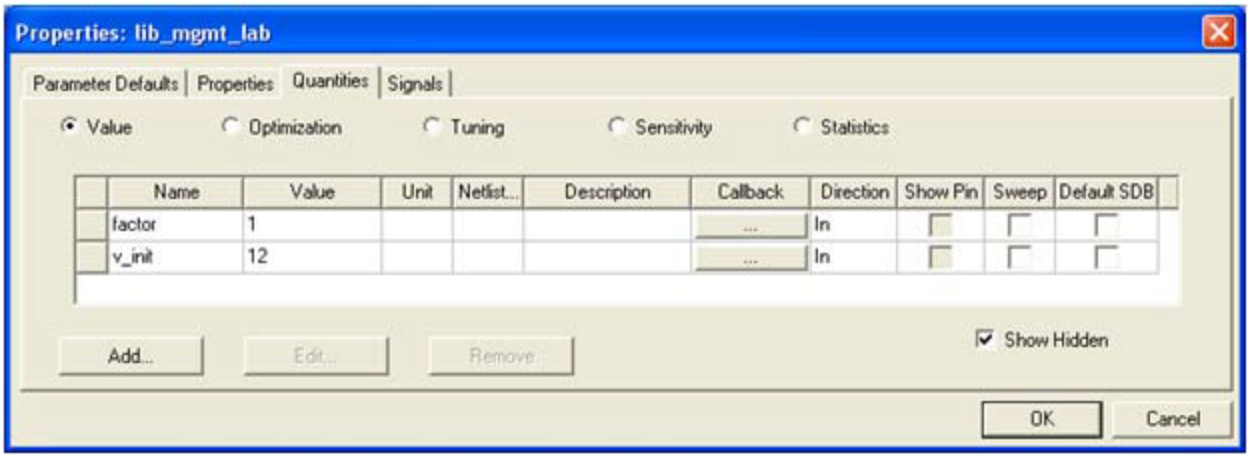


14. Select the **Quantities** tab, then click **Add** to add a quantity to be passed to the hierarchical symbol to the circuit capacitor. Define the name to be **factor** and set the default value to "**1**". Click **OK**. Define the **Direction** to be "**In**".



15. Repeat to add the quantity **v_init** with default value of "**12**" and Direction of "**In**".

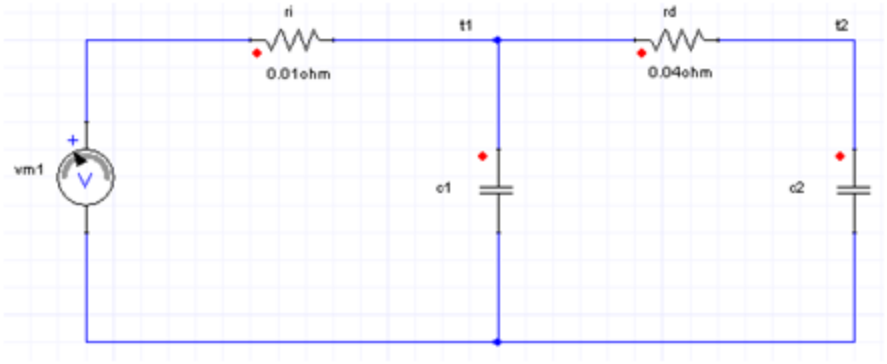
    When done, the **Quantities** tab should appear as shown below.



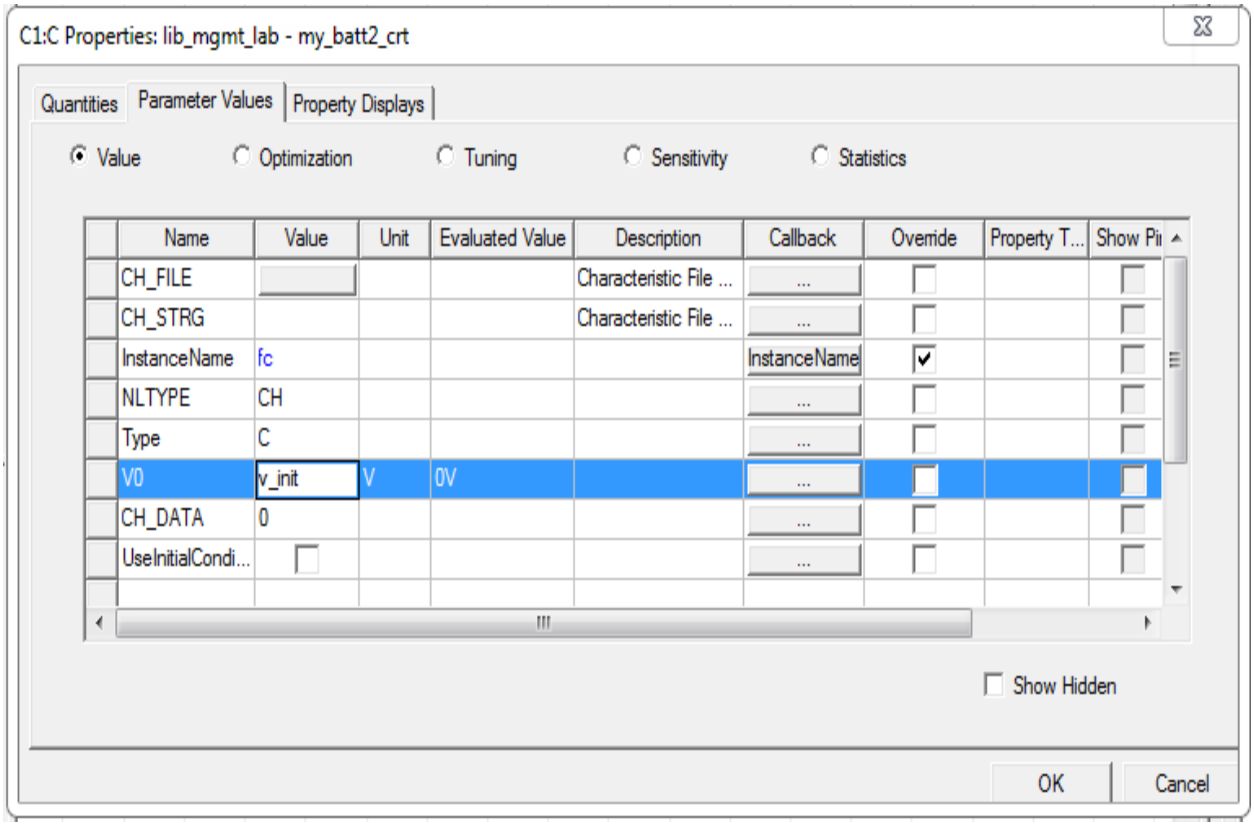
16. Click **OK**, and then **OK** again in the **Edit Component** dialog box. Save the project.

> **Note:**
>
> If you double-click the battery symbol on the schematic and select the **Quantities** tab, you should see these inputs to the battery model.

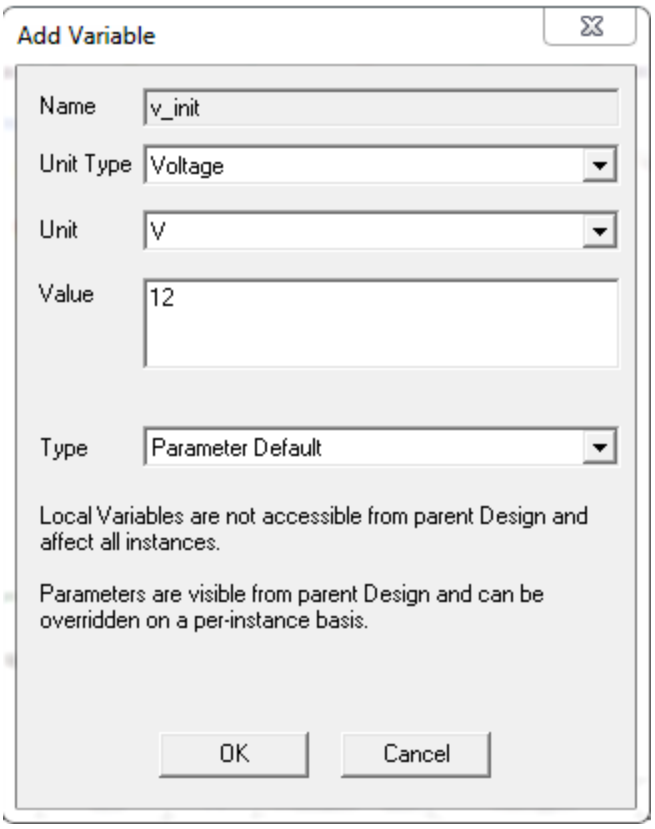
17. In the **Project Manager** pane, double-click the **SubSheet 1: my_batt2_crt** sub-design to get back to the circuit level model of the battery.



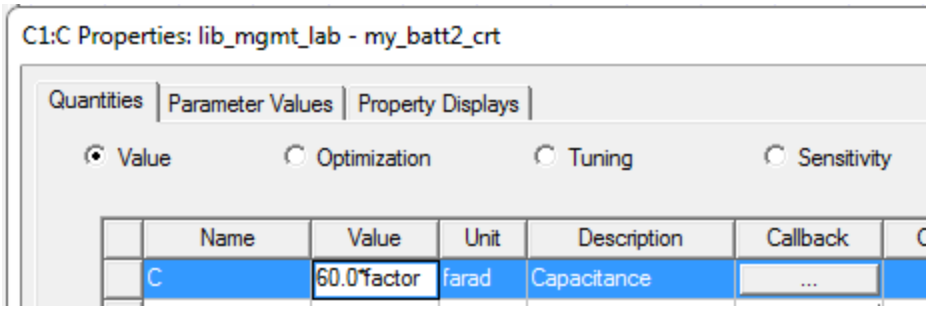
18. Right-click the first capacitor (c1) and define its properties. Select the **Parameter Values** tab and change the **InstanceName** to "**fc**", define the value of "**v0**" (initial voltage) to be the variable **v_init**.



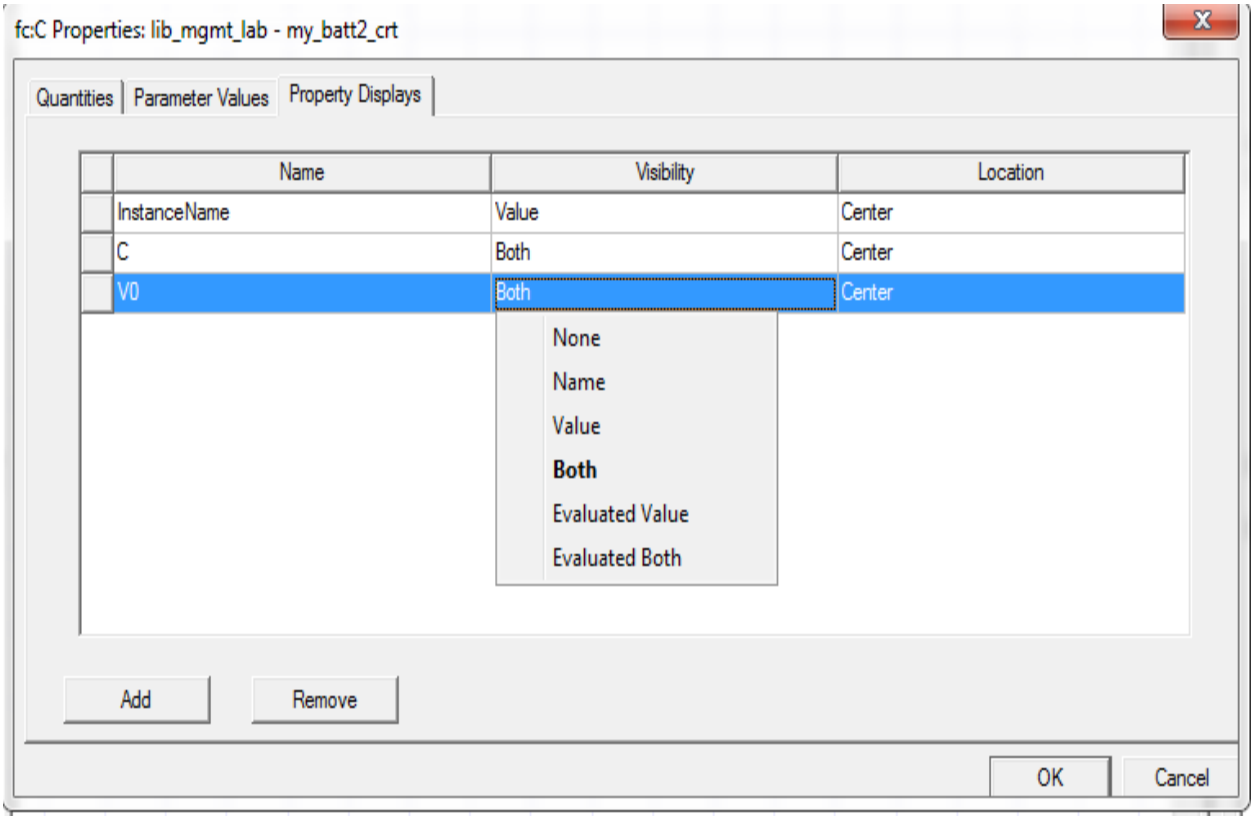
When you click **OK**, an **Add Variable** dialog box appears. Select the **Parameter Default** from the **Type** drop-down list and enter a value of **12**. Click **OK** to close the dialog box.



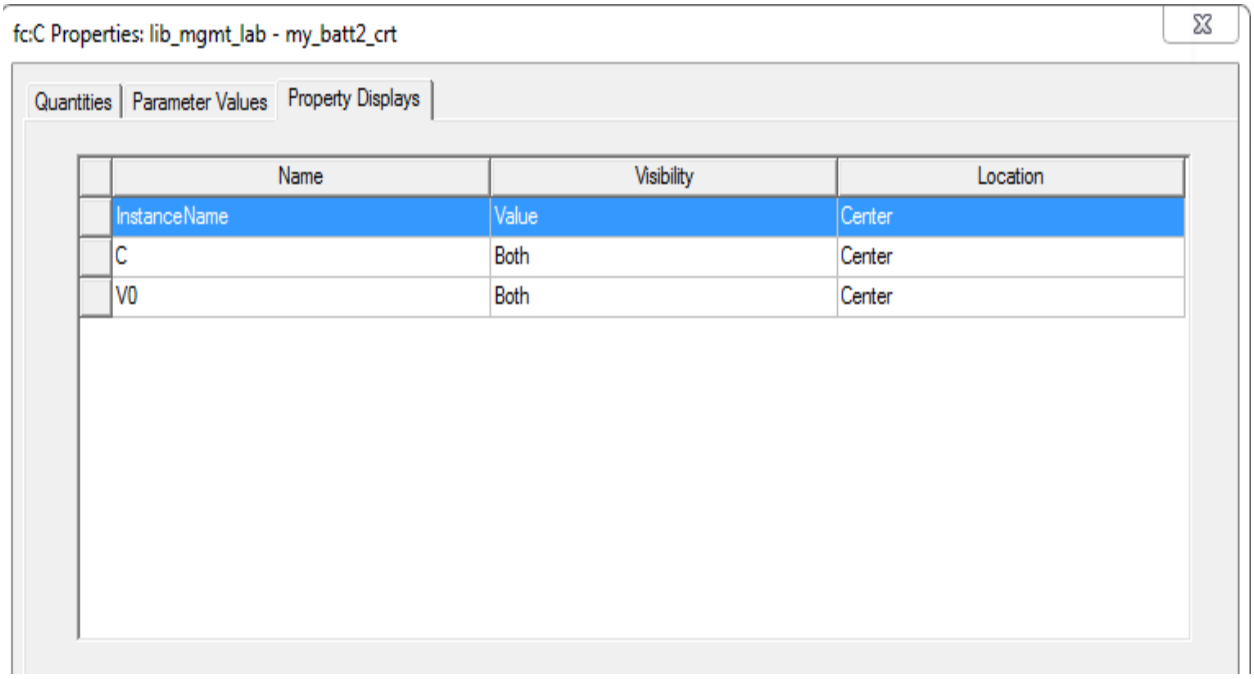
19. Select the **Quantities** tab and define the value to be "**60.0*factor**".



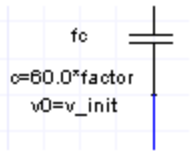
20. Select the **Property Displays** tab and **Add** another two displays for the "**c**" value, and the "**v0**" (initial voltage) value. Select to display **Both** the name and value on the schematic.



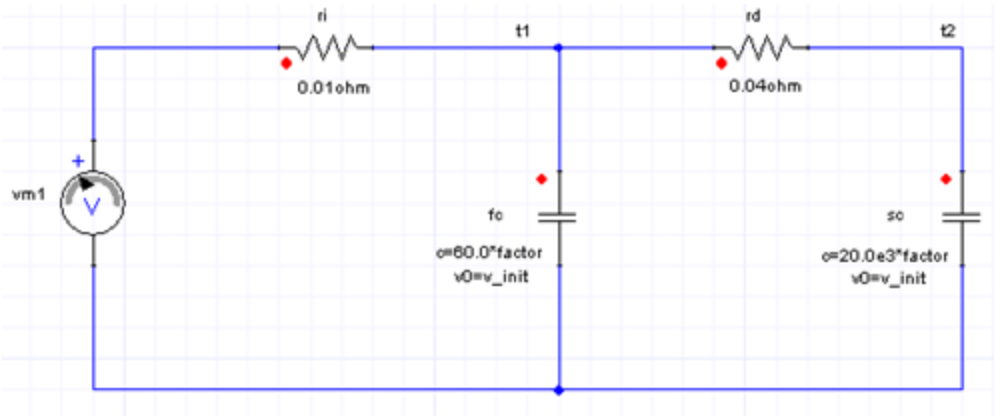
The final **Property Display** tab should look as shown below.



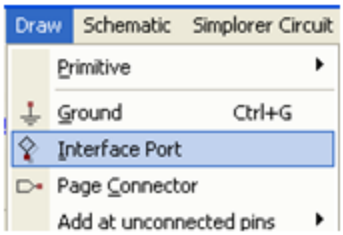
21. The schematic symbol should now appear as shown.



22. Repeat for the other capacitor (**InstanceName=sc**, **c=20.0e3*factor**, **v0=v_init**).
23. Save the project. The schematic should now appear as shown below.

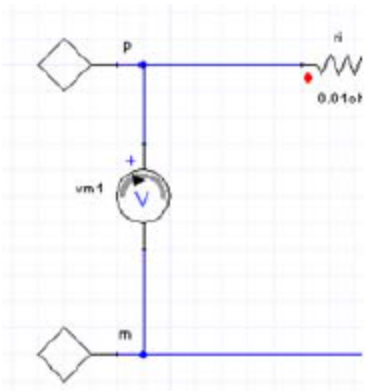


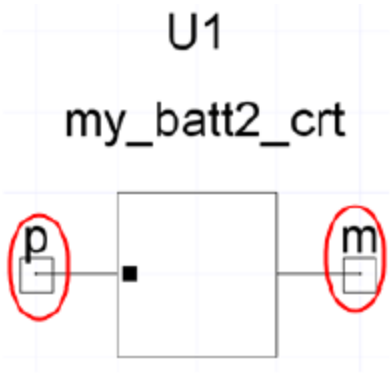
24. Add two Interface Ports ("**p**", and "**m**") to represent the input terminals to the battery model via **Draw/Interface Port** (double-click to edit the port name). This adds pins to the symbol for connection points into the battery circuit.



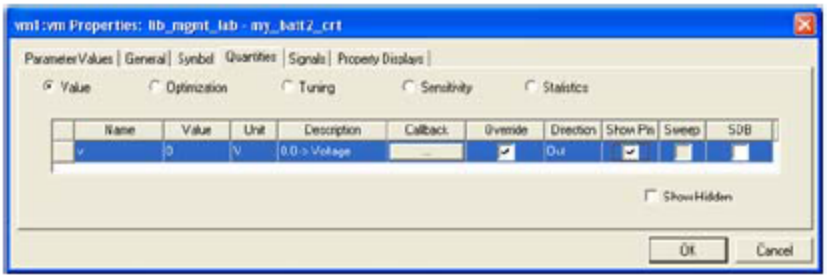
> **Note:**
>
> These Interface Ports can also be used to pass information in and out of the subcircuit as Quantities. (If this method is used, the defined Quantities can be displayed as pins as well.)

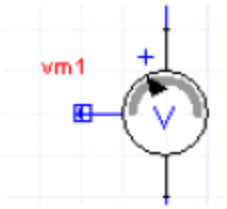
The results should appear as shown below. Note the symbol in the **my_batt2** design now shows the new pins.



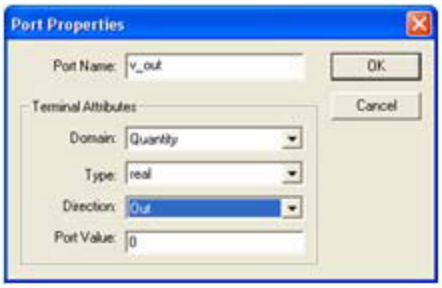


25. Add an interface pin for the output voltage of the voltage measurement component. First, select to "show" the output "pin" for the voltage measurement component "**vm1**" by double-clicking on the **vm1** symbol, selecting the Quantities tab and selecting to **Show Pin**. Click **OK** to close the dialog box. Note the symbol on the schematic now shows the output pin.

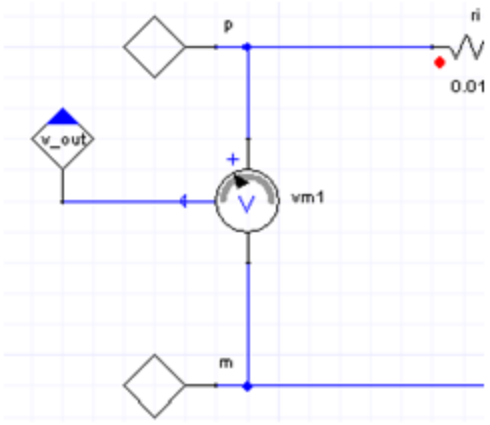




26. Add another Interface Port (to provide access to the battery's terminal voltage measurement), and then double-click it to define the properties of the port. Because this port will *not* be connected to a conservative pin, we define its Terminal Attributes where **Domain = Quantity**, **Type = Real**, and **Direction = Out**. Change the **Port Name** to **v_out** then connect it as shown below.

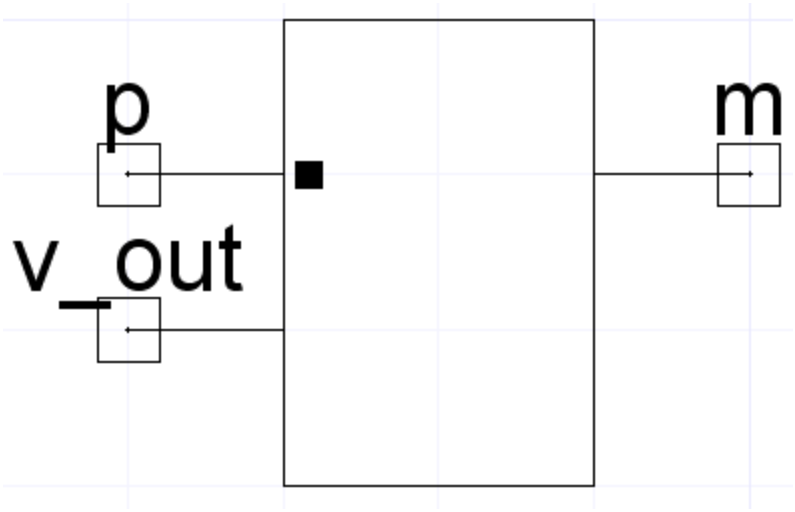


27. Connect **v_out** as shown below and save the project.

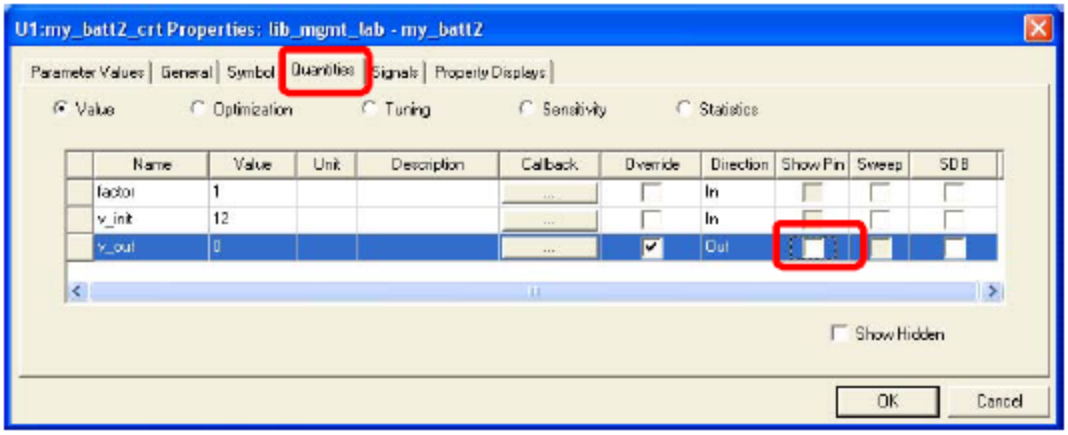


Note that the symbol in the **my_batt2_crt** design level now shows three pins for the three interface ports that were placed on the underlying schematic circuit (two conservative pins

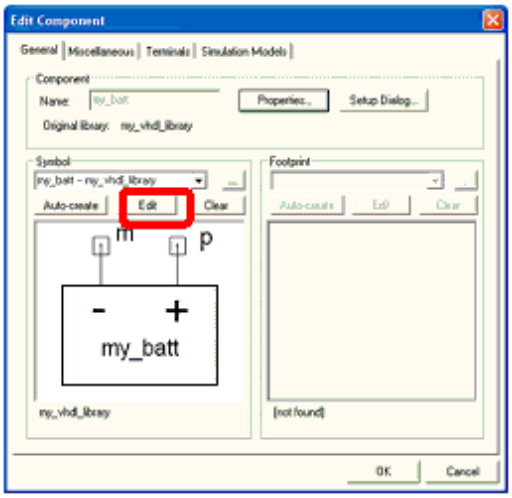
"**p**", "**m**" that represent the battery models terminals, and one non-conservative pin "**v_out**" that represents the voltage measurement across the battery terminals).



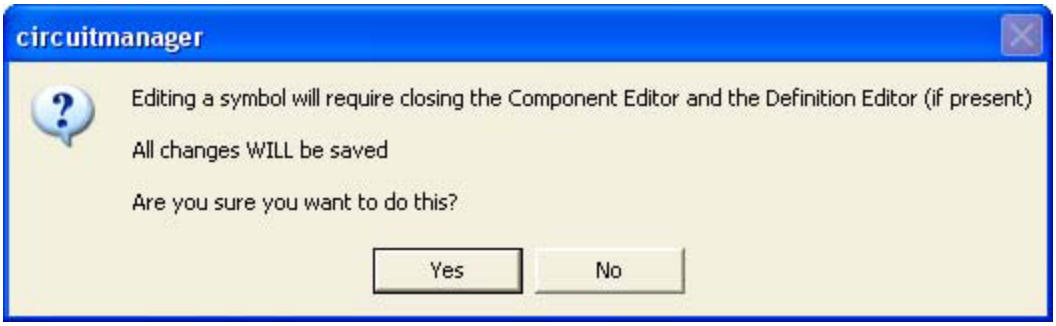
28. Double-click the battery symbol to open its properties dialog box, select the **Quantities** tab and deselect the **Show Pin** box for **v_out**. Click **OK** and save the design. The symbol should now show only the "**p**" and "**m**" pins.



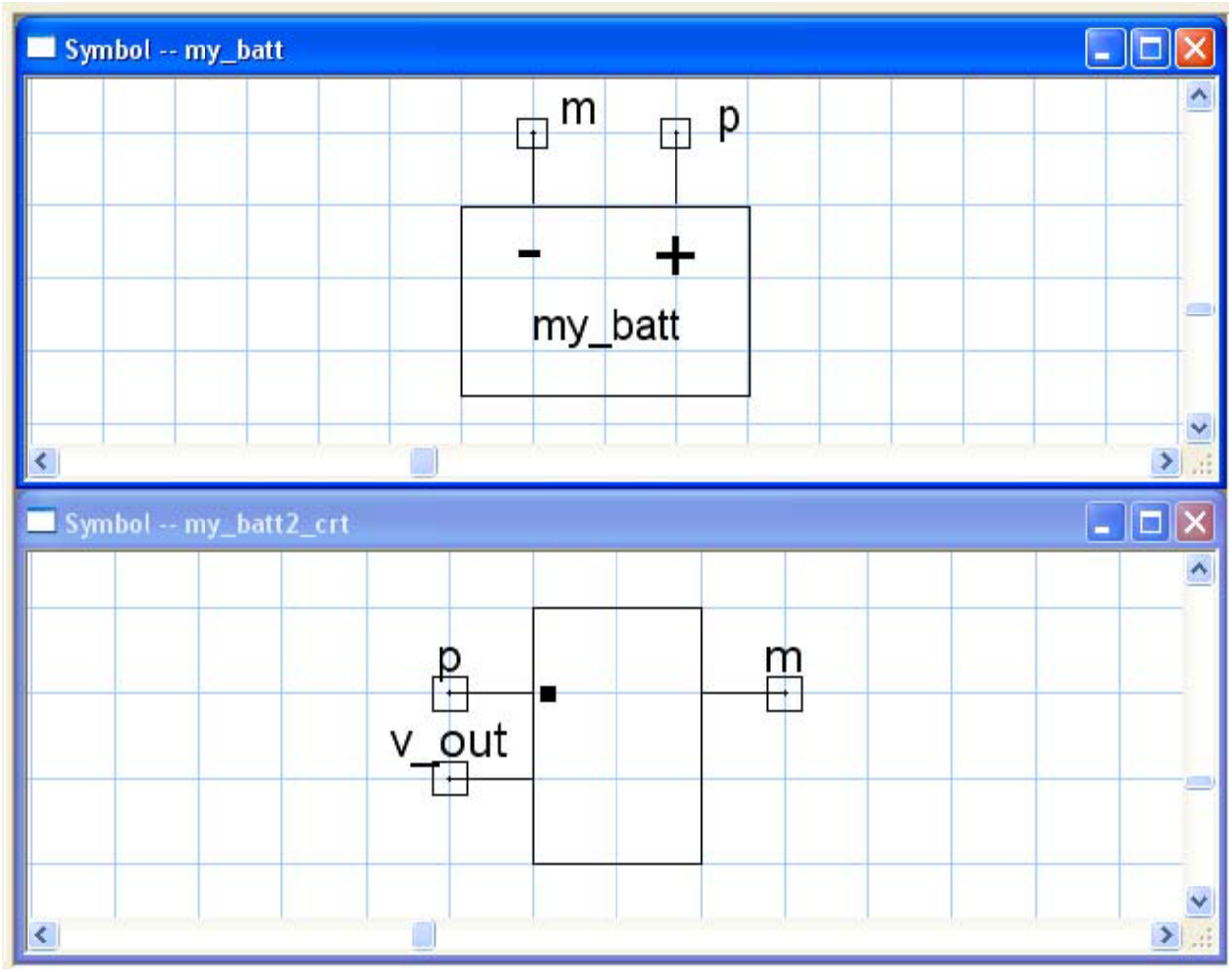
29. Edit the hierarchical battery model symbol based on the symbol created in the previous **Example #1**; Select the default battery model symbol in the **my_batt2** design, then right-click and choose **Edit Symbol**. This opens the symbol in the symbol editor window.
30. Copy and paste the battery symbol previously created in **Example #1**. In the **Component Libraries** window, expand the **my_vhdl_library** created in **Example #1**. Select the **my_batt** component, then right-click and select **Edit Component**. Under the Symbol section, select **Edit** to bring in its symbol as well.



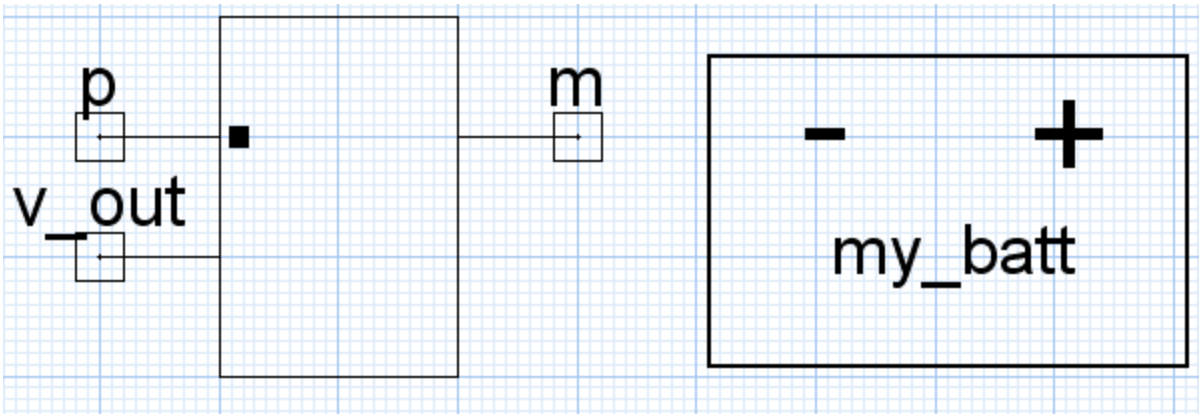
31. The following dialog box appears. Click **Yes**.



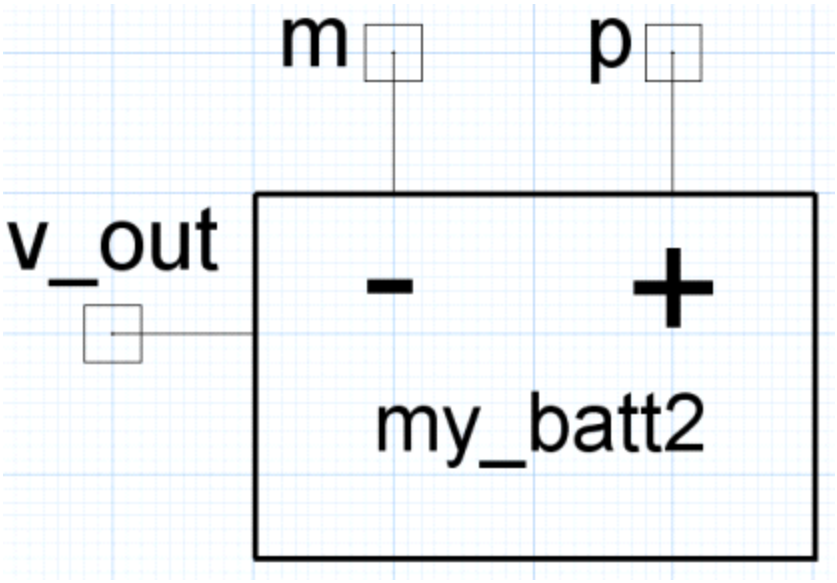
32. Cascade the two symbol edit windows side by side so you can copy and paste from one to the other using the menu item **Window** > **Cascade**.
33. Close all other windows, and then select **Window** > **Tile Horizontally** to view the two symbol editing windows simultaneously.



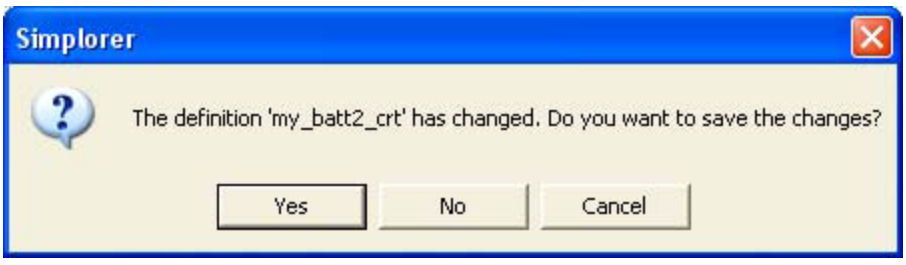
34. Select the body symbol from **my_batt**, copy it and paste it into the **my_batt2_crt** symbol window. Close the **my_batt** symbol window. You should now have only the **my_batt2_crt** symbol editing window with the copied graphics open.



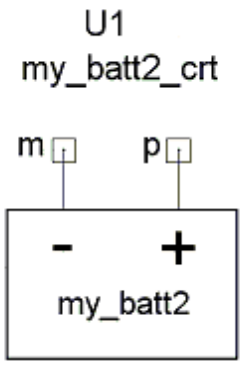
35. Move the pins from the default battery symbol, change the text name on the symbol to be **my_batt2**, delete extra graphics. The symbol should now appear as shown.



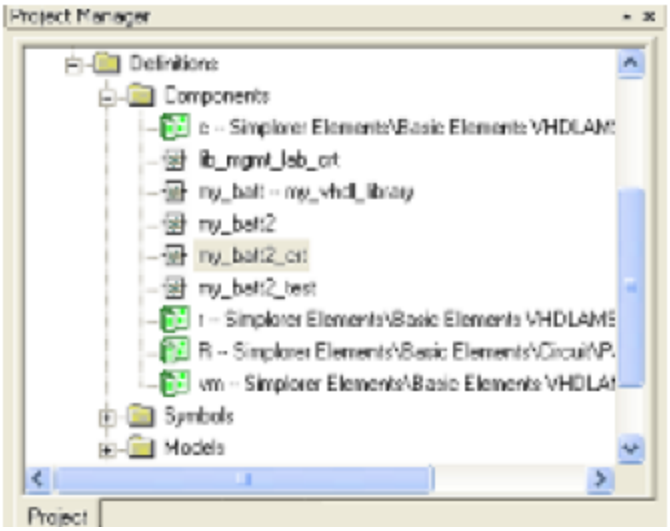
36. Save the new symbol. When the following window appears, click **Yes** to save changes.



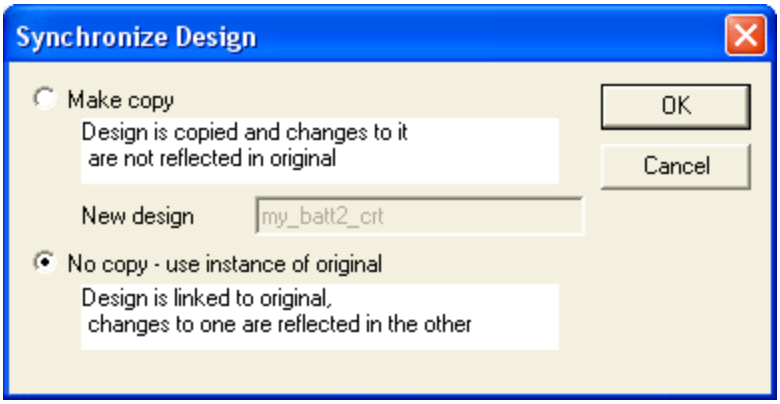
37. Select the **Project** tab in the **Project Manager** pane and double-click the **my_batt2** design. This should now also reflect the changes made to the symbol. Note the **v_out** non-conservative pin still doesn't appear due to the previous step of deselecting the **Show Pin** box.



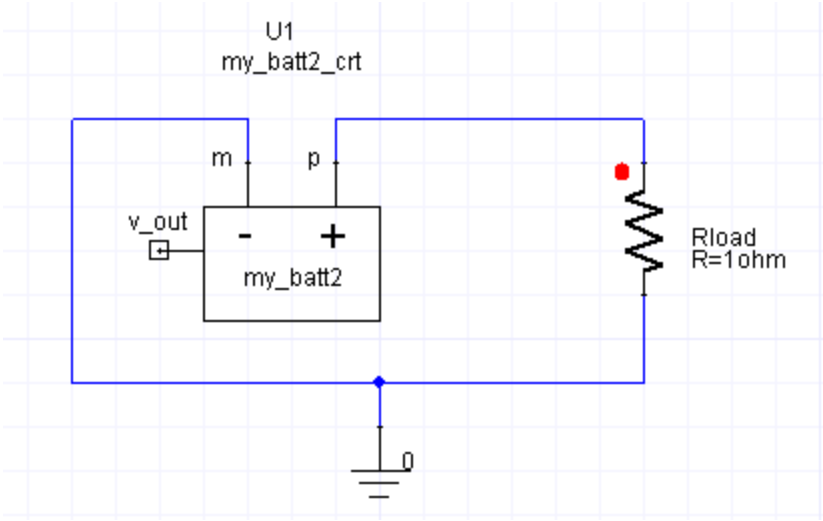
38. Create a new Twin Builder design, **my_batt2_test**, to verify the new hierarchical battery model using the same test circuit from **Example #1** (that is, use a one ohm resistance in parallel with the battery, same TR analysis set up, same parameters for the battery model, and so on). You can select the design from **Example #1**, **lib_mgmt_lab_crt**, right-click to copy, select the project to place it in, **lib_mgmt_lab**, then right-click and paste. This creates a new design, **lib_mgmt_lab_crt1**. Rename the copied version **my_batt2_test**.
39. Double-click the new design, **my_batt2_test**, to open it. Delete the on-schematic report graph and the **my_batt** symbol if you copied the design.
40. Place the new **my_batt2_crt** hierarchical battery model by selecting it from the **Definitions/Components** folder in the **Project** tab as shown below, and dragging it into the new design schematic, **my_batt2_test**.



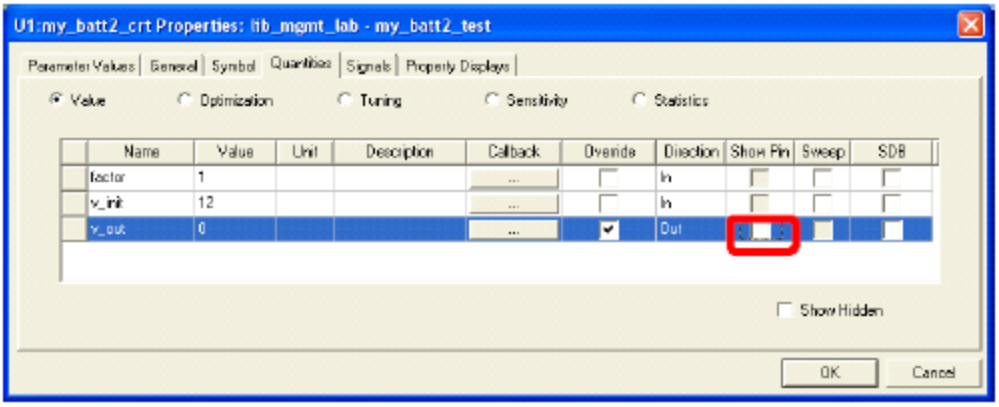
Make the **Synchronize Design** dialog box selection as shown below.



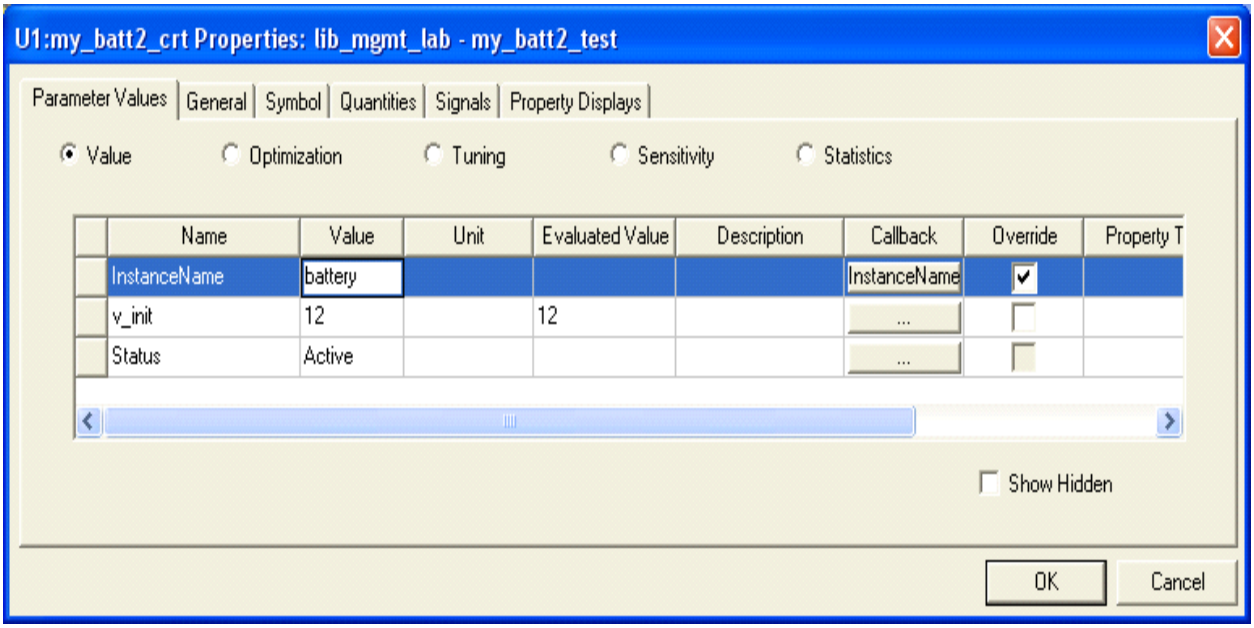
The test schematic should look as shown below. Note the non-conservative **v_out** pin is visible on the schematic.



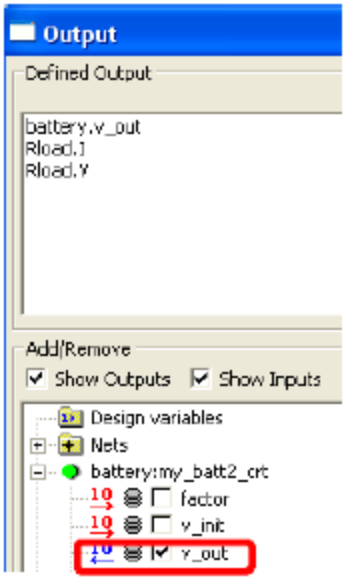
41. Double-click the battery symbol and clear the **Show Pin** box under the **Quantities** tab for **v_out**.



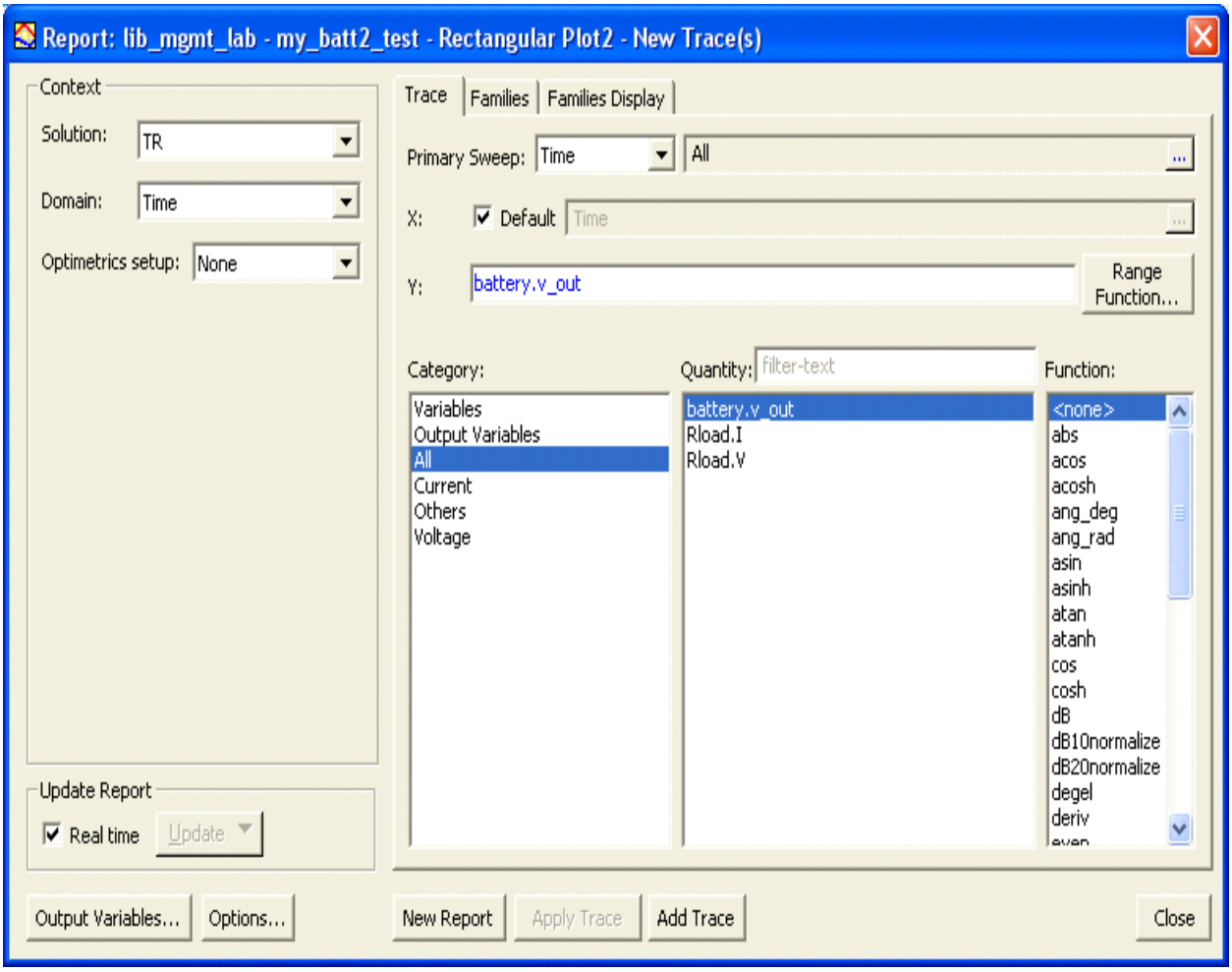
42. Select the Parameter Values tab and define the **InstanceName** to be **battery**. Keep **v_init** at its default value of **12** and click **OK**.



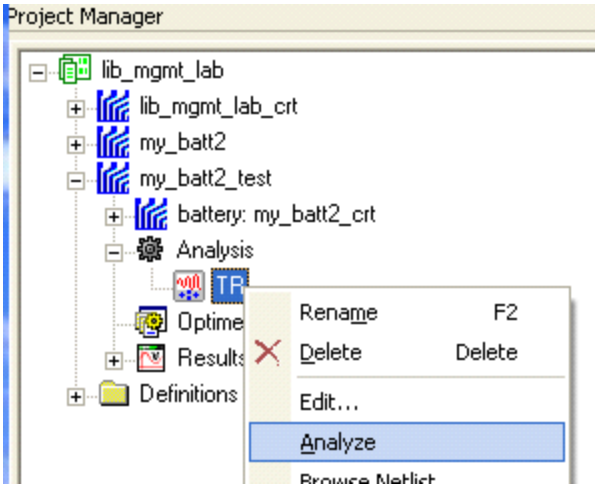
43. Next, define the outputs to make them available for plotting. Click **Twin Builder** > **Output Dialog…** to open the **Output** dialog box, then add the **v_out** signal by selecting it.



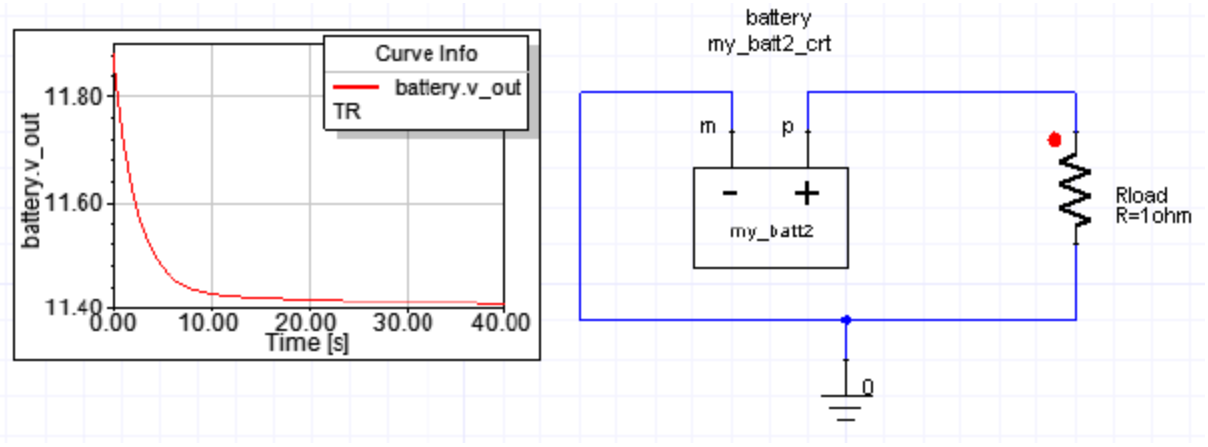
44. Zoom out on the schematic to make room for a plot. Add a report plot onto the schematic using the **Draw** > **Report** > **Rectangular Plot** menu item.
45. Double-click the report plot to define the signal. Select the battery output, **battery.v_out**, click **Add Trace**, then click **Close**.



46. Run the analysis by right-clicking the **TR** setup and selecting **Analyze**.



The results are shown below.

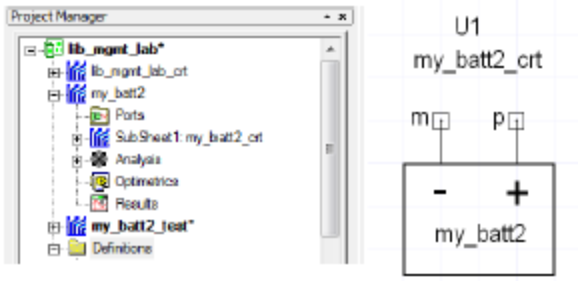


Note:

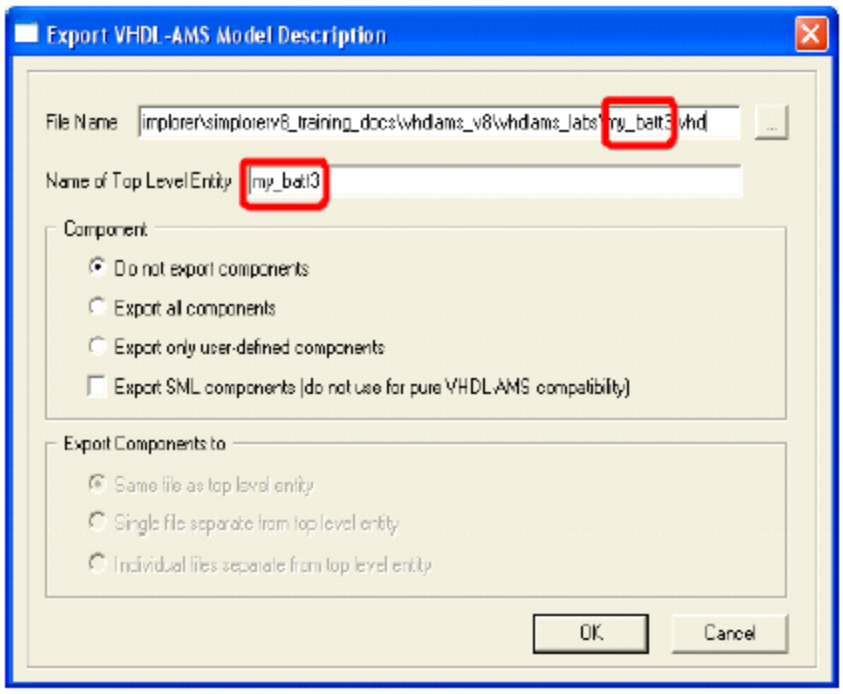
Because passing information into the subcircuit was set up using Quantities, changing values for **factor** or **v_init** should be done by double-clicking the battery symbol to open its **Properties** dialog box, then make the changes on the **Quantities** tab.

Exporting a Hierarchical VHDL-AMS Subcircuit to a single *.vhd file

1. Double-click the **my_batt2** design in the **Project Manager** pane. Observe that the hierarchical symbol for the battery model exists.



2. Export the hierarchical VHDL-AMS battery model to a single ***.vhd** file with the **Tools** > **Design Tools** > **Export to VHDL-AMS** menu item. Change the name to **my_batt3**.



3. Clean up all unused project definitions by selecting **Tools** > **Project Tools** > **Remove Unused Definitions**. Click **Select All** and **Apply** until all unused definitions are removed. Locate the **my_batt3.vhd** file; it should appear as shown below.

```
LIBRARY ieee;

USE ieee.electrical_systems.all;

USE ieee.fundamental_constants.all;

USE ieee.math_real.all;

LIBRARY basic_vhdlams;


ENTITY my_batt2_crt IS

GENERIC (

  v_init : voltage := 12.0

     );

PORT (

  TERMINAL m : ELECTRICAL;

  TERMINAL p : ELECTRICAL;
```

```
    QUANTITY v_out : OUT voltage;
    QUANTITY factor : IN REAL := 1.0
  );
  BEGIN
  END ENTITY my_batt2_crt;

  ARCHITECTURE struct OF my_batt2_crt IS
    TERMINAL t2 : ELECTRICAL;
    TERMINAL t1 : ELECTRICAL;
    QUANTITY Q_2 : voltage := 0.0;
  BEGIN
    fc : ENTITY basic_vhdlams.c(behav)
      GENERIC MAP ( use_v0 => true , v0 => v_init )
      PORT MAP ( m => m, p => t1, c => 60.0*factor -- Expression
      );

    sc : ENTITY basic_vhdlams.c(behav)
      GENERIC MAP ( use_v0 => true , v0 => v_init )
      PORT MAP ( m => m, p => t2, c => 20000.0*factor --
Expression
      );

    ri : ENTITY basic_vhdlams.r(behav)
      PORT MAP ( m => t1, p => p, r => 0.01 );

    rd : ENTITY basic_vhdlams.r(behav)
      PORT MAP ( m => t2, p => t1, r => 0.04 );
```

```
  vm1 : ENTITY basic_vhdlams.vm(behav)
    PORT MAP ( m => m, p => p, v => Q_2 );

  v_out == Q_2;
END ARCHITECTURE struct;
```

Note:

The following part of the VHDL-AMS code was generated for the top-level circuit and should be deleted (since we just want the code for the hierarchical subcircuit battery model that was shown above).

```
LIBRARY ieee;
USE ieee.electrical_systems.all;
USE ieee.fundamental_constants.all;
USE ieee.math_real.all;
LIBRARY basic_vhdlams;

ENTITY my_batt3 IS
BEGIN
END ENTITY my_batt3;

ARCHITECTURE struct OF my_batt3 IS
  TERMINAL unconnected1 : ELECTRICAL;
  TERMINAL unconnected0 : ELECTRICAL;
  QUANTITY Q_2 : voltage := 0.0;
BEGIN
  U1: ENTITY WORK.my_batt2_crt(struct)
    GENERIC MAP ( v_init => 1.200000e+001 )
```

```
        PORT MAP ( m => unconnected1, p => unconnected0, v_out => Q_
2 , factor => 1.000000e+000 );

    END ARCHITECTURE struct;
```

5. Now edit the remaining VHDL-AMS code for the battery model by changing the name of the Entity to **my_batt3**.

   Below is the final version of the VHDL-AMS code that was originally generated from the hierarchical subcircuit schematic version of the battery model. The “top-level circuit” part of the code has been deleted, and the name for the Entity changed to **my_batt3**.

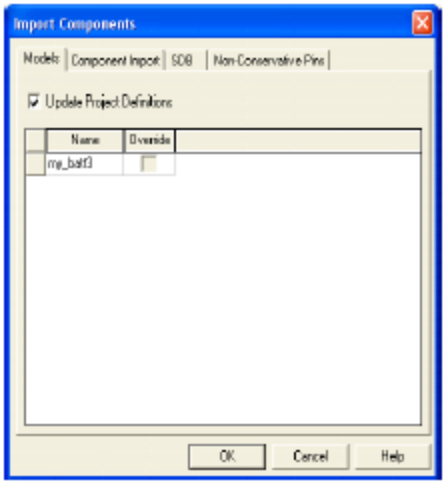
```
LIBRARY ieee;

USE ieee.electrical_systems.all;

USE ieee.fundamental_constants.all;

USE ieee.math_real.all;

LIBRARY basic_vhdlams;


ENTITY my_batt3 IS

  GENERIC (

    v_init : voltage := 12.0

  );

  PORT (

    TERMINAL m : ELECTRICAL;

    TERMINAL p : ELECTRICAL;

    QUANTITY v_out : OUT voltage;

    QUANTITY factor : IN REAL := 1.0

  );

BEGIN

END ENTITY my_batt3;


ARCHITECTURE struct OF my_batt3 IS
```

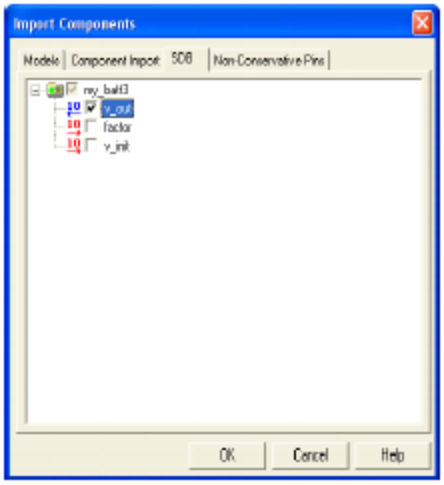
```
    TERMINAL t2 : ELECTRICAL;
    TERMINAL t1 : ELECTRICAL;
    QUANTITY Q_2 : voltage := 0.0;
  BEGIN
    fc : ENTITY basic_vhdlams.c(behav)
      GENERIC MAP ( use_v0 => true , v0 => v_init )
      PORT MAP ( m => m, p => t1, c => 60.0*factor -- Expression
      );

    sc : ENTITY basic_vhdlams.c(behav)
      GENERIC MAP ( use_v0 => true , v0 => v_init )
      PORT MAP ( m => m, p => t2, c => 20000.0*factor --
Expression
      );

    ri : ENTITY basic_vhdlams.r(behav)
      PORT MAP ( m => t1, p => p, r => 0.01 );

    rd : ENTITY basic_vhdlams.r(behav)
      PORT MAP ( m => t2, p => t1, r => 0.04 );

    vm1 : ENTITY basic_vhdlams.vm(behav)
      PORT MAP ( m => m, p => p, v => Q_2 );

    v_out == Q_2;
  END ARCHITECTURE struct;
```

Import the subcircuit based VHDL-AMS battery model

Importing the **my_batt3.vhd** subcircuit-based VHDL-AMS battery model into Twin Builder lets you have a single-level text model to import into your personal library.

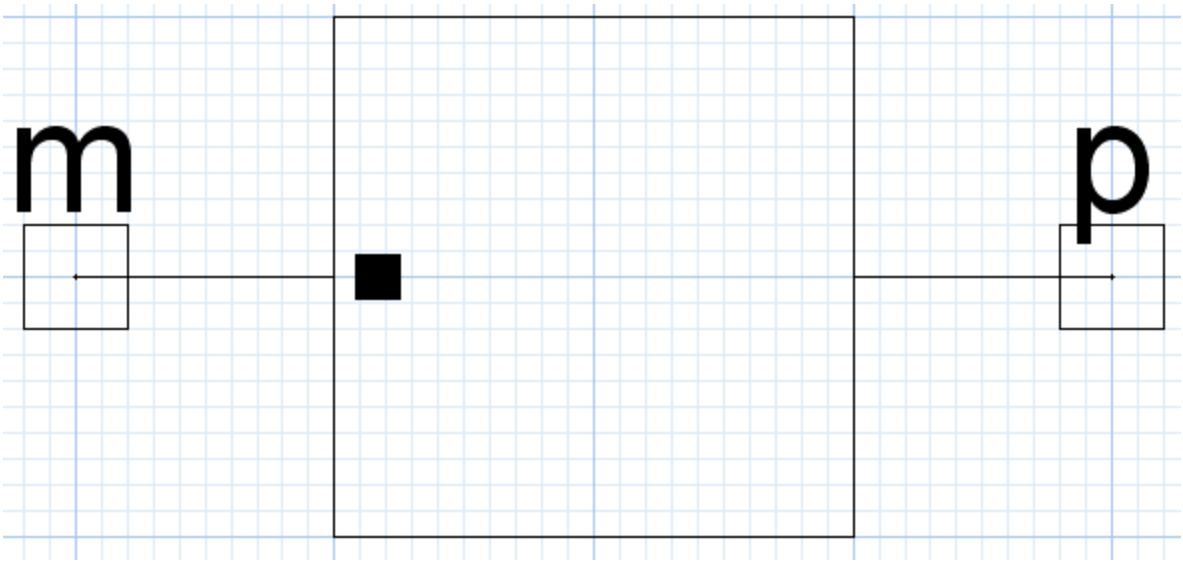
1. Import **my_batt3.vhd** into Twin Builder using **Tools** > **Project Tools** > **Import Simulation Models**. Select the **SDB** tab and select **v_out**. This sets **v_out** to be saved to the database when this model is used in Twin Builder. Leave all others at their default values. Click **OK.**



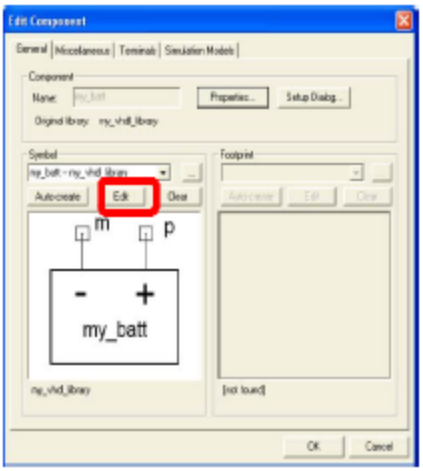


   Several definitions for **my_batt3** should now show up under the **Definitions/Components**, **Definitions/Symbols**, and **Definitions/Models** folders in the **Project Manager** pane **Project** tab.

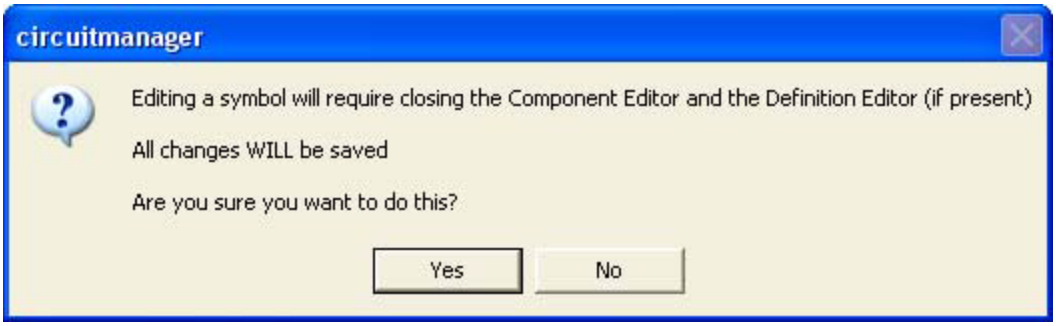
2. Edit the **my_batt3** symbol by double-clicking the file in the **Definitions/Symbols** folder. The following default symbol should appear:



3. Bring in the battery symbol created in **Example #1** (for use in this example); In the **Component Libraries** window, expand the Personal Library **my_vhdl_library** created in **Example #1**. Select the **my_batt** component, and then right-click and select **Edit Component**. In the **Symbol** section, click **Edit** to bring in its symbol as well.

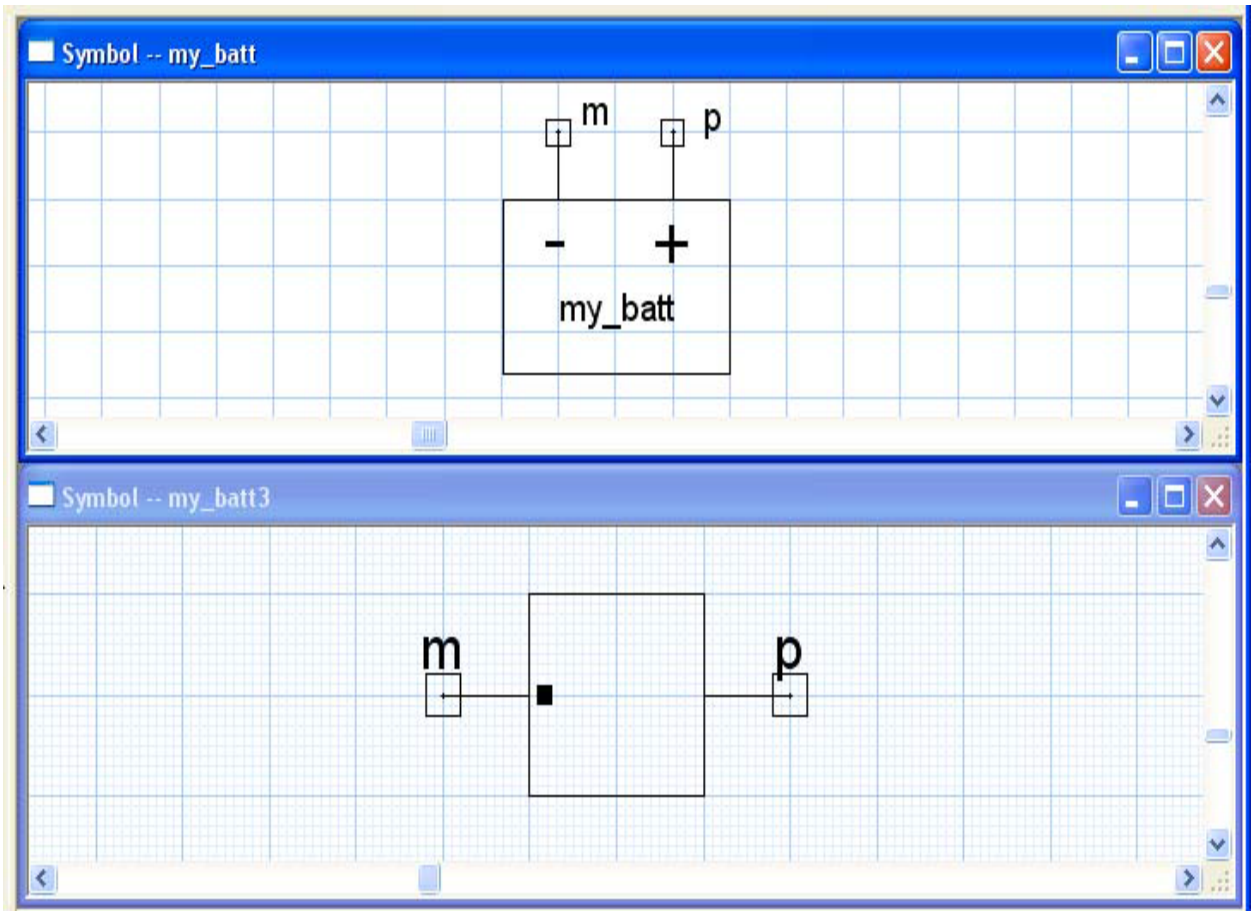


The following dialog box appears.

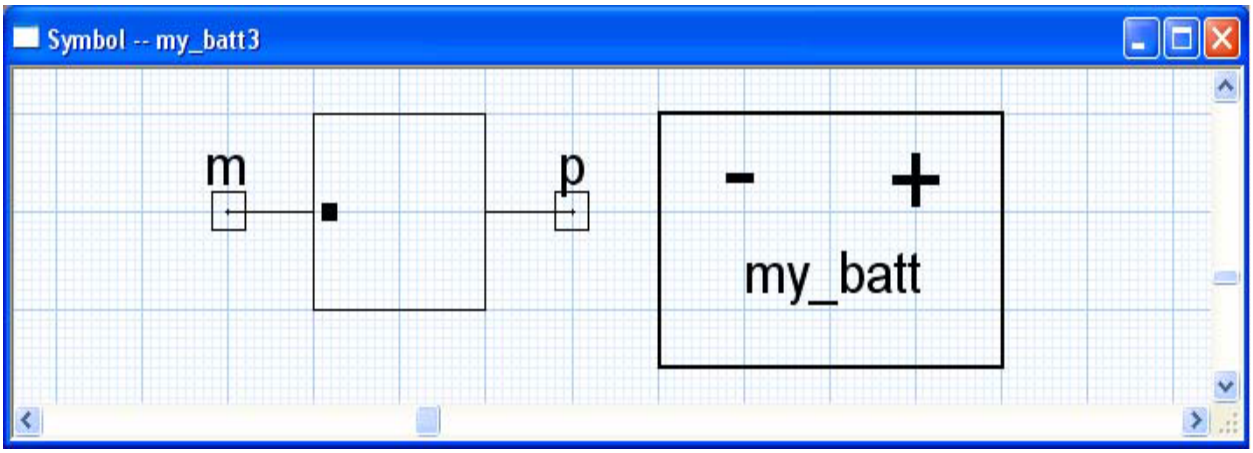


4. Click **Yes**.
5. Cascade the editing windows using the **Window/Cascade** menu item. Close all other editing windows.
6. Tile the two symbol windows horizontally to allow easy copy and paste

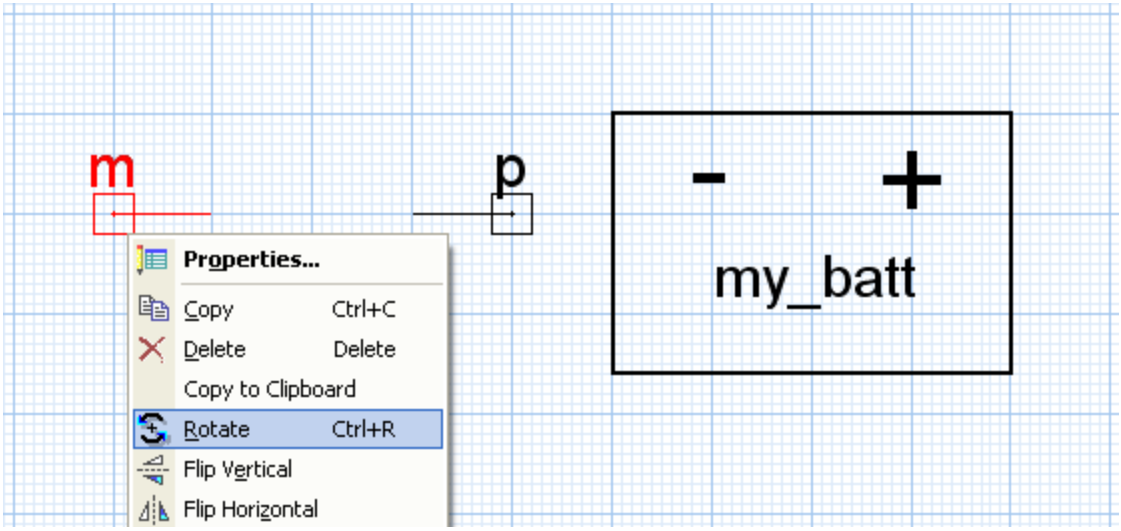
   The following should now appear:



7. Select the graphics from the **my_batt** symbol window, and copy and paste it into the **my_batt3** symbol window. You should now see the following in the **my_batt3** symbol edit window.

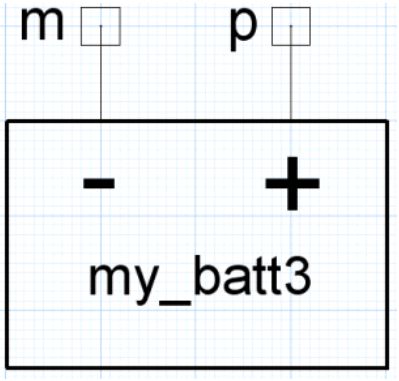


8. Close the other symbol editing window.
9. Delete unwanted graphics and rotate the two pins by right-clicking and using **Rotate** and **Flip Vertical** as needed.

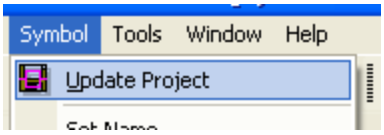


10. Rotate the pin names in similar manner.

11. Move the pins and rename the text so that the symbol now appears as follows:



12. Update the Project Symbol Definition for **my_batt3** via **Symbol>Update Project**, then close the symbol editing window.

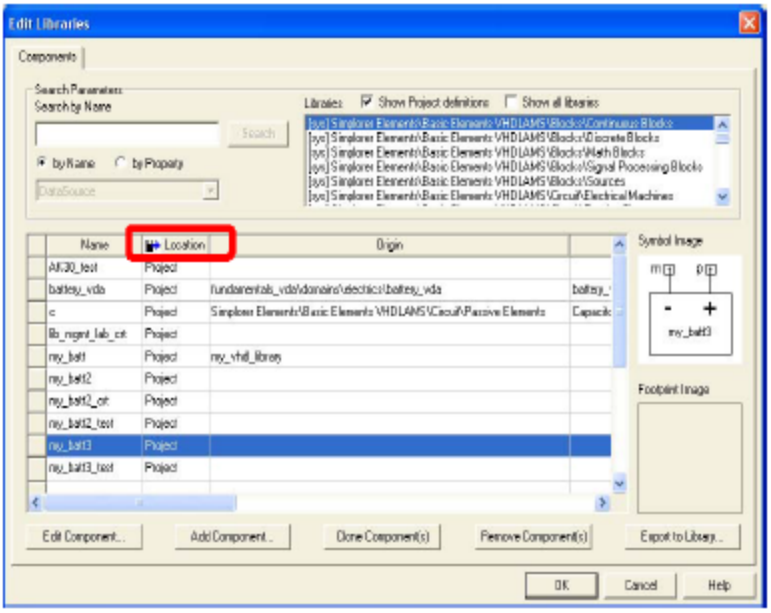


Exporting the my_batt3 component to a personal library

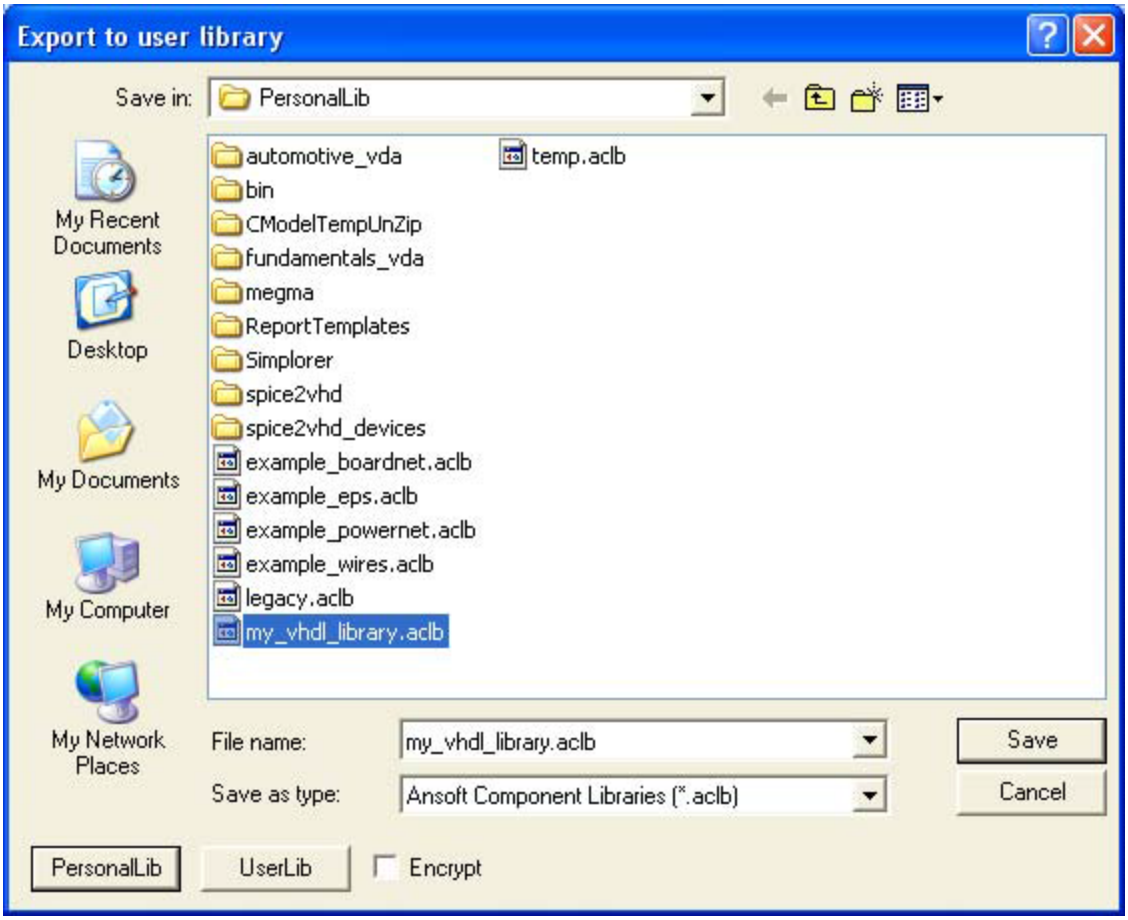
Now that the **my_batt3** VHDL_AMS model definition has been imported, a symbol and component definition defined for it, now “export” it into the personal library “my_vhdl_library” created in "Example #1" on page 5-7 .

1. Insert a new Twin Builder design and rename it **my_batt3_test**.
2. Select **Tools** > **Edit Libraries** > **Components**.

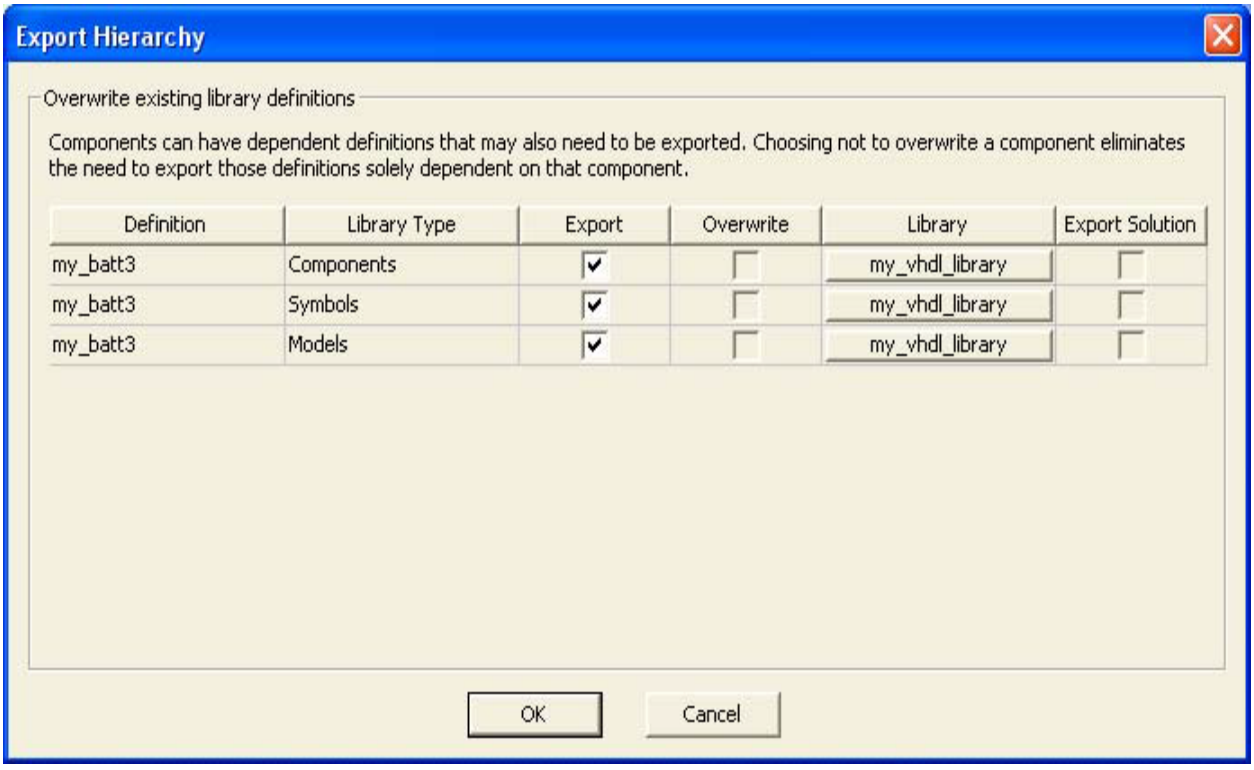
   The following figure shows the **Edit Libraries** dialog box. The **Project** definitions are listed for the library selected in the list at the top. You can sort the list by selecting the **Location** icon title. You should see the **my_batt3** component show up under the project location.



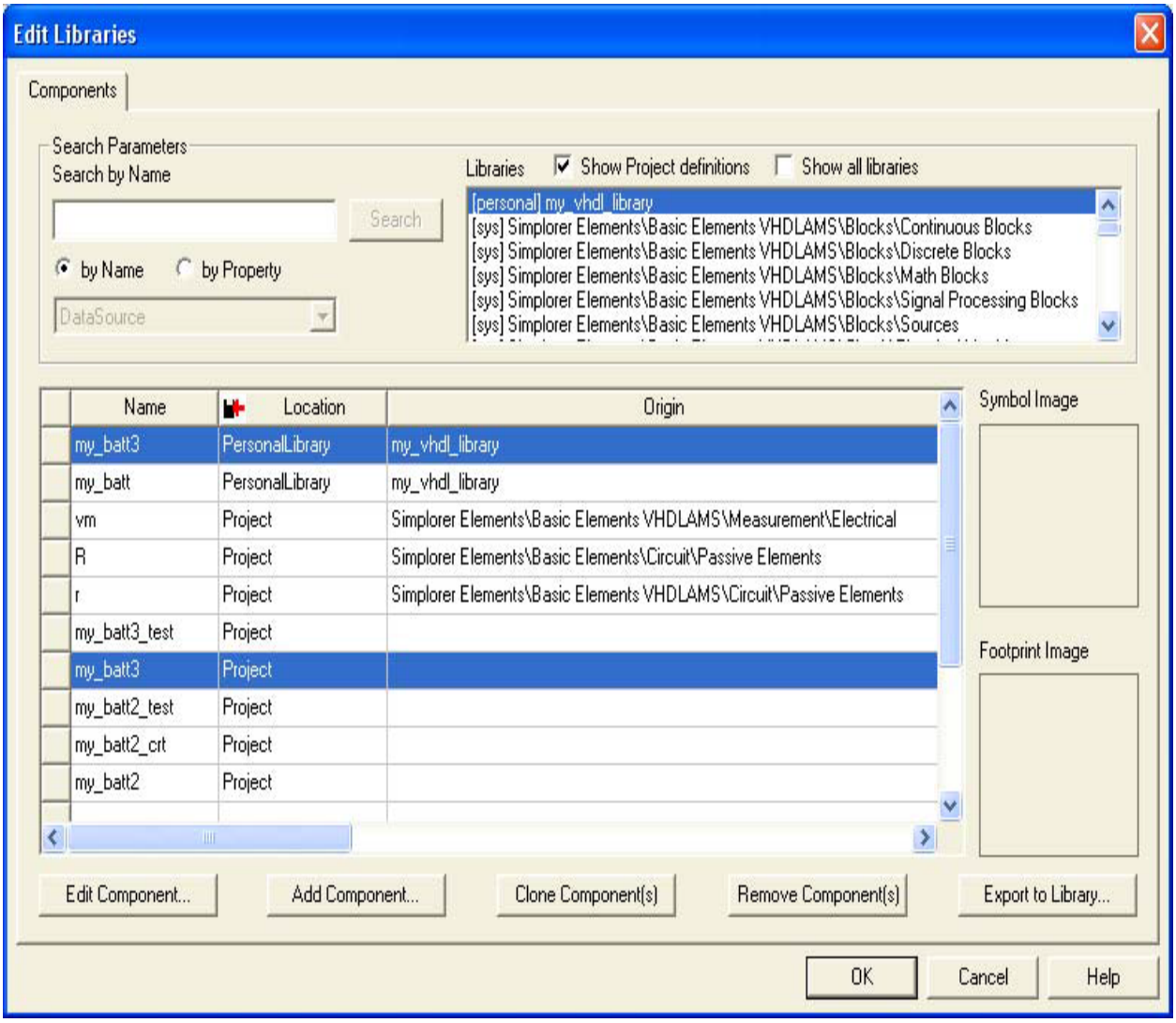
3. Select **my_batt3** from the project definitions, and then click **Export to Library** at the bottom. The following window appears. It should be open to your personal library location. Select the library where **my_batt3** will be placed, **my_vhdl_library.aclb** created in "Example #1" on page 5-7 for this example. Click **Save**.



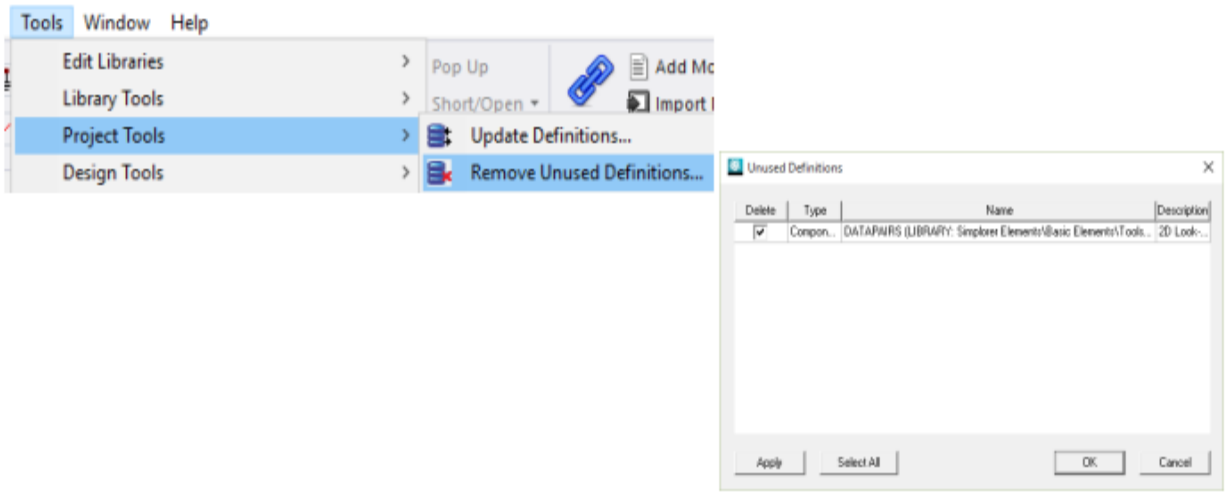
4. The following window appears showing that this component has three library type definitions associated with it (Components, Symbols, and Models) make sure all three are selected, then click **OK**.



5. The **Edit Libraries Components** dialog box should now list the **my_batt3** component you just exported in the **[personal] my_vhdl_library** library as shown below. Note that the other Project definitions for **my_batt3** still exist. Click **OK** to close.

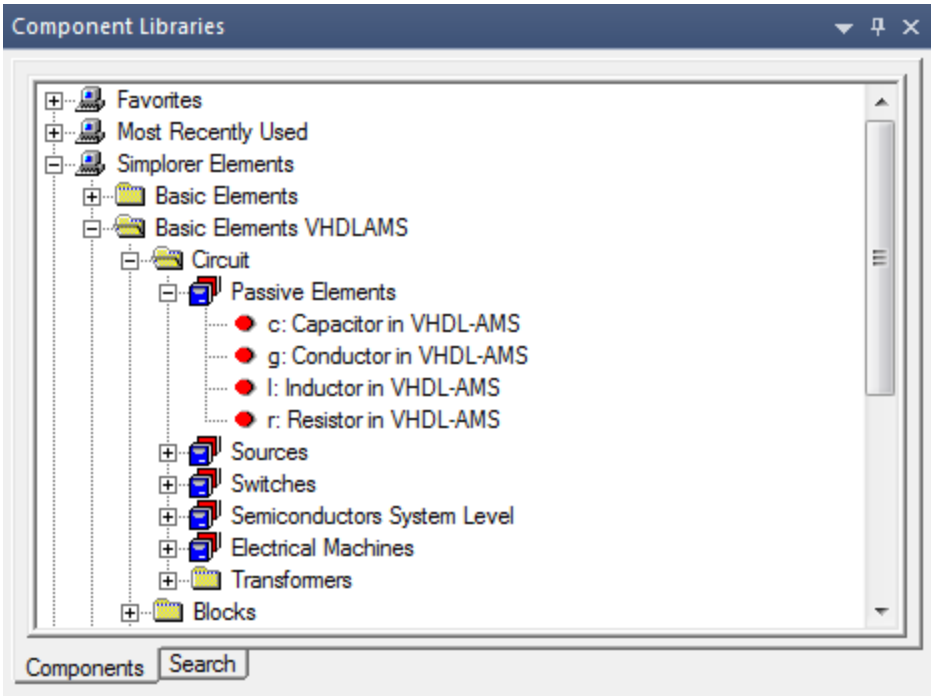


6. It is a good idea at this point to clean up the Project definitions. Select **Tools** > **Project Tools** > **Remove Unused Definitions**. Use the **Select All** and **Apply** buttons to remove all unused definitions; repeat until all are removed.

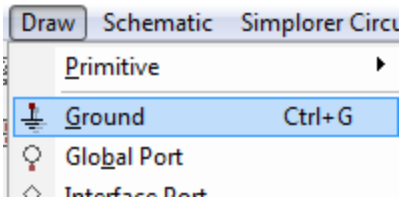


Create Twin Builder Design using new “my_batt3” component

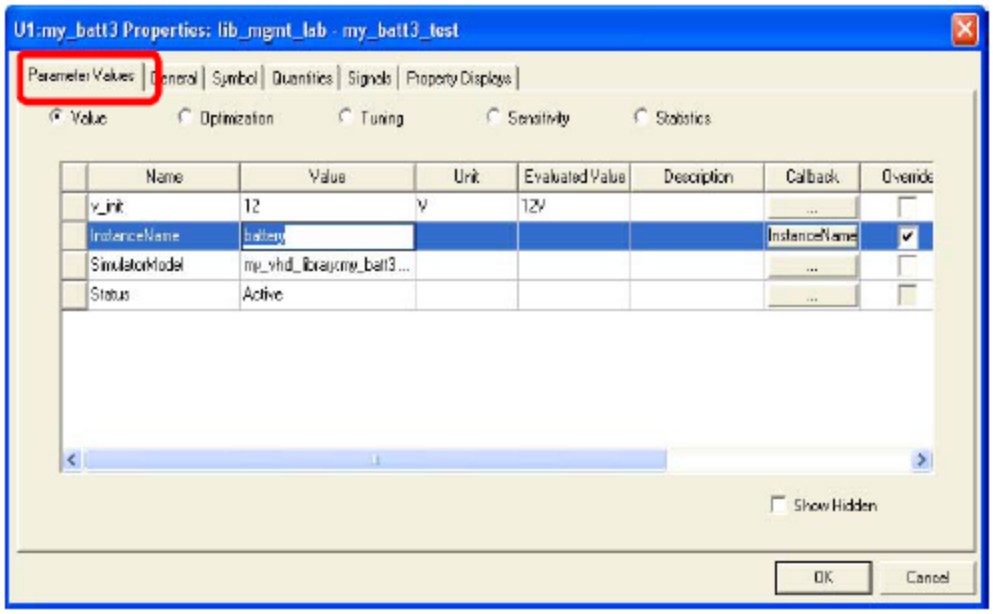
1. In the **Component Libraries** window, view the contents of the **my_vhdl_library** personal library. The VHDL-AMS model created in "Example #1" on page 5-7 should appear along with the newly exported **my_batt3**.
2. Drag the **my_batt3** VHDL-AMS model into the **my_batt3_test** Twin Builder design. Place a resistor from the **Simplorer Elements/Basic Elements VHDLAMS/Circuit/Passive Elements** library.



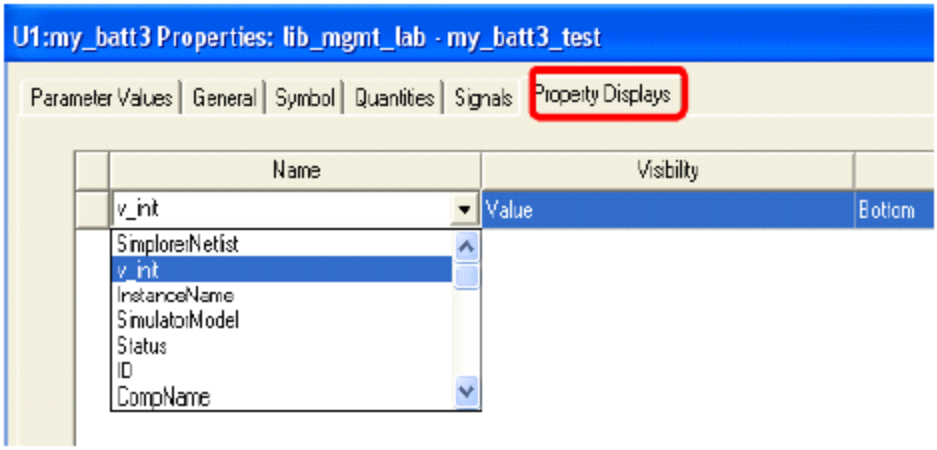
3. Add a ground node. (Ctrl-G also adds a ground.)



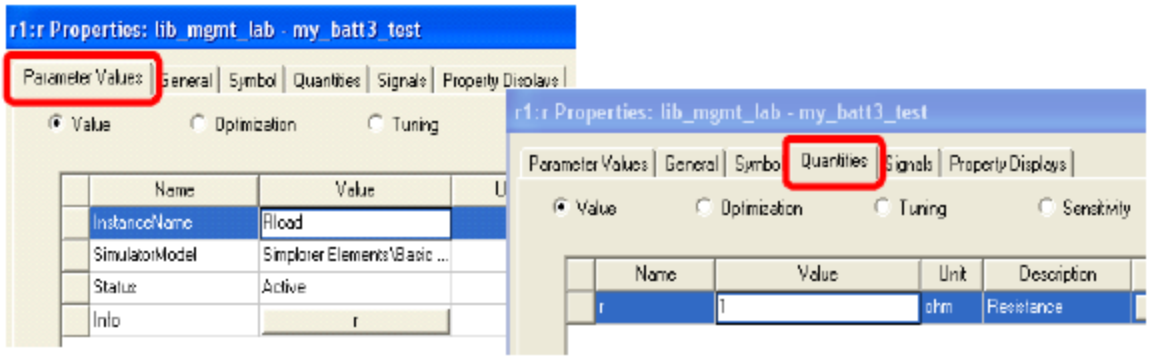
4. Connect the resistance in parallel to the battery model, and connect the ground to the negative battery terminal. Double-click **my_batt3** and select the **Parameter Values** tab to edit the model parameters. Leave the value for **v_init** at its default value that was defined in the VHDL-AMS code. Change the **InstanceName** to **battery**. Note the **factor** term is found under the **Quantities** tab, however leave it at its default value also.



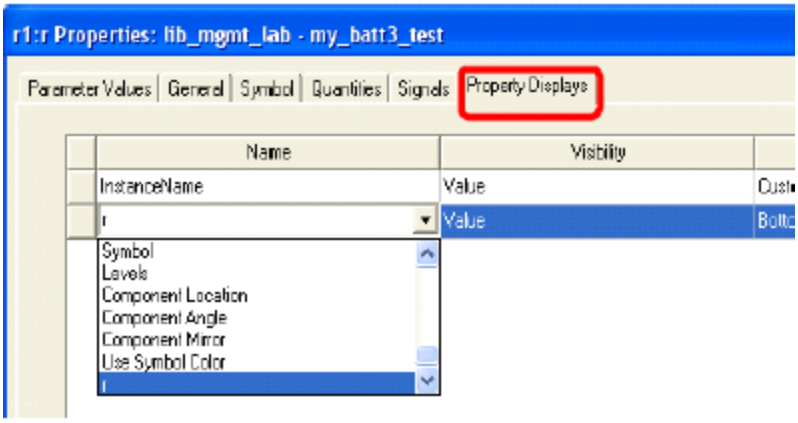
5. Select the **Properties Display** tab, then click **Add**. Add a display to show the value of **v_ init** which represents the initial charge voltage on the battery model. Click **OK**.



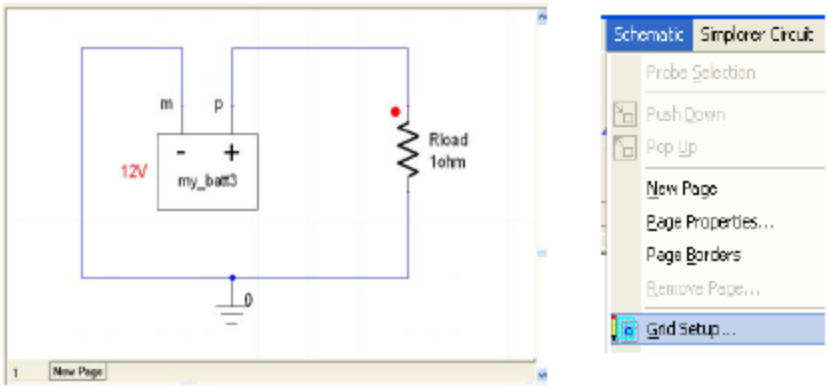
6. Double-click the resistor and change its name to **Rload** and its value to **1** ohm.



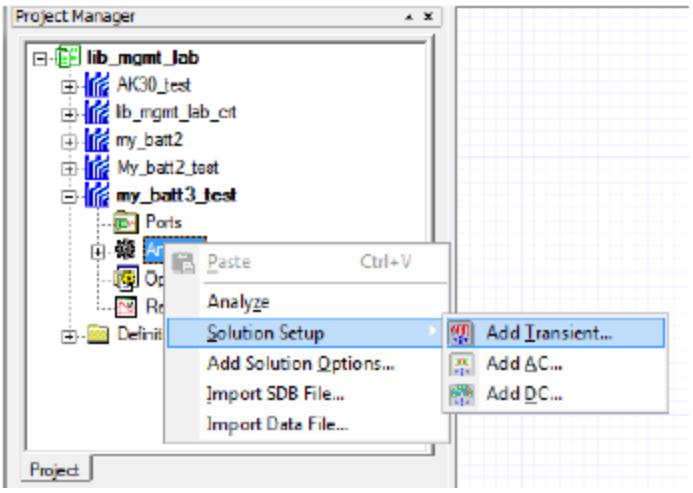
7. Select the **Property Display** tab on the resistor's **Properties** dialog box. Click **Add**, then use the pull-down menu to add the resistance value "**r**".



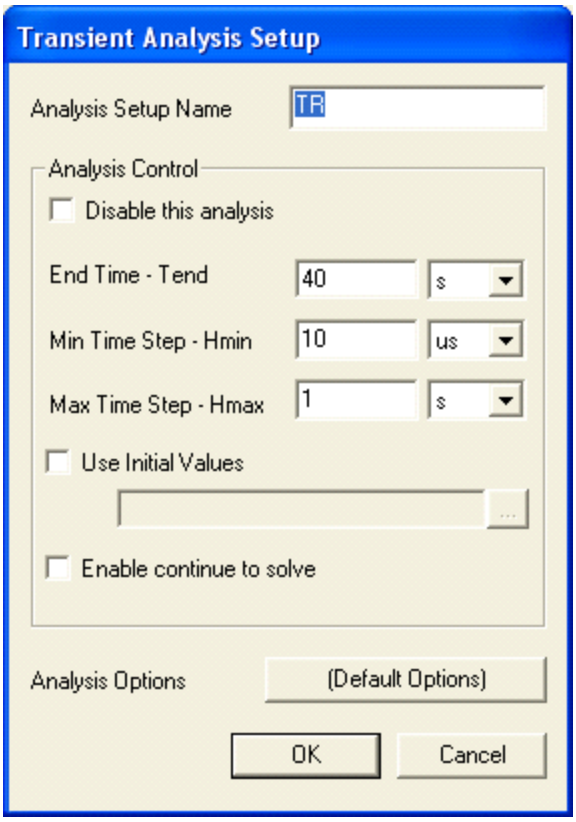
8. The circuit should appear as shown below. You can improve the text placement capability by deselecting **Snap Text and Graphics to Grid** in the **Grid Setup** dialog box. Save the design.



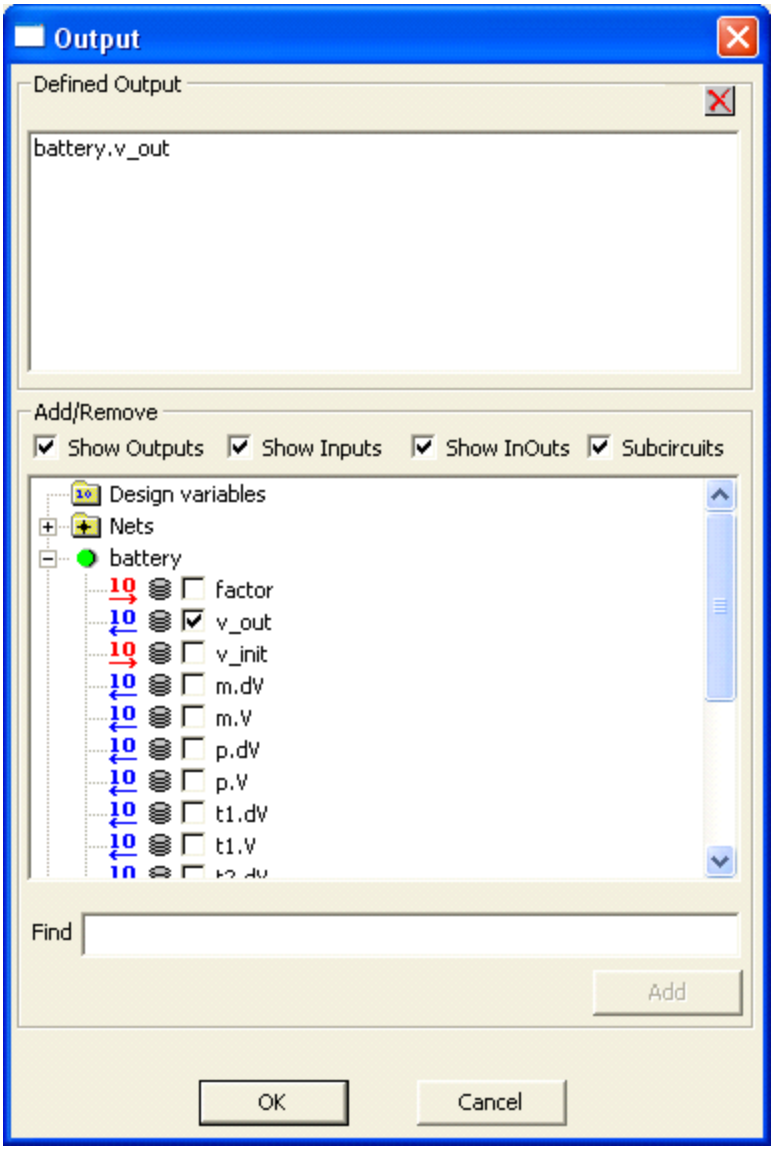
9. Add a Transient analysis set up by moving the cursor over the **Analysis** icon in the **Project Manager** pane, and then right-clicking and choosing **Solution Setup** > **Add Transient**.



10. Change the **End Time - Tend** to **40 s**, and the **Max Time Step - Hmax** to **1s**.



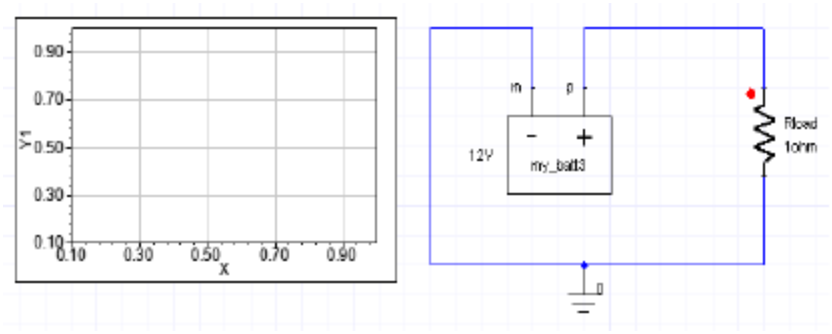
11. View/Modify the waveforms that will be available for plotting by selecting **Twin Builder Circuit** > **Output Dialog**.



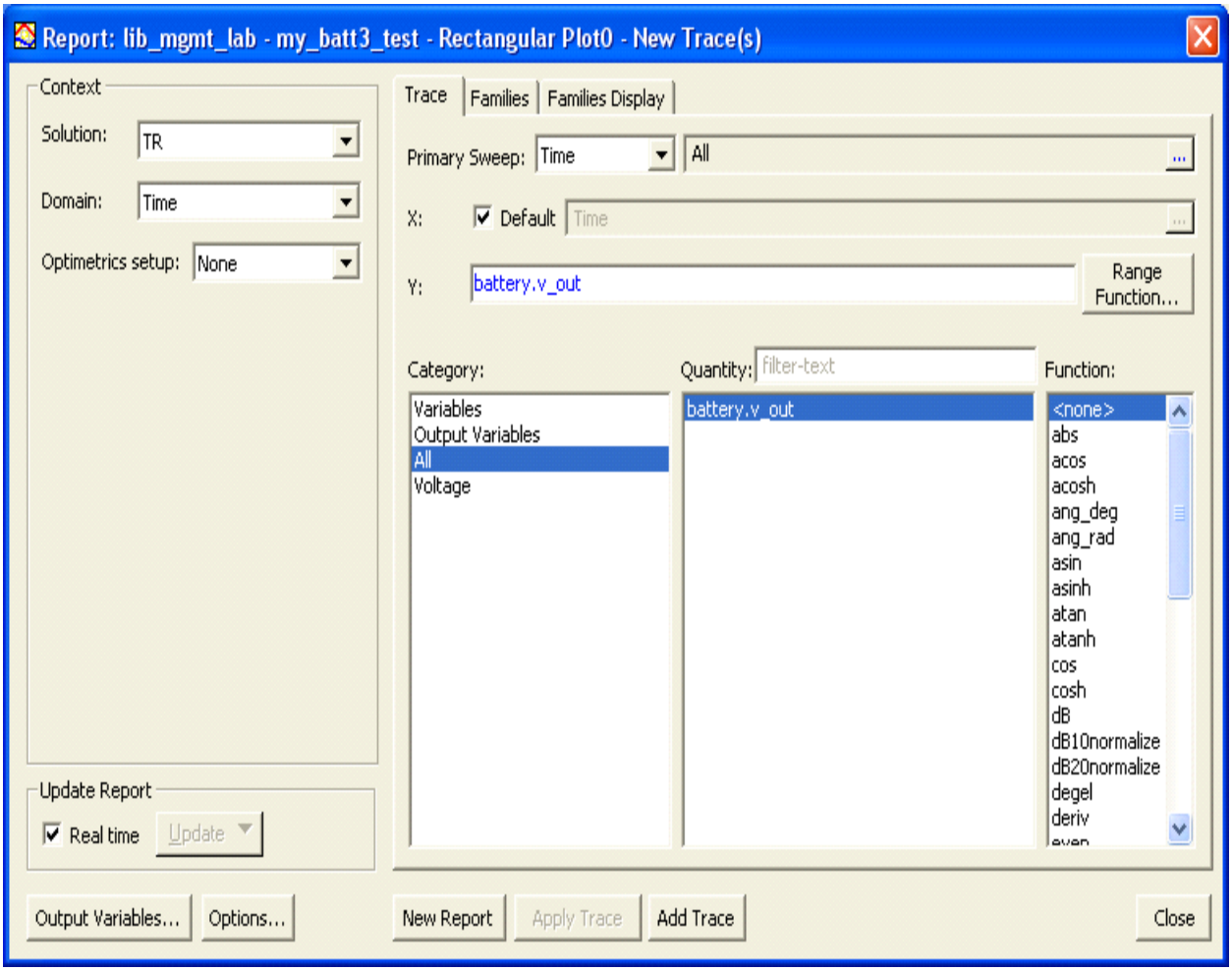
Note that **battery.v_out** had already been declared to be saved to the data base (and therefore available to plot) when the VHDL-AMS model was imported to Twin Builder. Click **OK**.

12. Zoom out on the schematic to make room for a plot, and then add a plot to the schematic via the **Draw** > **Report** > **Rectangular Plot** menu command.

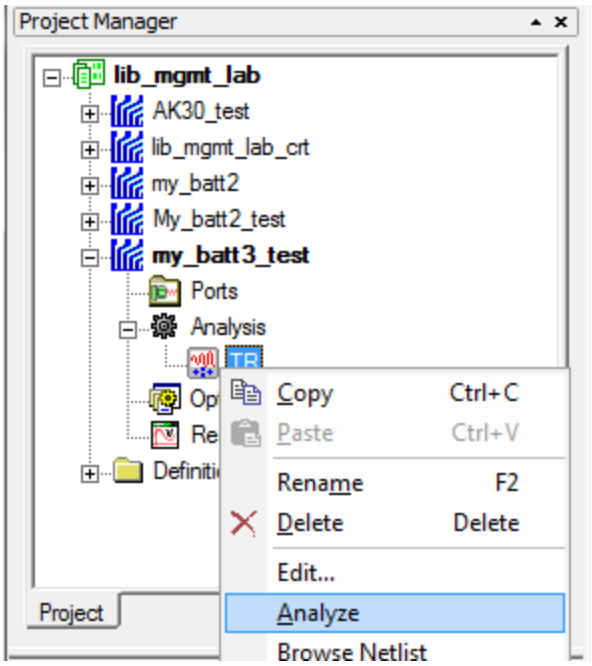
13. Draw the plot box to desired size on the schematic.



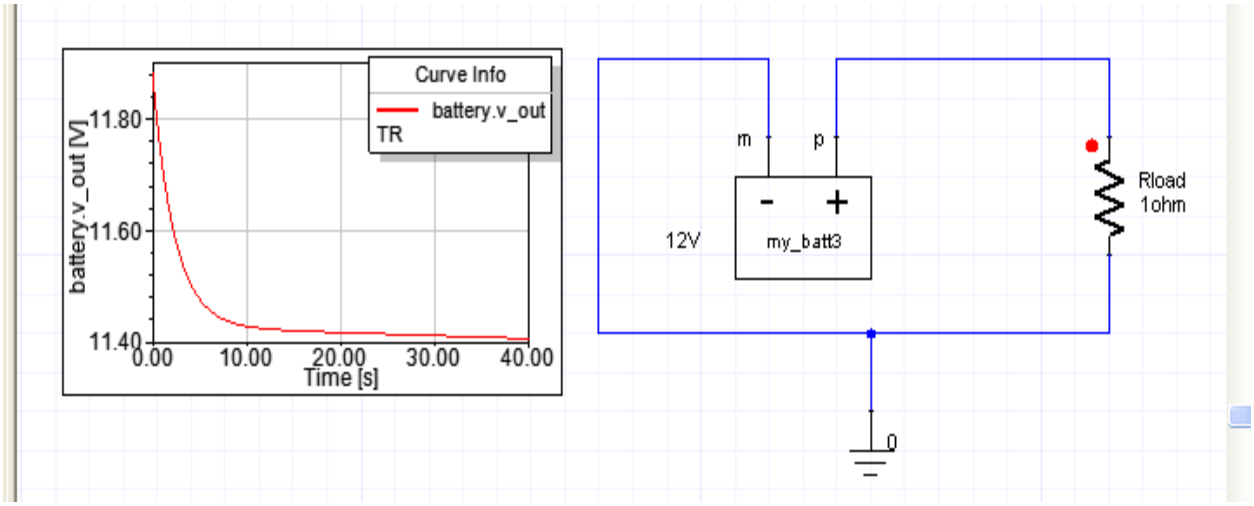
14. Double-click the plot box, select the **battery.v_out** waveform, then click **Add Trace** at the bottom. Click **Close**.



15. Analyze the design by moving the mouse over the **TR** analysis, then right-click and select **Analyze**.



16. The results should now be displayed on the schematic for the battery voltage as it discharges into the load resistance as shown below. Save the project.

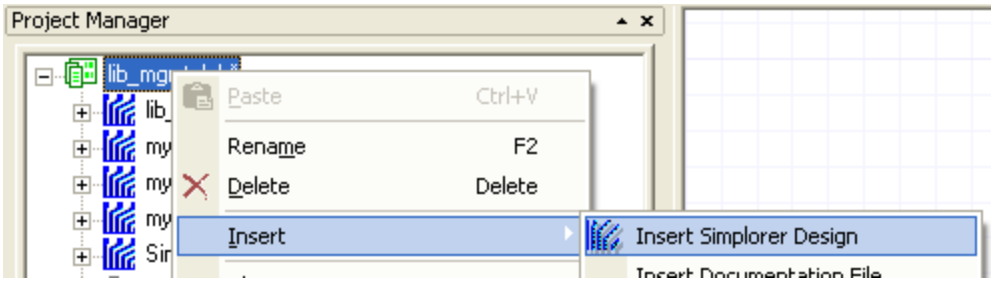


Example #3 - Use VDALibs VHDLAMS Libraries

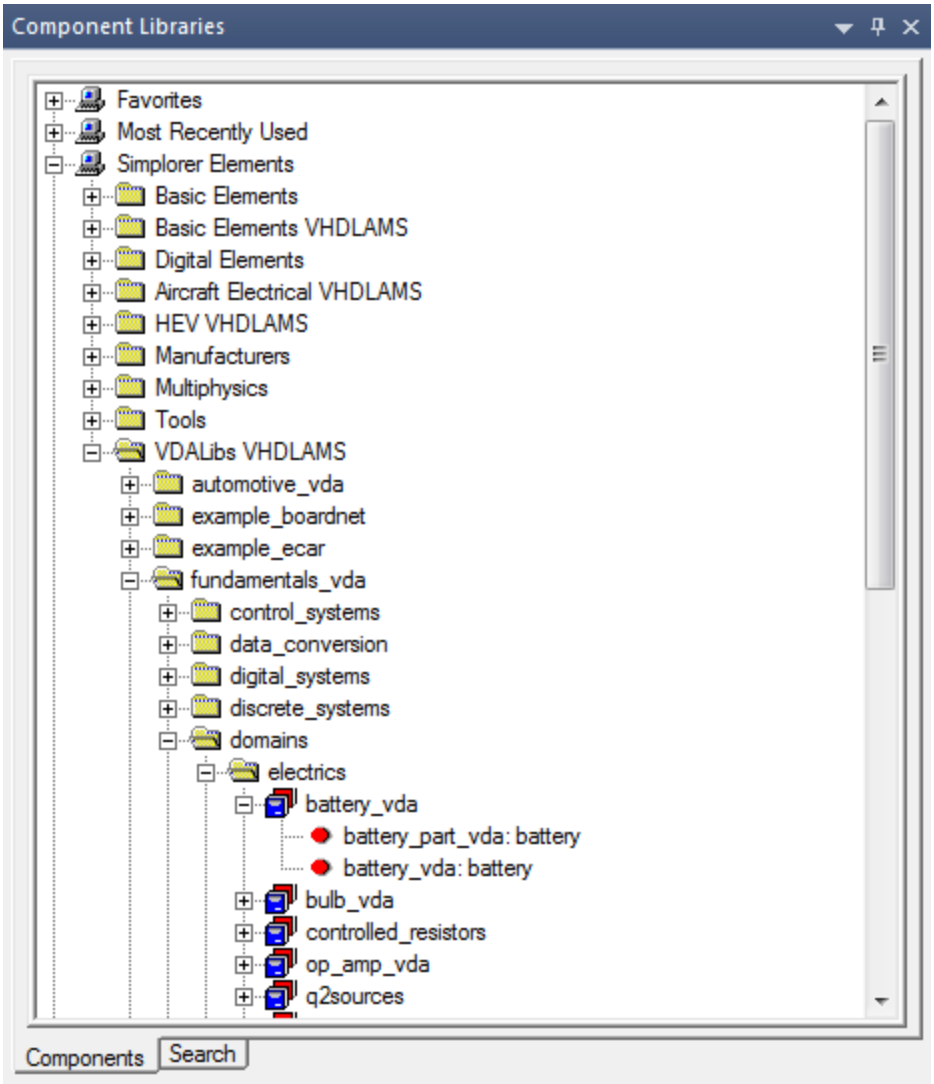
VHDL-AMS models from the *FAT-AK30* working group of the *German Association of the Automotive Industry (Verband der Automobilindustrie - VDA)* are available in Twin Builder in the **Component Libraries** window as VDALibs VHDLAMS.

The *German Association of the Automotive Industry* consists partly of automobile manufacturers and their development partners. The working group *FAT-AK30* is organized within the *German Association for Research in Automobile Technology* (FAT) of the VDA. The group's open-source VHDL-AMS model library supports the interchange of models between car manufacturers (OEMs) and their suppliers. These models support a wide range of design types.

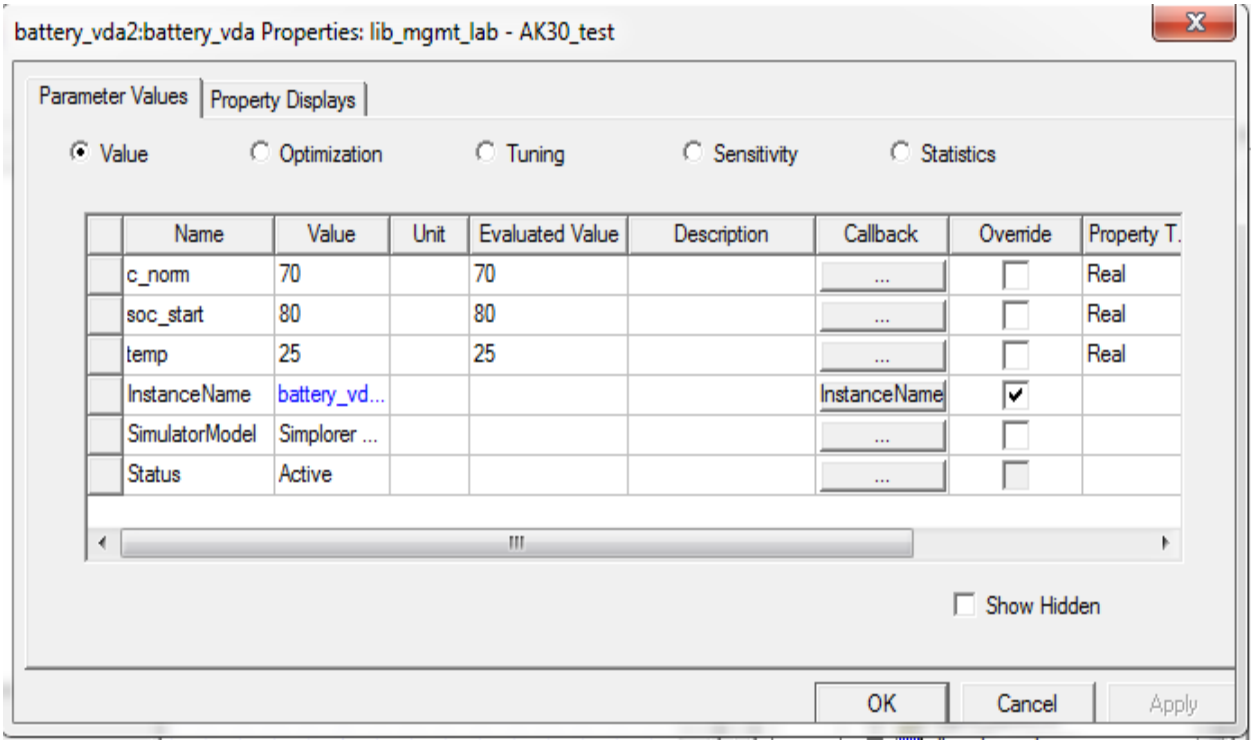
1. Select the project **lib_mgmt_lab** and add a Twin Builder Design by right-clicking on it and selecting **Insert** > **Insert Twin Builder Design**. Rename it **AK30_test**.



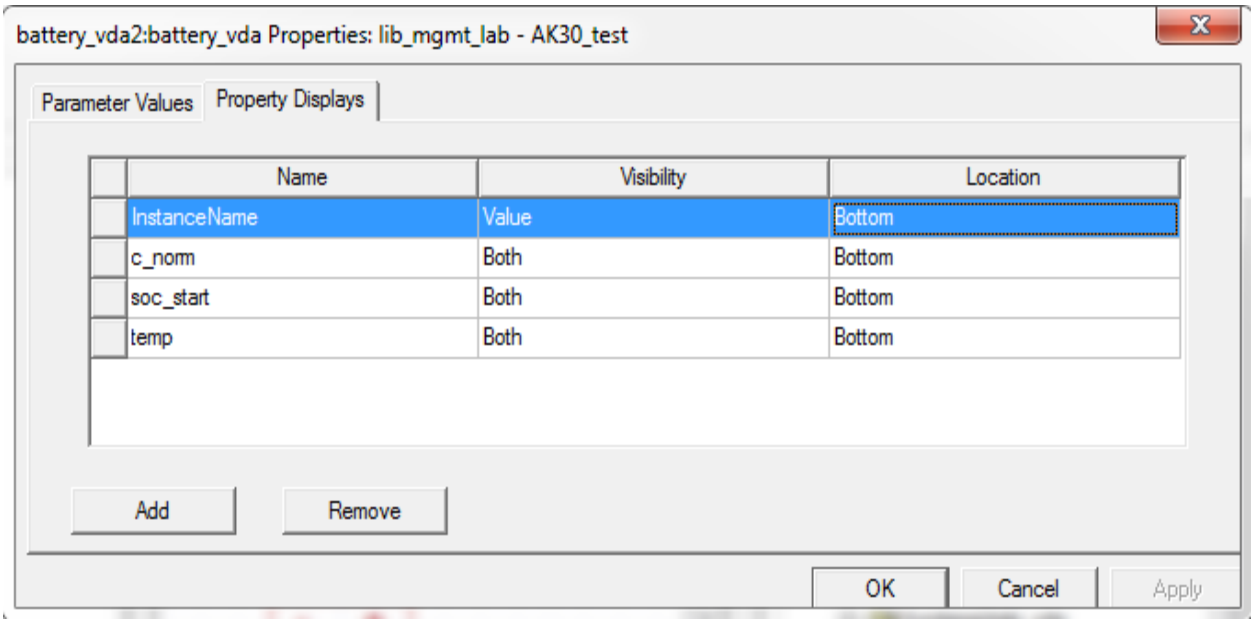
2. In the **Component Libraries** window, under **Simplorer Elements/VDALibs VHDLAMS/fundamentals_vda/domains/electrics,** select **battery_vda:battery_vda** and drag it into the **AK30_test** Twin Builder design.



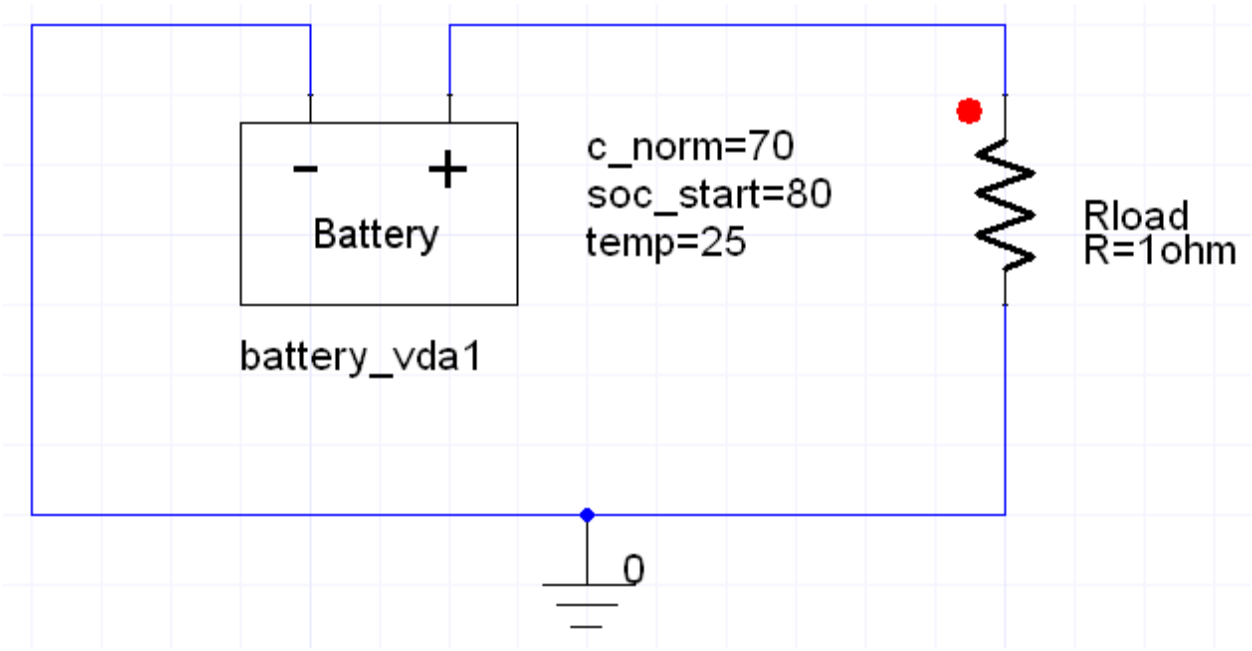
3. Create a test circuit using a one ohm resistor in parallel with the battery component as before. Double-click the battery symbol to bring up the **Properties** dialog box. Select the **Parameter Values** tab, and define the following values as shown below:
    - battery capacity: **c_norm = 70.0**
    - starting state of charge: **soc_start = 80.0**
    - temperature: **temp = 25.0**



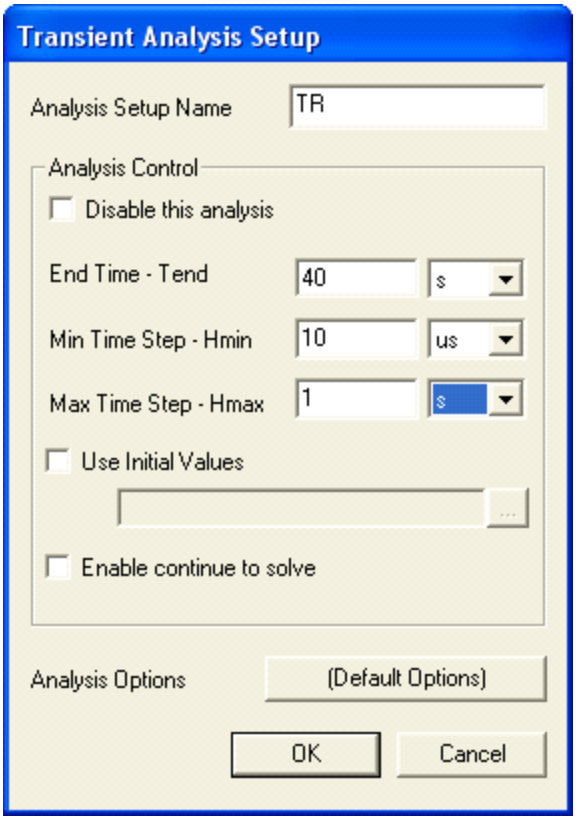
4. Select the **Property Displays** tab and click **Add**; then add three more displayed properties on the schematic for the parameters just defined. Click **OK**.



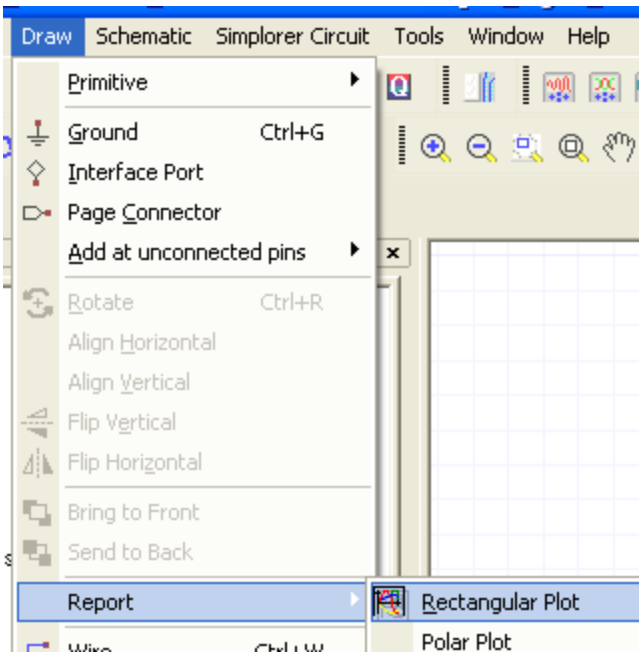
The circuit should appear as shown below.



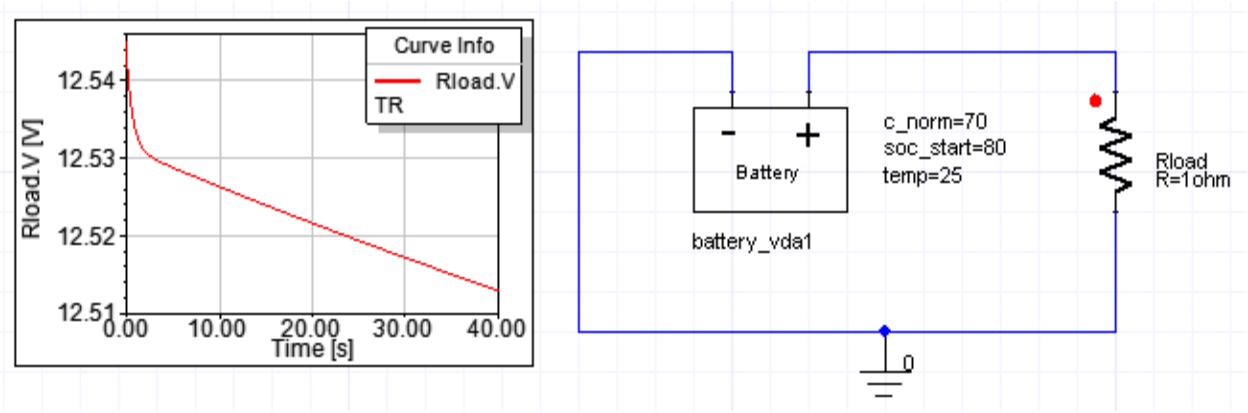
5. Define the TR analysis setup for the **AK30_test** design as shown below (**End time = 40s**, **Max Time Step = 1s**, all other settings remain at their default values).



6. Zoom out to make room on the schematic for a plot, and then add a rectangular report plot to the schematic.



7. Double-click the plot, and define the waveform to be displayed as **Rload.V**, click **Add Trace**, then click **Close**.
8. Run the analysis. The results should appear as shown below.



6 - VHDL-AMS Language Fundamentals

The VHDL-AMS language fundamentals described in this section provide a quick reference guide to find a specific statement, or the syntax for a specific statement, needed to write VHDL-AMS code.

See also information on the Web site of IEEE 1076.1 Working Group at http://www.eda.org/vhdl-ams.

The VHDL-AMS language is case insensitive; upper case letters are equivalent to lower case letters. In this document, reserved words are in **UPPER** case and shown in **BOLD**.

Identifiers are simple names starting with a letter and may have letters and digits. The underscore character is allowed but not as the first or last character of an identifier.

A comment starts with two consecutive hyphens, “--”, and continues until the end of the line.

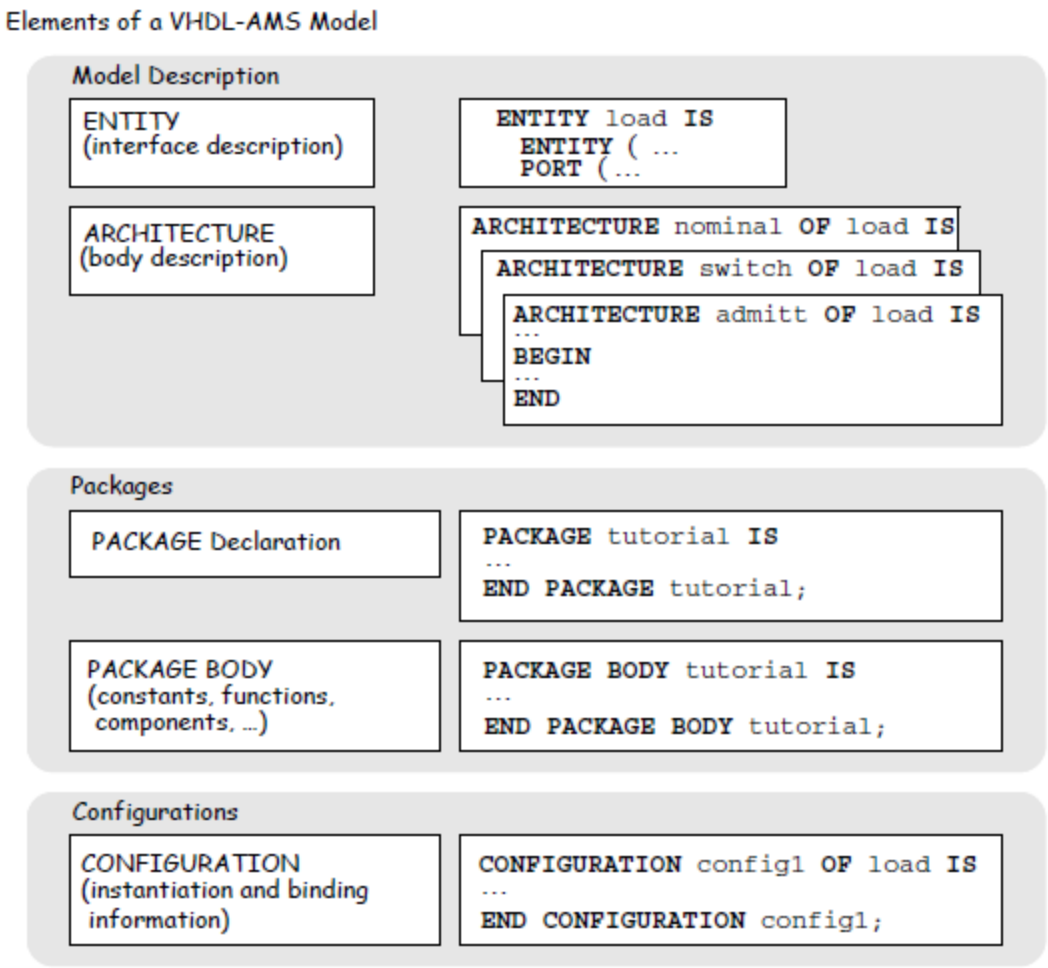
The following table shows the syntax used in the documentation.

Syntax	**Description**
ENTITY	VHDL-AMS keyword
[expression]	optional entry
[name | string]	alternative selection
identifier{,…}	repeated entries
=> <= :=	assignment operators
==	simultaneous statement

Design Units

The VHDL-AMS language allows the definition of models for analog, digital, and mixed signal circuits and systems in a standardized language. Design units (also library units) are segments of VHDL-AMS code that can be compiled separately and stored in a library.

An entity normally consists of five basic elements, or design units: entities, architectures, packages, package bodies, and configurations. Entities and architectures are the only two design units that must exist in any VHDL-AMS design description. Packages and configurations are optional.



Entities and Architectures

Each VHDL-AMS design description consists of an **ENTITY** declaration and one or more architectures. The **ENTITY** declaration defines the inputs to and outputs from the model, and any **GENERIC** parameters used by the different implementations. Each **ARCHITECTURE** defines a different implementation or behavior of a given design unit.

Entity Declaration

An entity declaration defines input to a model and outputs supported by the model, and generic parameters used by the different implementations of the model.
ENTITY entity_name **IS** **GENERIC** (generic_list); -- optional generic list

PORT (port_list); -- input/output signal ports **END** [**ENTITY**] name;	
generic_list	Specifies static information to be communicated to a model from its environment for all architectures. These include timing information (setup, hold, delay times), ambient information (temperature), and other parameters.
port_list	Specifies dynamic information to be communicated between a model and its environment for all architectures. A port can be represented by a quantity, terminal, or signal. The mode of a port defines the directions of the signals on that port, and is one of **IN**, **OUT**, **INOUT**, **BUFFER**, or **LINKAGE**.
Port Modes	An **IN** port can be read but not updated within the module. An **IN** port cannot appear on the left side of a signal assignment. An **OUT** port can be updated but not read within the module. An **OUT** port cannot appear on the right side of a signal assignment. An **INOUT** port is bidirectional and can be both read and updated, with multiple update sources possible. Signal objects can use the mode **IN**, **OUT**, and **INOUT**; quantity objects can use **IN** and **OUT** whereas terminals have no direction mode. Ports of type **BUFFER** and **LINKAGE** are transformed to **INOUT** types in Twin Builder.

ENTITY spring_tr **IS**

GENERIC(s0: DISPLACEMENT:= 0.0); -- list of generic parameters

PORT(**QUANTITY** c: **IN** STIFFNESS:= 100.0; -- list of quantity ports

TERMINAL tr1, tr2 : TRANSLATIONAL_V; -- list of terminal ports

SIGNAL ctrl: **IN** BIT); -- list of signal ports

END ENTITY spring_tr;

Architecture Declaration

An architecture defines one particular implementation of a design unit (model behavior), at some desired level of abstraction.

ARCHITECTURE architecture_name **OF** entity_name **IS**

…declarations

BEGIN

…concurrent/sequential/simultaneous statements **END** [**ARCHITECTURE**][architecture_name]	
declarations	Information used in the model description. Declarations include data types, constants, signals, files, components, attributes, subprograms, and others.
concurrent statements	Digital statements that are executed asynchronously with respect to each other. They describe a design unit at one or more levels of modeling abstraction, including dataflow, structural, and/or behavior.
sequential statements	Statements that are executed in the order in which they appear. They define algorithms for the execution of a subprogram or process.
simultaneous statements	Statements that are executed at the same time with respect to each other. They describe analog Differential Algebraic Equations (DAE).

```
ARCHITECTURE behav of spring_tr IS

QUANTITY v ACROSS f THROUGH tr1 TO tr2; -- branch quantity declaration

QUANTITY s: DISPLACEMENT; -- parameter declaration

BEGIN

 BREAK s => s0; -- value assignment

 f == c*s; -- model equation

 v == s'DOT; -- model equation

END ARCHITECTURE behav;
```

Packages

A VHDL-AMS package contains subprograms, constant definitions, and/or type definitions that may be used in one or more design units. Each package comprises a package declaration part and a package body. The declaration part represents the portion of the package that is visible outside of that package.

Package Declaration

Package declarations define the available types, constants, natures, subprograms, and attributes.

```
PACKAGE package_name IS

 …constant/type/subprogram/nature/attribute declarations

END package_name;
```

declarations	Information used in the model description. Declarations include data types, constants, signals, files, components, attributes, subprograms, and others.

```
PACKAGE omnicaster_package IS
 -- subprogram declarations
 FUNCTION b2bl (b: BIT) RETURN BOOLEAN;
 FUNCTION b2i (b: BIT; zv : INTEGER; ov : INTEGER) RETURN INTEGER;
END omnicaster_package;

PACKAGE misc IS
 TYPE short_integer IS range -100 TO 100; -- type declaration
 CONSTANT K : REAL := 1.3806503e-23; -- constant declaration
 --subtype declarations
 SUBTYPE TEMPERATURE IS REAL TOLERANCE "DEFAULT_TEMPERATURE";
 SUBTYPE HEAT_FLOW IS REAL TOLERANCE "DEFAULT_HEAT_FLOW";
 -- nature declaration
 NATURE THERMAL IS TEMPERATURE ACROSS HEAT_FLOW THROUGH THERM_REF
REFERENCE;
END misc;
```

Package Body

The package body defines the subprograms along with any internally used constants and types.

```
PACKAGE BODY package_name IS
 …subprogram bodies
END package_name;
```

subprogram_bodies	Specifies the subprograms declared in the package declaration.

```
PACKAGE BODY omnicaster_package IS
 FUNCTION b2bl (b : BIT) RETURN BOOLEAN IS
 BEGIN
```

```
IF (b = '1') THEN
RETURN TRUE;
ELSE
RETURN FALSE;
END IF;
END b2bl;
FUNCTION b2i (b : BIT; zv : INTEGER; ov : INTEGER) RETURN INTEGER IS
BEGIN
IF (b = '1') THEN
RETURN ov;
ELSE
RETURN zv;
END IF;
END b2i;
END omnicaster_package;
```

Package Visibility

The **LIBRARY** statement is used to make specified libraries such as the **IEEE** library visible in a model description. If the library file name is not consistent with VHDL-AMS naming conventions, you must define an alias. See "Alias for File Names" on page 6-61 .

A **USE** statement can precede the declaration of any entity or architecture which is to utilize items from the package. If the **USE** statement precedes the **ENTITY** declaration, the package is also visible to the architecture.

To make all items of a package visible to a design unit, precede the desired design unit with a **USE** statement accompanied by the **ALL** keyword. To make single items of a package visible, only the corresponding item is defined in the **USE** statement. This saves simulation (compilation) time, since all visible items in a **USE** statement must be loaded before simulation.

See "WORK Library" on page 6-60 .

```
LIBRARY library_name;
USE library_name.package_name.ALL; -- all items are visible
```

USE library_name.package_name.item_name; -- one item is visible	
library_name	Library name, for example **transformations**. You must omit the extension .**smd** appended to Twin Builder library files.
package_name	Name of the package in a library, for example **omnicaster_ package**.
item_name	Name of a type, subprogram, or constant in the package.

```
LIBRARY TRANSFORMATIONS; -- make library transformations.smd visible
LIBRARY VHDLAMS_TUTORIAL; -- make library vhdlams_turorial.smd visible

USE TRANSFORMATIONS.omnicaster_package.ALL; -- all information available
USE VHDLAMS_TUTORIAL.octal_conversions.int2oct; -- function int2oct available
```

VHDL-AMS Standard Packages and Types

Several standard packages are installed along with Twin Builder in the STD and IEEE libraries. These libraries can be found in the **Basic_VHDLAMS** folder in the **Component Libraries** window. The VHDL-AMS source code for most of the packages can be viewed in the VHDL Model Editor by opening the **Properties** dialog box for each package. The source code for the MATH_REAL, NUMERIC_STD and NUMERIC_BIT packages are protected by an IEEE copyright and consequently cannot be viewed in the VHDL Model Editor.

To use any of the packages, the **LIBRARY** statement and the **USE** statement need to be specified as follows:

```
LIBRARY library_name;
USE library_name.package_name.ALL;
```

STD Library

The STD library has two packages: STANDARD and TEXTIO.

- The STANDARD package provides a number of types, subtypes, and functions. This package is different from all other packages in that it is always visible and need not be explicitly included within a model.
- The TEXTIO package provides data types and subprograms that are required for reading and writing ASCII files.

The TEXTIO package can be made visible within a model description in the following manner:

```
LIBRARY STD;
USE STD.TEXTIO.ALL;
```

The IEEE Library

The IEEE library houses a number of packages that can be classified into three categories:

- **Digital** – Packages for the simulation of digital designs.
- **Physical Domains** – Packages for the simulation of multidomain systems.
- **Math** – Packages for mathematical operations.

Packages for the Simulation of Digital Designs

The following table lists the packages that are available for use in digital designs:

Package Name	Functionality	Usage
STD_LOGIC_1164	Defines the multi-value logic data type (STD_LOGIC) and associated operations.	**USE** IEEE.STD_LOGIC_1164.**ALL**;
STD_LOGIC_ARITH	Synopsys package based on multi-value logic that defines types and basic arithmetic operations for representing integers.	**USE** IEEE.STD_LOGIC_ARITH.**ALL**;
STD_LOGIC_SIGNED	Synopsys extension of the STD_LOGIC_ARITH package to handle multi-value logic values as signed integers.	**USE** IEEE.STD_LOGIC_SIGNED.**ALL**;
STD_LOGIC_UNSIGNED	Synopsys extension of the STD_LOGIC_ARITH library to handle multi-value logic values as unsigned integers.	**USE** IEEE.STD_LOGIC_UNSIGNED.**ALL**;
NUMERIC_STD	IEEE package based on multi-value logic that defines types and basic arithmetic operations for representing integers. This is similar to STD_LOGIC_ARITH and consequently should not be used together.	**USE** IEEE.NUMERIC_STD.**ALL**;
NUMERIC_BIT	IEEE package based on binary (BIT) data type that defines types and basic arithmetic operations for representing integers.	**USE** IEEE.NUMERIC_BIT.**ALL**;

Note:

The definitions for signed and unsigned data types are available in the NUMERIC_STD package, and in the Synopsys SIGNED and UNSIGNED packages. Consequently, the Synopsys packages cannot be used with the NUMERIC_STD package in any VHDL-AMS design.

Packages for the Simulation of Multidomain Systems

The following table lists the physical domain packages that are available for use in multidomain simulations:

Domain	**Usage**
Electrical	**USE** IEEE.ELECTRICAL_SYSTEMS.**ALL**;
Mechanical • Translational • Rotational	**USE** IEEE.MECHANICAL_SYSTEMS.**ALL**;
Thermal	**USE** IEEE.THERMAL_SYSTEMS.**ALL**;
Fluidic	**USE** IEEE.FLUIDIC_SYSTEMS.**ALL**;
Radiant	**USE** IEEE.RADIANT_SYSTEMS.**ALL**;

The following table lists the common declarations and constants that are valid in all physical domains:

Package Name	**Usage**
Energy Systems	**USE** IEEE.ENERGY_SYSTEMS.**ALL**;
Fundamental Constants	**USE** IEEE.FUNDAMENTAL_CONSTANTS.**ALL**;
Material Constants	**USE** IEEE.MATERIAL_CONSTANTS.**ALL**;

Packages for Mathematical Operations

Package Name	**Usage**
Math_Real (1076.2)	**USE** IEEE.MATH_REAL.**ALL**;

Subprograms

Subprograms define algorithms for calculating particular functions of a model. They can be used to divide complex model descriptions into smaller sections.

There are two forms of subprograms: procedures and functions. A procedure call is a statement; a function call is an expression and returns a value. A subprogram has two parts:

- Declaration statements
- Sequential statements defining the behavior

Subprograms can use constants, variables, and signals as parameters. The parameters that are used within a subprogram are called the formal parameters, while the parameters passed into the function or procedure are called the actual parameters.

Note that a simultaneous procedural statement is different from a subprogram. A procedural statement is used to describe analog behavior that occurs sequentially (rather than be solved simultaneously).

Procedures

<table>
<tr><td colspan="2">A procedure defines a group of sequential statements that are executed when the procedure is called. A procedure can return any number of values (or no values) via its parameter list. A procedure call invokes the execution of the procedure body.</td></tr>
<tr><td colspan="2">PROCEDURE procedure_name [(formal_parameters)] IS
…declarations
BEGIN
…sequential statements
END [PROCEDURE] [procedure_name];
…
procedure_name[actual_parameters] -- procedure call</td></tr>
<tr><td>formal_parameters</td><td>Specifies a list of parameters (constants, signals, or variables with the mode in, out, or inout).</td></tr>
<tr><td>declarations</td><td>Declarations include data types, constants, signals, files, variables, attributes, and subprograms.</td></tr>
<tr><td>sequential statements</td><td>Statements that are executed in the order in which they appear, that define algorithms for the execution of a subprogram or process.</td></tr>
<tr><td colspan="2">-- declaration with formal parameters int and bin</td></tr>
</table>

```
PROCEDURE int2bin (VARIABLE int: IN INTEGER; VARIABLE bin: OUT BIT_VECTOR) IS
 VARIABLE temp: INTEGER;
BEGIN -- start of sequential procedure statements
 temp := int;
 FOR i IN 0 TO (bin'LENGTH -1) LOOP
 IF (temp mod 2 = 1) THEN
 bin(i) := '1';
 ELSE
 bin(i) := '0';
 END IF;
 temp := temp/2;
 END LOOP;
END int2bin; -- end of sequential procedure statements
…
PROCESS -- continued model description
 VARIABLE in_var: INTEGER:=10;
 VARIABLE out_vec: BIT_VECTOR (1 to 8);
BEGIN
 -- procedure call with actual parameters int_var and out_vec
 int2bin (in_var, out_vec);
 WAIT;
END PROCESS;
```

Functions

A function defines a group of sequential statements that are executed when the function is called. A function returns a single value. Unlike procedures, functions are primarily used in expressions and only have inputs in their argument list. A function call invokes the execution of the function body.

[**PURE** | **IMPURE**] function function_name **RETURN** type_name

…declarations

BEGIN

…sequential statements

END [**FUNCTION**] [function_name];

…

function_name[actual_parameter] -- function call

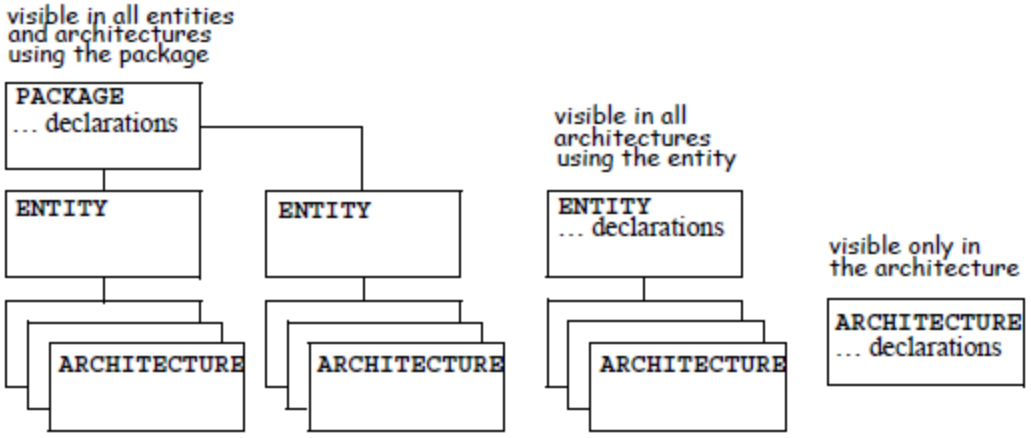
PURE | **IMPURE**	Pure functions return the same value each time they are called with the same values as actual parameters (default). Impure functions can return a different value each time they are called, even when multiple calls have the same actual parameter values.
formal_parameters	Specifies a list of parameters (constants, signals, or variables with the mode in, out, or inout).
type_name	Data type or data subtype name, for example BIT, INTEGER, REAL, … of the return value.
declarations	Declarations include data types, constants, signals, files, components, attributes, subprograms, and other information used in the model description.
sequential statements	Statements that are executed in the order in which they appear, that define algorithms for the execution of a subprogram or process.

```
-- declaration with formal parameters i and length and data type of return value
FUNCTION i2bv (i: INTEGER; length: INTEGER) RETURN BIT_VECTOR IS
 VARIABLE bv: BIT_VECTOR(length-1 DOWNTO 0);
 VARIABLE temp: INTEGER;
BEGIN -- start of sequential procedure statements
 temp := i;
 FOR j IN 0 TO (bv'LENGTH-1) LOOP
 IF (temp mod 2 = 1) THEN
 bv(j) := '1';
 ELSE
 bv(j) := '0';
 END IF;
```

```
 temp := temp/2;
 END LOOP;
 return bv;
END i2bv; -- end of sequential procedure statements
…
ARCHITECTURE bench OF entity_name IS
 SIGNAL in_var: INTEGER;
 SIGNAL out_vec: BIT_VECTOR (1 to 8);
 …
BEGIN
 in_var <= 10;
 -- function call with actual parameters in_var and value 8
 out_vec <= i2bv (in_var,8) AFTER 1ms;
END ARCHITECTURE bench;
```

Declarations

Declarations specify types, data objects, attributes, and components that can be used in design units. Declarations can be located in packages, entities, and architectures. The scope or visibility of a declaration depends on where they are used.



VHDL-AMS data types can be classified as scalar, composite, access and file types. VHDL-AMS supports different kinds of data objects for each data type. These data objects include constants, signals, variables, files, quantities, and terminals. VHDL-AMS includes a number of predefined data types, and allows user-defined data types as needed. **TYPE** statements declare a new data type. **SUBTYPE** statements constrain an existing type.

TYPE Declarations

A type declaration defines a new data type.

- **Scalar** – Declares a type used to create enumeration, integer, physical, and floating point elements.
- **Composite** –Declares a type for creating array or record elements.
- **File** – Declares a type for creating file handles.

TYPE type_name **IS** scalar_type_definition; -- scalar type definition

TYPE type_name **IS** composite_type_definition; -- composite type definition

TYPE type_name **IS FILE OF** subtype_name; -- file type declaration

type_name	Data type or data subtype name, for example **BIT**, **INTEGER**, **REAL**, …

TYPE scalar_int1 **IS RANGE** -5 **TO** 5;

TYPE scalar_int2 **IS RANGE** 31 **DOWNTO** 0;

TYPE scalar_bit **IS** ('0', '1');

TYPE scalar_enum **IS** (red, green, blue);

TYPE scalar_boolean **IS** (TRUE, FALSE);

TYPE composite1 **IS ARRAY** (0 **TO** 31) **OF** BIT;

TYPE composite2 **IS ARRAY** (natural **RANGE** <>) **OF** INTEGER;

TYPE composite3 **IS RECORD**

RE: REAL; IM: REAL;

END RECORD composite3;

TYPE composite4 **IS ARRAY** (INTEGER **RANGE** <>, INTEGER **RANGE** <>) **OF** REAL;

TYPE file_txt **IS FILE OF** INTEGER;

SUBTYPE Declaration

A subtype declaration defines a type derived from an existing type. **TYPE** creates a new type while **SUBTYPE** creates a type that is a constraint of an existing type. It is possible to assign values of objects belonging to subtypes to objects belonging to the base type.	
SUBTYPE name **IS** type_name [constraint] [tolerance];	
constraint	Specifies a range constraint for a scalar type with a **… RANGE … TO**(**DOWNTO**) **…** statement.
SUBTYPE electrical_units **IS** STRING (1 **TO** 20); **VARIABLE** var_names: electrical_units := "voltage, current"; **SUBTYPE** small_int **IS** INTEGER **RANGE** 0 **TO** 10; -- subtype with constraint definition **VARIABLE** little : small_int := 4;	

NATURE Declaration

A nature declaration defines a nature and its *across* and *through* quantities you can access through the model terminals. See "Across and Through Quantities of Natures" on page 3-2 .	
NATURE nature_name **IS** across_type_name **ACROSS** through_type_name **THROUGH** reference_terminal reference;	
nature_name	Name of a nature, for example ELECTRICAL, MECHANICAL, THERMAL, …
across_type_name through_type_name	Specifies the branch types of the nature, which define the type of branch quantities. Only terminals can represent a nature.
reference_terminal	Specifies the ground (zero) of the across type of a nature.
NATURE ELECTRICAL **IS** -- nature declaration of an electrical system voltage **ACROSS** current **THROUGH** electrical_ref **REFERENCE**; **NATURE** THERMAL **IS** -- nature declaration of a thermal system temperature **ACROSS** heat_flow **THROUGH** thermal_ref **REFERENCE**;	

Data Object Declarations

A data object holds a value of a specified type. Every data object belongs to one of six different classes, described in this section.

CONSTANT Declaration

A constant declaration assigns a value to an identifier of a given data type. The use of constants can improve the readability of VHDL-AMS code. The value of a **CONSTANT** object cannot be changed by any executable code after declaration.	
CONSTANT constant_name: type_name [expression];	
constant_name	List of constant names separated by a comma.
type_name	Data type or data subtype name, for example, **BIT**, **INTEGER**, **REAL**, …
expression	Expression that performs an arithmetic or logical computation by applying an operator to one or more operands (constant value).
CONSTANT Pi: REAL := 3.14159; **CONSTANT** Half_Pi: REAL := Pi/2.0; **CONSTANT** cycle_time: TIME := 11 ns; **CONSTANT** N, N5: INTEGER := 5; **CONSTANT** ctrl: BIT:= '1'; -- logic 1 constant **CONSTANT** zero4: BIT_VECTOR(0 to 3) := "0000";	

SIGNAL Declaration

A signal declaration defines an identifier as a signal object. A **SIGNAL** object holds a list of values, including the previous value, current value, and a set of possible future values that appear on the signal. A signal object has “digital properties”; this means that any change in a signal causes an event which can start a process or value assignment specified in the model description.	
SIGNAL name_list: type_name [expression];	
name_list	List of signal names separated by a comma.
type_name	Data type or data subtype name, for example, **BIT**, **INTEGER**, **REAL**, …
expression	Expression that performs an arithmetic or logical computation

	by applying an operator to one or more operands (default value).

```
SIGNAL a_bit : BIT := '0';
a_bit <= b_bit XOR '1';
SIGNAL clock : BIT := '0';
clock <= NOT clock AFTER 1 ms;
SIGNAL bool_sig : BOOLEAN := FALSE;
bool_sig <= TRUE WHEN clock = '1' ELSE FALSE;
SIGNAL gate_delay : TIME := 10 ns;
```

VARIABLE Declaration

A variable declaration defines an identifier as a variable object. A variable can be of any scalar or aggregate data type and is updated immediately when an assignment statement is executed. Variables can be declared only within subprograms, procedural statements, or process statements.

VARIABLE variable_name: type_name [expression];

variable_name	List of variable names separated by a comma.
type_name	Data type or data subtype name, for example, **BIT**, **INTEGER**, **REAL**, …
expression	Expression that performs an arithmetic or logical computation by applying an operator to one or more operands (default value).

```
clock <= NOT clock AFTER 10 us;
PROCESS (clock)
 VARIABLE num_events : INTEGER := 0;
BEGIN
 num_events := num_events + 1;
END PROCESS;
```

FILE Declaration

A file declaration defines an identifier as a file object.	
FILE file_name: type_name [[**OPEN** file_open_kind] **IS** file_logical_name];	
file_name	Name of the created file.
type_name	Data type name, for example **TEXT**, …
file_open_kind	Specifies the access mode for the file object: **READ_MODE**, **WRITE_MODE**, **APPEND_MODE**. These operation modes are defined in the **STD.TEXTIO** package.
file_logical_name	Defines the file name. The name is a string in quotes and its syntax must conform to the operating system where the VHDL-AMS model will be simulated.

FILE cycle: TEXT **OPEN** READ_MODE **IS** "cyc_mph_manhattan_tab.txt";

…

WHILE NOT ENDFILE (cycle) **LOOP**

READLINE (cycle,buf);

END LOOP;

QUANTITY Declaration

A quantity declaration defines one or more identifiers as quantity objects. A quantity object is specified by its type and a default value. Quantities can be declared in both entity and architecture declaration placeholders. Quantities can be classified as PORT, FREE and BRANCH quantities depending on where they are declared. PORT quantities appear in the **ENTITY** declaration, BRANCH quantities appear in the **ARCHITECTURE** declaration and specify the across and through values for terminals of a particular nature, and FREE quantities are declared as analog values in the **ARCHITECTURE** declaration.	
QUANTITY name_list : real_type_name [expression]; **QUANTITY** [across_aspect] [through_aspect] terminal_aspect;	
name_list	List of quantities separated by a comma.
real_type_name	Real type or real subtype name, for example, **REAL**, **VOLTAGE**, **TORQUE**...
across_aspect	Identifier that serves as the across quantity for the specified terminals.
through_aspect	Identifier that serves as the through quantity for the specified terminals.

terminal_aspect	Positive and negative terminal names.
ARCHITECTURE admittance **OF** load **IS** **QUANTITY** v **ACROSS** i **THROUGH** p TO m; **QUANTITY** ctrl_ramp: REAL := 0.0; **QUANTITY** ctrl_qty: REAL := 0.0; **BEGIN** -- … **END ARCHITECTURE** admittance;	

TERMINAL Declaration

A terminal declaration declares a terminal, as well as the reference quantity and contribution quantity of the terminal.	
TERMINAL name_list: nature_name;	
name_list	List of terminals separated by a comma.
nature_name	Name of the conserved physical domain.
TERMINAL terminal_e1 : ELECTRICAL; --Local Electrical Terminal **TERMINAL** terminal_t1 : THERMAL; --Local Thermal Terminal	

Other Declarations

ATTRIBUTE Declaration

An attribute declaration defines an attribute name and its type. An attribute specification assigns a value to the attribute. See ***Predefined Attributes***.	
ATTRIBUTE attribute_name: type_name; -- declaration **ATTRIBUTE** attribute_name **OF** name: entity_class **IS** expression; -- specification	
attribute_name	Name of the attribute.
type_name	Data type name, for example BIT, INTEGER, REAL, …
name	Identifier of the existing object.
entity_class	Specifies the name of the entity class, for example **TYPE**, **SIGNAL**, **FILE**, …
expression	Expressions to perform an arithmetic or logical computation by

	applying an operator to one or more operands.

ATTRIBUTE UNIT **OF** VOLTAGE: **SUBTYPEIS** "Volt"; -- attribute declaration

ATTRIBUTE SYMBOL **OF** RESISTANCE: **SUBTYPE IS** "OHM";

COMPONENT Declaration

A component declaration defines a component's interface and is typically placed in an **ARCHITECTURE** or **PACKAGE** declaration. The component or instances of the component are related to a design entity in a library using a configuration.

COMPONENT component_name [**IS**]

[local_generic_clause]

[local_port_clause]

END COMPONENT [component_name];

component_name	Name of the component.
local_generic_clause	Specifies static information to be communicated to a model from its environment in the form: variable_name : variable_ type := value;
local_port_clause	Specifies dynamic information to be communicated to a model from its environment in the form: variable_name : port_mode variable_type;

ENTITY inv4 **IS**

GENERIC (TP_LH, TP_HL: TIME);

PORT (I_4 : **IN** BIT_VECTOR(3 **DOWNTO** 0);

Y_4 : **OUT** BIT_VECTOR(3 **DOWNTO** 0));

END ENTITY inv4;

ARCHITECTURE inv4_struct **OF** inv4 **IS**

COMPONENT inv

GENERIC (TP_LH, TP_HL: TIME);

PORT (I1 : **IN** BIT;

Y1 : **OUT** BIT);

END COMPONENT;

SIGNAL INV_OUT : BIT_VECTOR(3 **DOWNTO** 0);

BEGIN

```
-- instantiation of the inv component
 INV_1 : inv GENERIC MAP (TP_LH,TP_HL) PORT MAP (I_4(0),Y_4(0));
 INV_2 : inv GENERIC MAP (TP_LH,TP_HL) PORT MAP (I_4(1),Y_4(1));
 INV_3 : inv GENERIC MAP (TP_LH,TP_HL) PORT MAP (I_4(2),Y_4(2));
 INV_4 : inv GENERIC MAP (TP_LH,TP_HL) PORT MAP (I_4(3),Y_4(3));
END inv4_struct;
```

Concurrent Statements

Concurrent statements are included within **ARCHITECTURE** bodies and **BLOCK** statements, representing concurrent digital behavior within the modeled design unit. These statements are executed in an asynchronous manner, with no defined order, modeling the overall behavior or structure of a design.

BLOCK Statement

A block statement groups related concurrent statements. The order of the concurrent statements does not matter, because all statements are always executed together.

If a guard expression appears after the reserved word **BLOCK**, a Boolean variable *GUARD* is automatically defined and set to the Boolean value of the guard expression. *GUARD* can then be tested within the block, to perform selected signal assignments or other statements only when the guard condition evaluates to *TRUE*.

```
[label_name:]
BLOCK [(guard expression)]
 …local declarations
BEGIN
 …concurrent statements
END BLOCK [label_name];
```

guard expression	Defines the value of the signal. The type of the expression must be BOOLEAN.
local declarations	Declarations include data types, constants, signals, files, components, attributes, subprograms, and other information used in the model description.
concurrent statements	Digital statements that are executed asynchronously with respect to each other, that describe a design unit in one or

	more modeling styles such as dataflow, structural, and behavior styles.

```
-- D Latch: Transfer D input to Q output when Enable = '1'
B1:
BLOCK (Enable = '1')
BEGIN
 Q <= guarded D AFTER 5ns;
END BLOCK B1;
```

PROCESS Statement

A **PROCESS** statement contains sequential statements but is itself a concurrent statement within an architecture. An independent sequential process represents the behavior of some portion of a design. The body of a process is a list of sequential statements. See "Processes" on page 6-55 .

The sequential statements in the process are executed in order, commencing with the beginning of simulation. After the last statement of a process has been executed, the process is repeated from the first statement, and continues to repeat until suspended. Processes can be suspended with a **WAIT** statement. The **WAIT** statement is a versatile statement that lets you suspend a process for a specific period of time, Boolean condition to occur, or/and an event to occur on a signal. The optional sensitivity list is equivalent to providing a **WAIT ON** statement and causes the process to be suspended. A process cannot have both a sensitivity list as well as a **WAIT ON** statement. See "WAIT Statement" on page 6-27 .

```
[label_name:]
PROCESS [(sensitivity_list)]
 …local declarations
BEGIN
 …sequential statements
END PROCESS [label_name];
```

sensitivity_list	List of signal names separated by a comma. The list represents a set of signals to which the **WAIT** statement within the process is sensitive. If no sensitivity list is defined, a **WAIT** statement within the process must be specified.

local declarations	Declarations declare data types, constants, signals, files, components, attributes, subprograms, and other information used in the model description.
sequential statements	Statements that are executed in the order in which they appear, that define algorithms for the execution of a subprogram or process.

```
clock <= NOT clock AFTER 10 us;

PROCESS (clock) -- sensitivity list consists of clock

 VARIABLE num_events : INTEGER := 0;

BEGIN

 num_events := num_events + 1;

END PROCESS;
```

Concurrent Procedure Call Statement

A concurrent procedure call, used in an **ARCHITECTURE** or a **BLOCK** statement, invokes an externally defined subprogram with parameters passed to it as necessary. See ***Procedures***.

```
[label_name:]

procedure_name [actual_parameter_list];
```

actual_parameters	Values of these parameters are transferred to the procedure for the formal parameters (the parameters specified in the procedure declaration).

```
-- procedure ReadMem procedure can be defined in a package and then used in many places

ReadMem (DataIn, DataOut, RW, Clk);
```

Concurrent ASSERT Statement

A concurrent **ASSERT** statement checks a condition (occurrence of an event) and provides a report if the condition is *FALSE*. You can specify a severity level to generate several types of messages.

```
ASSERT (condition)

[REPORT string] [SEVERITY severity_level];
```

condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE*

	or *FALSE*). If the condition expression is *FALSE* (indicating the assertion failed), the text specified in the string expression displays.
string	Defines the text string which appears in the message. String literals are one-dimensional arrays of characters enclosed in double quotation marks (” ”).
severity_level	Defines how the message appears and the effect on simulation. • **NOTE** – Message output in the message window. • **WARNING** – Message output in a dialog box and a request for user input to break or continue simulation. • **ERROR/FAILURE** – Message output in the message window and termination of simulation.

ASSERT (TS > 0.0)

REPORT "Sample Time must be specified" **SEVERITY** ERROR;

ASSERT (UL >= LL)

REPORT "Upper Limit must be greater than Lower limit" **SEVERITY** WARNING;

ASSERT (arg > 100.0)

REPORT "Invalid Function argument " & REAL'IMAGE(arg) & " at time " & TIME'IMAGE(now) & "." **SEVERITY** INFO;

Concurrent SIGNAL Assignment Statement

A concurrent signal assignment statement, one of the most common signal assignments, represents a process that assigns values to signals and specifies logical relationship between different signals. There is no significance to the order in which the assignments appear in the description. There are two forms: conditional signal assignment and selected signal assignment statements. See "SIGNAL Assignment Statement" on page 6-29 .

[label_name:]

target_signal <= var; -- signal assignment;

target_signal <= var1 WHEN condition ELSE var2; -- conditional signal assignment

target_signal <= var2 WHEN condition var3 WHEN OTHERS; -- selected signal assignment

target_signal <= var4 AFTER 5ns; -- signal assignment with delay

target_signal	Name of the signal which obtains the value.

conditional_signal_ assignment	Special form of signal assignment that assigns values only if a sequence of related conditions are true.
selected_signal_assignment	Special form of signal assignment that assigns values only if a sequence of related conditions are true but differs in that the input conditions specified have no implied priority.
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*).

ARCHITECTURE behav **OF** test **IS**

SIGNAL A: BIT :='1';

SIGNAL B: BIT :='0';

BEGIN

A <= B;

B <= '1'; -- value of B=1 value of A=1

END;

Component Instantiation Statement

The component instantiation statement instantiates (creates an instance of) predefined components within an architecture. Each such component is first defined in the declaration section of that architecture, and then instantiated one or more times in the body of the architecture. See ***WORK Library***.

label_name: **ENTITY** [library_name.]entity_name [(architecture_name)]

[**GENERIC MAP** (generic_map_list)]

[**PORT MAP** (port_map_list)];

generic_map_list	Specifies a list of generics (separated by a comma) which associate values with the formal generics in the corresponding component declaration or entity interface.
port_map_list	Specifies a list of ports (separated by a comma) which associate signals, quantities, and/or terminals with the formal ports in the corresponding component declaration or entity interface.

LIBRARY vhdlams_tutorial;

LIBRARY IEEE;

USE vhdlams_tutorial.res; -- model res of the tutorial_vhdlams library is declared

```
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY bat_multi IS
 GENERIC(
 factor : REAL := 1.0;
 v_init : VOLTAGE := 12.0);
 PORT(
 TERMINAL p,m : ELECTRICAL;
 QUANTITY v_out : OUT VOLTAGE := 0.0);
END ENTITY bat_multi;
ARCHITECTURE struct OF bat_multi IS
 CONSTANT ri: RESISTANCE := 1.0e-2;
 CONSTANT fc: CAPACITANCE := 60.0;
 CONSTANT rd: RESISTANCE := 4.0e-2;
 CONSTANT sc: CAPACITANCE := 2.0e4;
 TERMINAL t1,t2 : ELECTRICAL;
 QUANTITY v ACROSS p TO m;
BEGIN
 fc1: ENTITY WORK.cap(behav) -- instantiation of a component in the WORK lib
 GENERIC MAP (c_nom => fc*factor, v_init => v_init)
 PORT MAP (p => t1, m => m);
 sc1: ENTITY WORK.cap(behav) -- instantiation of a component in the WORK lib
 GENERIC MAP (c_nom => sc*factor, v_init => v_init)
 PORT MAP (p => t2, m => m);
 ri1: ENTITY res(behav) -- instantiation of a component declared in the entity
 GENERIC MAP (r_nom => ri)
 PORT MAP (p => p, m => t1);
 rd1 : ENTITY res(behav) -- instantiation of a component declared in the entity
 GENERIC MAP (r_nom => rd)
```

```
PORT MAP (p => t1, m => t2);
v_out == v;
END ARCHITECTURE struct;
```

Concurrent BREAK Statement

A **BREAK** statement explicitly indicates the occurrence of discontinuities in a VHDL-AMS description. The concurrent **BREAK** statement represents a process containing a **BREAK** statement.

[label_name:]

BREAK [break_list] [sensitivity_clause] **WHEN** condition;

break_list	Specifies a list of break elements (separated by a comma) which can cause a discontinuity.
sensitivity_clause	Specifies a list of signals to which the **BREAK** statement is sensitive.
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*).

```
ARCHITECTURE behav OF LMT IS
 SIGNAL lower_crossing : BOOLEAN := FALSE;
 SIGNAL upper_crossing : BOOLEAN := FALSE;
BEGIN
 BREAK ON lower_crossing;
 BREAK ON upper_crossing;
…
END ARCHITECTURE;
```

Sequential Statements

Sequential statements appear in process statements and in subprograms (procedures and functions), representing sequential behavior within the modeled design unit. These statements are executed in the order in which they appear in the model.

WAIT Statement

A **WAIT** statement suspends subprogram execution until a signal changes, a condition becomes *TRUE*, or a defined time period has elapsed. You cam also use combinations of these.	
WAIT [**ON** sensitivity_list] [**UNTIL** condition] [**FOR** time_expression];	
sensititvity_list	List of signals which can cause an event.
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*).
time_expression	Specifies a time expression which evaluates to a *TIME* value.
WAIT ON INPUT; **WAIT UNTIL** ctrl > 1.5; **WAIT FOR** TDELAY*unit_time;	

ASSERT Statement

A sequential **ASSERT** statement checks a condition and provides a report if the condition is not *TRUE*. You can specify a severity level to generate several types of messages.	
ASSERT (condition) [**REPORT** string] [**SEVERITY** severity_level];	
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition expression is *FALSE* (indicating the assertion failed), the text specified in the string expression displays.
string	Defines the text string which appears in the message. String literals are one-dimensional arrays of characters enclosed in double quotation marks (” ”).
severity_level	Defines how the message appears and the effect for simulation. • **NOTE** – Message output in the **Message Manager** pane. • **WARNING** – Message output in a dialog box and a request for user input to break or continue simulation. • **ERROR/FAILURE** – Message output in the **Message Manager** pane and termination of simulation.
ENTITY sequential_assert **IS** **END**; **ARCHITECTURE** behav **OF** sequential_assert **IS**	

```
SIGNAL clk:bit;
BEGIN
 clk<=not clk AFTER 1 ns;
 PROCESS(clk)
 variable a:integer := 0;
 BEGIN
 a:=a+1;
 ASSERT a/=1 REPORT "a!=1" SEVERITY FAILURE;
 ASSERT a/=2 REPORT "a!=2" SEVERITY WARNING;
 ASSERT a/=3 REPORT "a!=3" SEVERITY ERROR;
 ASSERT a<10 REPORT "a==" & INTEGER'IMAGE(a) SEVERITY NOTE;
 END PROCESS;
END;
```

SIGNAL Assignment Statement

A signal assignment statement assigns a waveform to one signal driver (edits the event queue). Signal assignments are always performed at the end of a process. See ***Concurrent SIGNAL Assignment Statement*** and ***Signal Assignments with Delay***.

[label_name:]

target_signal <= [delay_mechanism] source_signal

target_signal	Name of the signal which obtains the value.
delay_mechanism	Specifies a delay for signal assignment after one of the reserved words: **TRANPORT**, **REJECT**, or **INERTIAL**.
source_signal	Name of the signal which provides the value.

```
SIGNAL A: BIT :='1';
SIGNAL B: BIT :='0';
PROCESS
 BEGIN
 A <= B;
```

```
B <= '1'; -- value of B=1 value of A=0
WAIT;
END PROCESS;
```

VARIABLE Assignment Statement

A variable assignment updates a process, procedure, or function variable with the value of an expression. The update takes effect immediately, unlike signal assignments where there is at least a delta delay before the update is reflected.

```
[label_name:]
variable := expression;
```

variable_name	Name of the variable.
expression	Expression that performs an arithmetic or logical computation by applying an operator to one or more operands.

```
clock <= NOT clock AFTER 10 us;
PROCESS (clock)
 VARIABLE num_events : INTEGER := 0;
BEGIN
 num_events := num_events + 1;
END PROCESS;
```

Procedure Call Statement

A procedure call statement invokes an externally-defined subprogram in the same manner as a concurrent procedure call. A sequential procedure call statement differs from a concurrent procedure call in that it is placed in a process, procedure, or function, and is executed in the order in which it appears. See ***Subprograms***.

```
[label_name:]
procedure_name [(actual_parameter)]
```

formal_parameters	Specifies a list of parameters (constants, signals, or variables with the mode **in**, **out**, or **inout**).

```
-- procedure call with actual parameters int_var and out_vec
```

```
int2bin (in_var, out_vec);
```

IF Statement

An **IF** … **THEN** … **ELSE** statement performs a sequence of statements depending on the defined condition. **ELSIF** and **ELSE** clauses are optional.

```
[label_name:]
IF condition THEN
 …sequential statements
[{ELSIF condition THEN
 …sequential statements}]
[ELSE
 …sequential statements]
END IF [label_name];
```

condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition expression is *TRUE*, the corresponding statements after the condition are executed.
sequential statements	Statements that are executed in the order in which they appear; they define algorithms for the execution of a subprogram or process.

```
pwm_ctrl:
PROCESS (cond1,cond2)
BEGIN
 IF (cond1) THEN
 ctrl_sig <= 0.0;
 ELSIF (cond2 AND (NOT cond1)) THEN
 ctrl_sig <= 1.0;
 END IF;
END PROCESS pwm_ctrl;
```

CASE Statement

A **CASE** statement selects one of a number of alternative sequences of statements for execution based on the value of an expression. The choices must be constants of the same discrete type as the expression. Case choices can be expressions or ranges. Case statements must also include all possible values of the control expression. Use the **OTHERS** expression to guarantee that all conditions are covered.

```
[label_name:]
CASE control_expression IS
 WHEN choice1 =>
 …sequential statements
 WHEN choice2 =>
 …sequential statements
 WHEN OTHERS =>
 …sequential statements
END CASE [LABEL];
```

control_expression	Specifies a value that selects one statement sequence among the list of alternatives. The expression must be of a discrete type, or of a one-dimensional array.
choice	A choice specifies the value of the control expression for which the alternative is chosen. Each choice in a case statement alternative must be of the same type as the expression.
sequential statements	Statements that are executed in the order in which they appear, that define algorithms for the execution of a subprogram or process.

```
CASE J_K IS
 WHEN "11" =>
 state <= NOT state;
 WHEN "10" =>
 state <= '1';
 WHEN "01" =>
 state <= '0';
 WHEN OTHERS =>
 NULL;
```

END CASE;

LOOP Statements

The **WHILE** and **FOR** loop statements control the repetition of sequentially executed statements. Loop termination statements allow termination of one iteration, loop, or procedure.

NEXT [**WHEN** condition]; -- end current loop iteration

EXIT [**WHEN** condition]; -- exit innermost loop entirely

[label_name:]

LOOP

…sequential statements -- use exit statement to leave the loop

END LOOP [label];

[label_name:]

FOR variable **IN** start_val **TO** end_val **LOOP**

…sequential statements

END LOOP [LABEL];

[label_name:]

WHILE condition **LOOP**

…sequential statements

END LOOP [LABEL];

sequential statements	Statements that are executed in the order in which they appear, that define algorithms for the execution of a subprogram or process.
variable/start_val/end_val	Implicit index variables of the loop. Index variables need not be declared separately.
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition is *TRUE*, the corresponding statements after the condition executes.

LOOP

input_something;

exit when end_file;

```
END LOOP;

FOR I IN 1 TO 10 LOOP
 AA(I) := 0;
END LOOP;

WHILE NOT end_file LOOP
 input_something;
END LOOP;

PROCESS
 VARIABLE buf : LINE;
 VARIABLE t_val_old, t_val_new, t_val, s_val : REAL := 0.0;
BEGIN
 WHILE NOT ENDFILE(cyc) LOOP
 READLINE(cyc,buf);
 WAIT FOR 4ms;
 END LOOP;
 WAIT;
END PROCESS;
```

NEXT Statement

A **NEXT** statement causes the next iteration in a loop.

```
[label_name1:]
NEXT [label_name2] [when condition];
```

condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition is *TRUE*, the next iteration in the loop executes.

```
NEXT;
NEXT outer_loop;
NEXT WHEN A>B;
```

NEXT this_loop **WHEN** C=D **OR** done; -- done is a BOOLEAN variable

EXIT Statement

An **EXIT** statement causes the immediate termination of the loop.	
[label_name:] **EXIT** [label2] [**WHEN** condition];	
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition is *TRUE*, the loop terminates.
EXIT; **EXIT** outer_loop; **EXIT** when A>B; **EXIT** this_loop **WHEN** C=D **OR** done; -- done is a BOOLEAN variable	

RETURN Statement

A **RETURN** statement terminates a subprogram. A function definition requires a return statement to specify the return value. In a procedure definition, a **RETURN** statement is optional.	
[label_name:] **RETURN** [expressions];	
expression	Expression that performs an arithmetic or logical computation by applying an operator to one or more operands.
RETURN; -- from somewhere in a procedure **RETURN** a+b; -- returned value in a function	

NULL Statement

A **NULL** statement explicitly states that no action is required. It is often used in **CASE** statements because all choices must be covered, even if some choices are ignored.
[label_name:] **NULL**;

```
CASE J_K IS
 WHEN "11" =>
 state <= NOT state;
 WHEN "10" =>
 state <= '1';
 WHEN "01" =>
 state <= '0';
 WHEN OTHERS =>
 NULL;
END CASE;
```

BREAK Statement

A **BREAK** statement explicitly indicates the occurrence of discontinuities in a VHDL-AMS description. The sequential **BREAK** statement represents a process containing a **BREAK** statement.

The execution of a **BREAK** statement notifies the analog solver that it must determine the discontinuity augmentation set for the next analog solution point. It can also specify reset and initial values. The effect is conditional if the statement includes a condition.

[label_name:]

BREAK [break_list] [when condition];

break_list	Specifies a list of break elements (separated by a comma) which can cause a discontinuity.
condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*).

```
LIBRARY IEEE;
USE IEEE.MATH_REAL.ALL;

ENTITY sequential_break1 IS
END ENTITY sequential_break1;

ARCHITECTURE behav OF sequential_break1 IS
```

```
SIGNAL xs1,xs2 : BOOLEAN := FALSE;
SIGNAL aVal : REAL := 0.0;
BEGIN
xs1 <= NOT xs1 AFTER 1 ms;
xs2 <= NOT xs2 AFTER 2.5 ms;
PROCESS (xs1, xs2)
BEGIN
BREAK aVal => 4.0 WHEN xs2;
BREAK aVal => 2.0 WHEN xs1;
END PROCESS;
END ARCHITECTURE behav;
```

Simultaneous Statements

Simultaneous statements appear in the architecture body of a model and are placed in the same parts of a VHDL-AMS description as concurrent statements. Simultaneous statements are used to express Differential Algebraic Equations (DAE) that together with implicit equations describe the analog behavior of a model. This section describes the five types of simultaneous statements.

Simple Simultaneous Statement

A simple simultaneous statement specifies expressions that constrain the values of quantities by the analog solver.

[label_name:]

expression == expression;

expressions	Expressions that perform an arithmetic or logical computation by applying an operator to one or more operands.

```
TERMINAL plus, minus : ELECTRICAL;
QUANTITY voltage ACROSS current THROUGH plus TO minus;
QUANTITY power : REAL;
power == voltage * current; -- power calculation
```

Simultaneous IF Statement

A simultaneous **IF** statement specifies analog behavior of a system based on a set of conditions. A simultaneous **IF** statement differs from a sequential **IF** statement in that the simultaneous statement syntax is "**IF-USE-END USE**" while the sequential statement syntax is "**IF-THEN-END IF**".

[label_name:]

IF condition **USE**

…simultaneous_statements

[{ **ELSIF** condition **USE**

…simultaneous_statements }]

[**ELSE**

…simultaneous_statements]

END USE [label_name];

condition	Specifies an expression which evaluates to a BOOLEAN value (*TRUE* or *FALSE*). If the condition expression is *TRUE*, the corresponding statements after the condition execute.
simultaneous statements	Statements that execute at the same time with respect to each other; they describe Analog Differential Algebraic Equations (DAE).

IF (sw_on) **AND** (CTRL > 0.0)) **USE**

v == 0.0;

ELSE

i == 0.0;

END USE;

Simultaneous CASE Statement

A simultaneous **CASE** statement specifies analog behavior by selecting one of a number of alternatives based on the value of an expression. A simultaneous **CASE** statement differs from a sequential **CASE** statement in that the simultaneous statement syntax is "**CASE-USE-END CASE**" while the sequential statement syntax is "**CASE-IS-END CASE**".

[label_name:]

CASE control_expression **USE**

WHEN choices =>

…simultaneous_statements

{ when choices =>

…simultaneous_statements}

END CASE [**LABEL**];

control_expression	Specifies a value that selects one statement sequence among the list of alternatives. The expression must be of a discrete type, or of a one-dimensional array.
choice	A choice specifies the value of the control expression for which the alternative is chosen. Each choice in a case statement alternative must be of the same type as the expression.
simultaneous statements	Statements that execute at the same time with respect to each other; they describe Analog Differential Algebraic Equations (DAE).

```
LIBRARY IEEE;
USE IEEE.MATH_REAL.ALL;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;

ENTITY simultaneous_case IS
END ENTITY simultaneous_case;

ARCHITECTURE behav OF simultaneous_case IS
 TERMINAL n1, n2 : ELECTRICAL;
 QUANTITY vin ACROSS iin THROUGH n1 TO electrical_ref;
 QUANTITY vout ACROSS iout THROUGH n2 TO electrical_ref;
 CONSTANT Amp : REAL := 1.0;
 CONSTANT a : REAL := 1.0E3;
 CONSTANT f : REAL := 1.0E3;
 SIGNAL clk : INTEGER := 0;
BEGIN
 clk <= clk+1 AFTER 1ms;
```

```
vout == vin'SLEW(1.0e38,-1.0e38);
CASE clk USE
WHEN 0 => vin == Amp*1.0*sin(2.0*math_pi*f*15.0*NOW);
WHEN 1 => vin == Amp*2.0*sin(2.0*math_pi*f*15.0*NOW);
WHEN 2 => vin == Amp*3.0*sin(2.0*math_pi*f*15.0*NOW);
WHEN 3 => vin == Amp*4.0*sin(2.0*math_pi*f*15.0*NOW);
WHEN 4 => vin == Amp*5.0*sin(2.0*math_pi*f*15.0*NOW);
WHEN others => vin == Amp*6.0*sin(2.0*math_pi*f*15.0*NOW);
END CASE;
END ARCHITECTURE simple;
```

Simultaneous PROCEDURAL Statement

A simultaneous procedural statement provides a sequential notation for expressing differential and algebraic equations. Procedural statements are included within the architecture of a model and are not invoked with a calling mechanism.

```
[label_name:]
PROCEDURAL [IS]
 …declarations
BEGIN
 …sequential_statements
END PROCEDURAL [label_name];
```

declarations	Declarations include data types, constants, signals, files, variables, attributes, subprograms, and other information used in the model description.
sequential statements	Statements that execute in the order in which they appear; they define algorithms for the execution of a subprogram or process.

```
ENTITY fktarccos IS
 GENERIC (TS : REAL := 0.0);
 PORT (QUANTITY INPUT : IN REAL;
```

```
 QUANTITY VAL : OUT REAL);
END ENTITY fktarccos;

ARCHITECTURE behav OF fktarccos IS
 QUANTITY temp_val : REAL := 0.0;
BEGIN
 PROCEDURAL
 BEGIN
 IF (INPUT>-1.0) AND (INPUT<1.0) THEN
 temp_val := arccos(INPUT);
 ELSIF (INPUT = -1.0) THEN
 temp_val := MATH_PI;
 ELSE
 temp_val := 0.0;
 END IF;
 END PROCEDURAL;
 VAL == temp_val'ZOH(TS);
END behav;
```

Simultaneous NULL Statement

A simultaneous **NULL** statement explicitly specifies that no action be performed.

```
[label_name:]
NULL;
```

```

LIBRARY IEEE;
USE IEEE.ELECTRICAL_SYSTEMS.ALL;

ENTITY two_port IS
 GENERIC
```

```
(param : REAL := 1.0;
 mod_type : INTEGER := 0);
 -- RESISTOR(0); CAPACITOR(1);
 -- INDUCTOR(2)
 PORT
 (TERMINAL p,m : ELECTRICAL);
END ENTITY two_port;

ARCHITECTURE behav OF two_port IS
 QUANTITY v ACROSS i THROUGH p TO m;
BEGIN
 CASE mod_type USE
 WHEN 0 => v == i*param;
 WHEN 1 => i == param*v'dot;
 WHEN 2 => v == param*i'dot;
 WHEN OTHERS => NULL;
 END CASE;
END ARCHITECTURE behav;
```

Identifiers, Literals, and Expressions

Identifiers and literals are operands representing values of constants, variables, functions, and so on in expressions. Expressions perform arithmetic or logic computations by applying an operator to one or more operands.

Identifiers

An identifier is the name of a constant, variable, function, signal, entity, port, subprogram, or parameter and returns that object's value to an operand.

Identifiers in VHDL-AMS must begin with a letter, and can comprise any combination of letters, digits, and underscores. Identifiers must not end with an underscore and must not include two successive underscore character. Also no space is allowed within an identifier since a space is a separator.

An indexed identifier is the name of one element of an array variable or signal. The expression must return a value within the array's index range. The value returned to an operator is the specified array element.
letter { [underscore] letter_or_digit } -- identifier letter { [underscore] letter_or_digit } (expressions) -- indexed identifier
Voltage1, power_dissipated, product_of_sums voltage(2), current(3+1)

Literals

A literal (constant) operand can be a numeric, a character, an enumeration, or a string literal.
There are two forms of numeric literals: integer and real literals. Integer literals represent a whole number. Real literals can represent fractional numbers and always include a decimal point preceded and followed by at least one digit. Both types of numeric literals can use exponential notation and can expressed in a base. A decimal literal is written in base 10 and a based literal is written in a base from 2 to 16 and is composed of the base number, a number sign (#), the value in the given base, and another number sign (#).
base#digits# -- base must be a decimal number 2#101# -- decimal 5 16#AA# 16#f.1f#E+2 -- floating, exponent is decimal
Character literals are single characters enclosed in single quotation marks, for example 'a'. Character literals are used both as values for operators and in defining enumerated types, such as CHARACTER and BIT.
Enumeration literals are values of enumerated types. The two kinds of enumeration literals are character literals and identifiers. Character literals are described earlier. Enumeration identifiers are those listed in an enumeration type definition. If two enumerated types use the same literals, those literals are overloaded.
TYPE ENUM_1 **IS** (AAA, BBB, 'A', 'B', ZZZ); **TYPE** ENUM_2 **IS** (CCC, DDD, 'C', 'D', ZZZ); AAA -- Enumeration identifier of type ENUM_1 'B' -- Character literal of type ENUM_1

CCC -- Enumeration identifier of type ENUM_2 'D' -- Character literal of type ENUM_2 ENUM_1'(ZZZ) -- Qualified because overloaded
String literals are one-dimensional arrays of characters enclosed in double quotation marks (” ”). The two kinds are: • Character strings, which are sequences of characters in double quotation marks, for example, “ABCD”. • Bit strings, which are similar to character strings but represent binary (B), octal (O), or hexadecimal (X) values. For example, B”1101”. The digits in a bit string literal value can be separated with underscores (_) for readability.
x"ffe" -- 12-bit hexadecimal value, digits 0 to 9 and A to F -- each value represents four BITs in the generated bit vector (array) o"777" -- 9-bit octal value, digits 0 to 7 -- each octal digit represents three BITs in the generated bit vector (array) b"1111_1101_1101" -- 12-bit binary value, digits 0 or 1, each bit in the string represents -- one BIT in the generated bit vector (array)

Arithmetic and Logical Expressions

Expressions in VHDL-AMS are similar to those of most high-level languages. Data elements must be of the same type, or subtypes of the same base type. Expressions perform arithmetic or logical computations by applying an operator to one or more operands. Operators specify the computation to perform, operands are the data for the computation.	
Logical	• Predefined operators for types BIT and BOOLEAN. AND | OR | NAND | NOR | XOR | XNOR | NOT
Relational	• Predefined operators for all types except files. = | /= • Predefined operators for scalar and discrete array types. < | <= | > | >=

	Relational operators include tests for equality (=), inequality (/=), and ordering of operands (<, <=, >, >=). The operands of each relational operator must be of the same type. The result type of each relational operator is the predefined type BOOLEAN.
Shift	sll | srl Shift left/right logical. sla | sra Shift left/right arithmetic. rol | ror Rotate left/right logical. Operators for any one-dimensional array type whose element type is either of the predefined types BIT or BOOLEAN (left operant) and predefined type *INTEGER* (right operand). The result type of each shift operator is the same as the left operand.
Adding	• Predefined operators for any numeric type. + | – • Predefined operator for any one-dimensional array type. & Concatenate, a & b makes one array For each of these adding operators, the operands and the result are of the same type.
Multiplying	• Predefined for any integer and any floating point type. * | / • Predefined for any integer type. mod | rem a mod b takes sign of b, a rem b takes sign of a For each of these multiplying operators, the operands and the result are of the same type.
Miscellaneous	• Predefined for any numeric type. abs absolute value ** a** is a^2 The exponential operator ** is predefined for each integer type and for each floating point type.

Predefined Data Types

The following table shows predefined types from the package *STANDARD*. The data type names are not technically reserved words but they represent a common standard. To avoid

misunderstandings, do not re-define them. The standard package file is provided on the tutorial CD.

Data Type	Values	Example
BIT	'1', '0'	Q <= '1';
BIT_VECTOR	Array of bits.	DataOut <= "00010101";
BOOLEAN	TRUE, FALSE	EQ <= True;
INTEGER	-2, -1, 0, 1, 2, 3, 4 …	Count <= Count + 2;
REAL	1.0, -1.0E5	V1 = V2 / 5.3
TIME	1 ua, 7 ns, 100 ps	Q <= '1' after 6 ns;
CHARACTER	'a', 'b', '2', '$' …	CharData <= 'X';
STRING	Array of characters.	Msg <= "MEM: " & Addr

Predefined Type Declarations

```
TYPE INTEGER IS RANGE -2147483647 TO -2147483648;
TYPE BOOLEAN IS (FALSE,TRUE);
TYPE BIT IS ('0','1');
TYPE CHARACTER IS ( character_set_used_by_the_system );
TYPE SEVERITY_LEVEL IS (note,warning,error,failure);
TYPE REAL IS RANGE -9.9e99 TO 9.9e99;
TYPE TIME IS RANGE -9.9e99 TO 9.9e99
 UNITS fs;
 ps = 1000 fs;
 ns = 1000 ps;
 us = 1000 ns;
 ms = 1000 us;
 sec= 1000 ms;
 min= 60 sec;
 hr = 60 min;
```

```
END UNITS;

TYPE DOMAIN_TYPE IS (QUIESCENT_DOMAIN, TIME_DOMAIN, FREQUENCY_
DOMAIN);

TYPE STRING IS ARRAY( 1 TO 2147483647 ) OF CHARACTER;

TYPE BIT_VECTOR IS ARRAY( 0 TO 2147483647 ) OF BIT;
```

Predefined Attributes

An object attribute returns information about a signal or data type. Predefined attributes denote values, functions, types, and ranges associated with various kinds of named entities.

The syntax of an attribute is some named entity followed by an apostrophe and one of the following attribute names. A parameter list is used with some attributes. The following abbreviations are used in the tables:

- **Q** – represents a quantity
- **S** – represents a signal
- **X** – represents a signal or variable
- **T** – represents a type
- **A** – represents an array or constrained array type
- **t** – represents an expression for time
- **e** – represent a static expression

Quantity Attributes

```
Q'DOT
-- value of the derivative with respect to time of Q at the time the attribute
-- is evaluated
```

```
Q'INTEG
-- value of the time integral of Q from time 0 to the time the attribute is evaluated
```

```
Q'DELAYED[(t) ]
-- value of quantity Q delayed by t; if t is omitted, it defaults to 0.0
```

```
Q'ABOVE(e)
-- true if Q-e is sufficiently larger than 0.0, false if Q-e is sufficiently smaller
-- than 0.0. This attribute always returns a Boolean signal (TRUE/FALSE).
```

```
Q'ZOH(e[,INITIAL_DELAY])
-- constant value of Q at the sample times INITIAL_DELAY+k*e until the next sample
-- time, with e as sample frequency and INITIAL_DELAY as the time of the first sampling
-- (k is any non-negative integer); if INITIAL_DELAY is omitted, it defaults to 0.0
```

```
Q'LTF(NUM,DEN)
-- Laplace transfer function of Q with NUM as the numerator and NEN as
-- denominator polynomials
```

```
Q'ZTF(NUM,DEN,e[,INITIAL_DELAY])
-- Z transfer function of Q with NUM as the numerator and NEN as denominator
-- polynomials, with e as sample frequency and INITIAL_DELAY as the time of the
-- first sampling; if INITIAL_DELAY is omitted, it defaults to 0.0
```

```
S'RAMP[(TRISE[,TFALL])]
-- quantity which follows the corresponding value of S with delay of rise time and
-- fall time; if TRISE or TFALL is greater than 0.0, the corresponding value change
-- is linear from the current value of S to its new value, whenever S has an event
```

```
S'SLEW[(RISING_SLOPE[,FALLING_SLOPE])]
-- quantity which follows the corresponding value of S with rising slope and falling
-- slope; if RISING_SLOPE is less than REAL'HIGH, or if the value or FALLING_SLOPE is
-- greater than REAL'LOW, the corresponding value change is linear from the current
-- value of S to its new value, whenever S has an event
```

```
Q'SLEW[(MAX_RISING_SLOPE[,MAX_FALLING_SLOPE])]
-- quantity which follows the corresponding value of Q, but its derivative time
-- is limited by MAX_RISING_SLOPE and MAX_FALLING_SLOPE
```

Signal Attributes

```
S'DELAYED[(t)] -- value of signal S at time now-t; if t is omitted, it defaults to 0.0
S'STABLE -- true if no event is occurring on signal S
S'STABLE(t) -- true if no event has occurred on signal S for t time units
```

S'QUIET -- true if signal S is quiet (no event this simulation cycle)

S'QUIET(t) -- true if signal S has been quiet for t time units

S'TRANSACTION -- bit value which toggles each time when signal S changes

S'EVENT -- true if an event has occurred on signal S in the current cycle

S'ACTIVE -- true if signal S is active in the current cycle

S'LAST_EVENT -- time since the last event on signal S

S'LAST_ACTIVE -- time since signal S was last active

S'LAST_VALUE -- value of signal S prior to latest change

S'DRIVING -- false if the current driver of signal S is a null transaction

S'DRIVING_VALUE --current driving value of signal S

Data Type Bounds

T'BASE -- base type of data type T

T'LEFT -- left bound of data type T (largest if downto)

T'RIGHT -- right bound of data type T (smallest if downto)

T'HIGH -- upper bound of data type T (may differ from left bound)

T'LOW -- lower bound of data type T

T'ASCENDING -- true if range of T defined with to

Enumeration Data Types

T'IMAGE(X) -- string representation of X that is of discrete type T

T'VALUE(X) -- value of discrete type T converted from the string X

T'POS(X) -- integer position number of value of X of discrete type T

T'VAL(X) -- value of discrete type T at position number X

T'SUCC(X) -- value of discrete type T at position number X-1

T'PRED(X) -- value of discrete type T at position number X+1

T'LEFTOF(X) -- value of discrete type T at position left of X
T'RIGHTOF(X) is the value of discrete type T at position right of X

Array Indexes for an Array A

For multi-dimensional array, N^{th} index must be indicated in the attribute specifier. N can be omitted for a one-dimensional array.
A'LEFT -- leftmost subscript of array A or constrained array type
A'LEFT(N) -- leftmost subscript of dimension N of array A
A'RIGHT -- rightmost subscript of array A or constrained array type
A'RIGHT(N) -- rightmost subscript of dimension N of array A
A'HIGH -- highest subscript of array A or constrained array type
A'HIGH(N) -- highest subscript of dimension N of array A
A'LOW -- lowest subscript of array A or constrained array type
A'LOW(N) -- lowest subscript of dimension N of array A
A'RANGE -- range A'LEFT to A'RIGHT or A'LEFT downto A'RIGHT
A'RANGE(N) -- RANGE of dimension N of array A
A'REVERSE_RANGE -- range of array A with to and downto reversed
A'REVERSE_RANGE(N) -- REVERSE_RANGE of dimension N of array A
A'LENGTH -- integer value of the number of elements in array A
A'LENGTH(N) -- number of elements of dimension N of array A
A'ASCENDING -- Boolean True if range of array A is defined with to
A'ASCENDING(N) -- Boolean True if dimension N of array A is defined with to

Reserved Words

The following words are reserved for the VHDL-AMS language, so they cannot be used as identifiers. Reserved words are printed in bold in this document.

Word		Description
ABS		Operator, absolute value of right operand.
ACCESS		Declares a type for creating access objects, pointers.
ACROSS		Declares the across quantity of a particular **NATURE** type.
AFTER		Defines a delay value (time after *NOW*) in signal assignment statements.
ALIAS	*	Defines an alternate name for an object.
ALL		Makes all items in a package visible, refers to all names of a class, or refers to all instances of a component.
AND		Operator, logical **AND** of left and right operands.
ARCHITECTURE		Primary design unit. Defines an architecture.
ARRAY		Declares an array data type containing a collection of elements that are all of the same type (array, vector, or matrix).
ASSERT		Checks a condition (occurrence of an event) and provides a report if the condition is not *TRUE*.
ATTRIBUTE		Defines an attribute name and its type.
BEGIN		Defines the begin of a **BLOCK**, **ENTITY**, **ARCHITECTURE**, **IF**, or **PROCESS** statement.
BLOCK		Starts the description of a block structure.
BODY		Starts the description of various subprograms that are declared by the package body's associated package declaration.
BUFFER		Port mode. Indicates a port which can be used for both input and output, and it can have only one source.
BUS		Signal mode. Defines a bus that floats to a user-specified value when all of its drivers are turned off.
CASE		Sequential statement. Executes one of several sequences of statements within a **PROCESS**, **PROCEDURE**, or **FUNCTION** structure, depending on the value of a single expression.
COMPONENT		Starts a component declaration.
CONFIGURATION	*	Primary design unit. Defines a configuration for an **ENTITY**.
CONSTANT		Declares an identifier name for a constant value (read only).
DISCONNECT	**	Signal driver condition. Defines the time delay to disconnect the guarded feature of a signal which is part of a guarded signal statement.
DOWNTO		Defines a descending range in a **RANGE** statement or other statement which includes a range.

Word		Description
ELSE		Defines the final alternative in an **IF** or **WHEN** statement.
ELSIF		Defines an interim alternative in an **IF** statement.
END		Defines the end of an **ENTITY**, **ARCHITECTURE**, **CONFIGURATION**, **PACKAGE**, **PACKAGE** body, or **PROCESS** statement.
ENTITY		Primary design unit. Declares the input and output ports of a design.
EXIT		Sequential statement. Causes the immediate termination of a **LOOP**.
FILE		Declares a type for creating file handles and an identifier as a file object.
FOR		Identifies a parameter specification in a **LOOP** statement or time expression in a **WAIT** statement.
FUNCTION		Defines a group of sequential statements that are executed when the function is called.
GENERATE		Copies a set of concurrent statements, or executes a set of concurrent statements selectively if a specified condition is met.
GENERIC		Specifies static information to be communicated to a model from its environment for all architectures.
GROUP	**	Defines a group template or specific group that can get an attribute.
GUARDED		Limits the execution of a **SIGNAL** statement. Causes a **WAIT** until a signal changes from *FALSE* to *TRUE*.
IF		Sequential statement. Performs a sequence of statements dependent on a defined condition.
IMPURE		Declares a function that is assumed to have side effects (return a different value given the same actual parameters).
IN		Port mode. Indicates a port which can be used for input.
INERTIAL		Clause in delay mechanism, followed from a time. Signals smaller than delay time are suppressed.
INOUT		Port mode. Indicates a port which can be used for both input and output.
IS		Used as a connective in various statements.
LABEL		Defines a label name in attribute statements.
LIBRARY		Designates a simple library name to identify libraries from which design units can be referenced.
LINKAGE		Port mode. Indicates a port which can be used for both input and output, and it can only correspond to a signal.
LITERAL		Entity class. Specifies an **ENTITY** in **ATTRIBUTE** statements.

Word		Description
LOOP		Sequential statement. Executes a series of sequential statements multiple times.
MAP		Associates values of constants or ports within a structure to constants and ports outside the structure.
MOD		Operator, left operand modulo right operand.
NAND		Operator, logical **NAND** of left and right operands.
NATURE		Defines the **ACROSS** and **THROUGH** types of an object with a particular energy domain (nature).
NEW	**	Creates an object of a specified type and returns an access value that refers to the created object.
NEXT		Sequential statement. Cause the next iteration in a **LOOP**.
NOISE	**	Declares the source aspect of a noise source quantity (serves as a source in a frequency domain model) in a quantity declaration.
NOR		Operator, logical **NOR** of left and right operands.
NOT		Operator, complement of right operand.
NULL		Sequential statement. Specifies explicitly that no action is needed.
OF		Used as a connective in various statements.
ON		Used as a connective in various statements.
OPEN		Designates the initial file characteristic or indicates a port that is not connected to any signal.
OR		Operator, logical **OR** of left and right operands.
OTHERS		Defines all remaining elements for example in a **CASE** statement, a selected assignment, and **ATTRIBUTE** specification.
OUT		Port mode. Indicates a **PORT** which can be used for output.
PACKAGE		Primary design unit. Defines a package and package body.
PORT		Specifies dynamic information to be communicated to a model from its environment for all architectures.
POSTPONED		Declares a **PROCESS** as a postponed process. Postponed processes do not execute until the final simulation cycle at the currently modeled time.
PROCEDURE		Starts the description of a group of sequential statements that are to be executed when the procedure is called.
PROCESS		Starts the description of a group of sequential statements and is considered to be a single concurrent statement within a VHDL-AMS architecture.

Word		Description
PURE		Declares a **FUNCTION** that is assumed to have no side effects.
QUANTITY		Declares a **PORT** in an **ENTITY** declaration or **ACROSS**, **THROUGH**, and **REFERENCE** types in a **NATURE** declaration.
RANGE		Defines a range constraint for a scalar type.
RECORD	*	Defines a record type and its corresponding element types.
REFERENCE		Declares the reference quantity of a particular **NATURE** type.
REGISTER		Signal mode. Defines a storage register that retains its last driven value when all of its drivers are turned off.
REJECT		Clause in delay mechanism, followed from a time. Signals smaller than reject time are suppressed.
REM		Operator, remainder of left operand divided by right operand.
REPORT		Defines a text string that is displayed when the condition in an **ASSERT** statement is not *TRUE*.
RETURN		Sequential statement. Terminates a subprogram (**PROCEDURE** or **FUNCTION**) and returns control to the calling object.
ROL		Operator, left operand rotated left by right operand.
ROR		Operator, left operand rotated right by right operand.
SELECT		Selects and assigns a value to a target signal from among a list of alternatives, based on the value of a given expression.
SEVERITY		Defines the type of the message in an **ASSERT** statement.
SHARED		Declares shared objects.
SIGNAL		Declares an identifier as a signal object.
SLA		Operator, left operand shifted left arithmetic by right operand.
SLL		Operator, left operand shifted left logical by right operand.
SPECTRUM		Declares the source aspect of a spectral source quantity (serves as a source in a noise domain model) in a **QUANTITY** declaration.
SRA		Operator, left operand shifted right arithmetic by right operand.
SRL		Operator, left operand shifted right logical by right operand.
SUBNATURE		Declares a nature that is a subnature of an existing **NATURE**.
SUBTYPE		Declares a type that is a subtype of an existing **TYPE**.
TERMINAL		Declares a terminal object of a particular **NATURE**.
THEN		Defines the first choice in an **IF** statement when the condition is *TRUE*.
THROUGH		Declares the through quantity of a particular **NATURE** type.

Word		Description
TO		Defines an ascending range in a **RANGE** statement or other statement which includes a range.
TOLERANCE		Declares a tolerance that can be applied to scalar quantities.
TRANSPORT		Clause in delay mechanism, followed from a time. Defines a non-inertial delay in a signal assignment statement.
TYPE		Declares a type.
UNAFFECTED		Indicates in a conditional or selected signal assignment when the signal is not to be given a new value.
UNITS		Declares physical types.
UNTIL		Defines a condition to terminate a **WAIT** statement.
USE		Makes a package available to this design unit.
VARIABLE		Declares an identifier as a variable object.
WAIT		Suspends temporarily a process until a specified time has passed, a specified condition is met, or an event occurs which affects one or more signals.
WHEN		Defines a condition during which an **EXIT** or **NEXT** statement will be executed or defines a choice (or choices) within a **CASE** statement.
WHILE		Defines a condition during which a **LOOP** will be executed.
WITH		Defines an expression in a selected signal assignment statement.
XNOR		Operator, logical exclusive **NOR** of left and right operands.
XOR		Operator, logical exclusive **OR** of left and right operands.

***** Partially supported in Twin Builder 12.0.

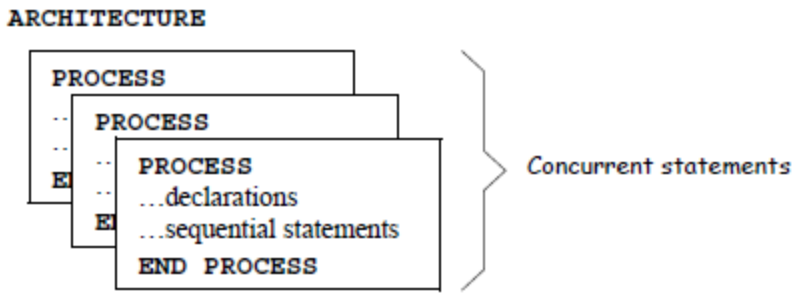
****** Not supported in Twin Builder 12.0.

Modeling Aspects in Twin Builder

Processes

A process statement defines an independent sequential process and is considered to be a single concurrent statement within a VHDL-AMS architecture.

Process statements contain sequential statements but are themselves concurrent statements. They are the primary means of defining sequential statements. A process statement can include all declarations and sequential statements.



Quantities, Signals, and Variables

	Description	Direction	Assignment
Quantities	Analog objects	**IN OUT**	== =>
Signals	Digital objects	**IN OUT INOUT**	<=
Variables	Auxiliary objects to store intermediate values	None	:=

Signal Assignments with Delay

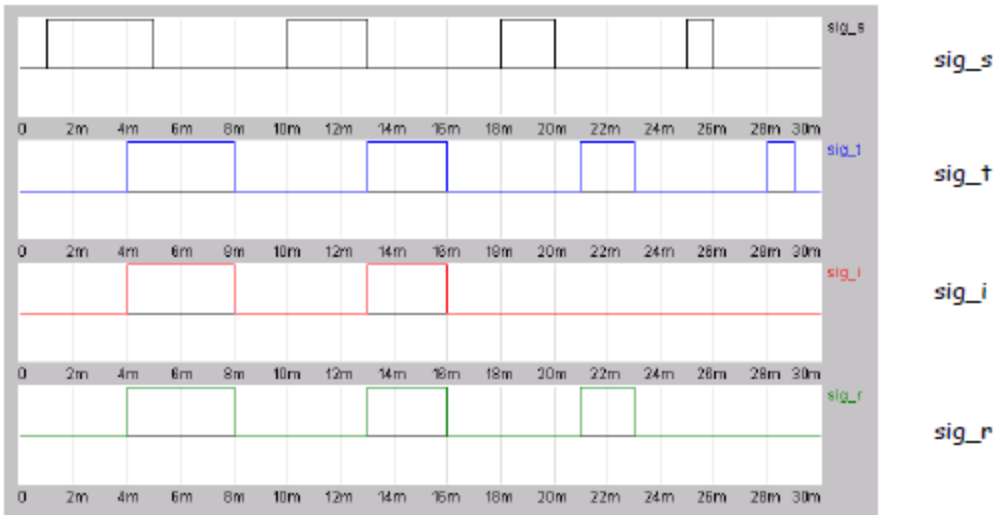
VHDL-AMS offers several variants to perform signal assignments. The following example shows three delay forms:

```
ENTITY sequential_sig_assign IS
 PORT(SIGNAL output: out bit);
END;

ARCHITECTURE behav OF sequential_sig_assign IS
 SIGNAL sig_s,sig_t,sig_i,sig_r : bit;
BEGIN
 sig_s <= '1' AFTER 1 ms, '0' AFTER 5 ms,
 '1' AFTER 10 ms, '0' AFTER 13 ms,
 '1' AFTER 18 ms, '0' AFTER 20 ms,
 '1' AFTER 25 ms, '0' AFTER 26 ms;
PROCESS (sig_s)
 BEGIN
```

```
sig_t <= TRANSPORT sig_s AFTER 3 ms; -- signal assignment 1:1
sig_i <= INERTIAL sig_s AFTER 3 ms;
-- signal assignment without signals smaller than delay time are suppressed
sig_r <= REJECT 2 ms INERTIAL sig_s AFTER 3 ms;
-- signal assignment without signals smaller than reject time are suppressed
END PROCESS;
END;
```

The following figure shows the results of the previous modeling code. The signal *sig_s* is delayed and assigned to signals in three different ways:

- The signal *sig_t* represents *sig_s* with a delay of 3ms.
- The signal *sig_i* represents *sig_s* with a delay of 3ms without signal changes occurring in an interval smaller than delay time.
- The signal *sig_r* represents *sig_s* with a delay of 3ms without signal changes occurring in an interval smaller than reject time.



Data Exchange in Mixed-Signal Models

Mixed-signal models use analog and digital statements in their model description. Mixed-signal assignments should be avoided when other modeling variants can be used. Compiler errors or simulator instabilities can occur when data assignments are not performed in the correct way.

The following sections show examples of assignments from digital to analog values and analog to digital values. In the examples, *quantity_name* stands for an analog value, *signal_name* for a digital value.

Signal to Quantity Assignment (Digital to Analog)

The first example shows a value assigned in the correct syntax but without any support for the solver to find a good solution.

```
quantity_name == signal_name; syntax correct but solver problems can occur
```

To improve the solver stability, use a '*RAMP* or '*SLEW* attribute or a **BREAK** statement. The '*RAMP* and '*SLEW* attributes, as well as the **BREAK** statement, force a synchronization on the minimum simulator time step HMIN and thus minimize the error deviation of the solver.

The following examples show value assignments in connection with variants to define rise/fall time and positive/negative slew rate of a digital signal. The rise and fall time as well as positive and negative slew rate should be chosen so as to avoid infinite values of the first derivative for these quantities since this can cause simulator instabilities.

```
quantity_name == signal_name'RAMP(1.0e-9, 2.0e-9); -- define rise and fall time
quantity_name == signal_name'SLEW(100.0, 200.0); -- limit pos and neg slew rate
```

The following examples show value assignments dependent on events applied to a signal.

```
BREAK ON signal_name; -- assignment when event on signal_name
BREAK quantity_name'DOT => 1000.0 ON signal_name
BREAK qv => -qv when not quantity_'Above(0.0);
quantity_name==signal_name;
```

Quantity to Signal Assignment (Analog to Digital)

The first example shows an incorrect way of using a value assignment. The statement is performed only during initialization, and the signal value is unchanged during the rest of the simulation. Signal assignments are characterized by events, and in this case quantities do not create any events.

```
signal_name <= quantity_name; -- only at init step, wrong assignment
```

The following example shows a value assignment if the quantity crosses a threshold value of 3.0. The signal value is of type BOOLEAN since the 'ABOVE attribute returns a *TRUE* or *FALSE*. In the next process, the actual value of the quantity is assigned to the signal. The next value assignment follows only after the quantity crosses the threshold again.

```
signal_name <= quantity_name'ABOVE(3.0); -- value of signal_name is true or false
PROCESS(signal_name)
BEGIN
 signal_name <= quantity_name; -- value of signal_name is actual value + maximum hmin
END;
```

The following example shows a value assignment at each time interval specified as delay:

```
clock <= NOT clock AFTER 1ms; -- generates an event at each time step defined as delay
PROCESS(clock)
BEGIN
 signal_name <= quantity_name;
END;
```

Solvability

The analog solver needs a specific set of equations when it evaluates the statements in the model.

- The number of equations specified in the model must be equal to the sum of the number of free quantities, through quantities, and port quantities of mode **OUT**.
- All free quantities and port quantities of mode **OUT** must appear in a simultaneous equation within the architecture of the model.

The following example uses two relevant quantities: *I* (port quantity of mode **OUT**) and *ct* through quantity. Both quantities occur in the equations of the model.

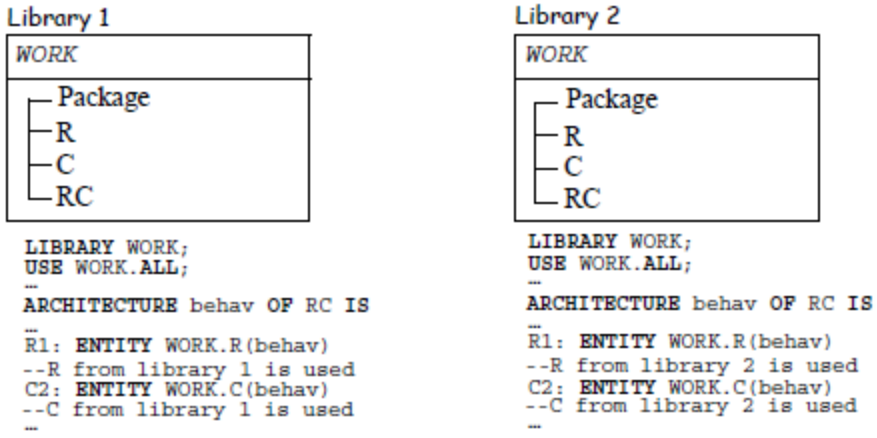
```
LIBRARY IEEE;
```

```
USE IEEE.ELECTRICAL_SYSTEMS.ALL;
ENTITY AM IS
 PORT (QUANTITY I : OUT REAL; -- port quantity of mode out
 TERMINAL p,m : ELECTRICAL);
END ENTITY AM;
ARCHITECTURE behav OF AM IS
 QUANTITY v ACROSS ct THROUGH p TO m; -- through quantity ct
BEGIN
 I == ct; -- first equation with port and through quantity
 v == 0.0; -- second equation
END behav;
```

The digital solver needs a minimum simulation step HMIN that is less than or equal to the least delay specified with an **AFTER** statement.

WORK Library

Packages and models of a library are always compiled in the current WORK library. All packages and models of that library are visible to other models of the same library and can be instantiated as components in these models. If a model that instantiates models of the WORK library is copied to another library, the WORK library of this other library is used. That means, the same instantiation uses different models, because the WORK library was changed.



Models defined in VHDL-AMS text subsheets that are placed on a model sheet are also compiled in the current sheet's WORK library. All models of these subsheets are visible to other subsheets of the same sheet at any hierarchy level.

Entity names defined in subsheets must be unique within the entire sheet, because an entity with a duplicate name will overwrite the previously defined one.

Alias for File Names

You can define a file alias name for a library name in Twin Builder. Select **Tools** > **Library Tools** > **Manage Aliases**. Create a new file alias and define alias name and path of the corresponding file. To use files with names that do not conform to VHDL-AMS identifier rules, such as a file name with spaces, specify a file alias.

Values on Sheet

In the Twin Builder Schematic, parameter values for VHDL-AMS models are entered in component dialogs. Values can be assigned to each parameter corresponding to the inputs defined in the model entity. If no value is entered, the default value specified in the model is used.

> **Note:**
>
> The syntax of user-defined values must conform to the SML conventions. VHDL-AMS attributes and syntax cannot be used.

Each value assigned to a parameter is from type *REAL*. The following table lists possible value assignments for VHDL-AMS data types and their effects.

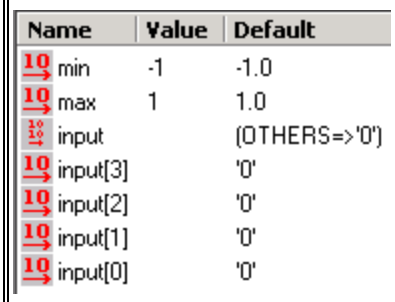
Name	Value	Default	Data Type	Object	Description
inp1	-1	-1.0	REAL	SIGNAL	All values are possible. If no default value is specified, name'LEFT is used (-1.7e-308).
inp2	var1	0.0	REAL	QUANTITY	
inp3	ctrl	'1'	BIT	SIGNAL	Values between 0 and 2 are possible, otherwise an error message occurs during simulation. if *inp*<1 then use value 0 if *inp*>=1 the use value 1
inp4	0	inp3'left	BIT	SIGNAL GENERIC	
inp5	2.5	10	INTEGER	SIGNAL	All values are possible. If *inp* is not integer value, the integer part of the value is used.
inp6	var2	0	INTEGER	GENERIC	
inp7	7	'H'	STD_ LOGIC	SIGNAL	Values between 0 and 8 are possible, otherwise an error message occurs during

Name	Value	Default	Data Type	Object	Description
inp8	var3	'0'	STD_ LOGIC	SIGNAL	simulation.

In addition to value specifications in the model dialog box, *OmniCaster* models can be used to perform value assignments. *OmniCasters* convert different data types correctly if a conversion is possible, otherwise they reject a connection between the parameters.

Vector Inputs on Sheet

Vector inputs appear in the component input dialog box as parameters with index. Values for dynamic vectors can only be assigned with a wire.




```
ENTITY DAC IS
 GENERIC (MIN: REAL := -1.0;
 MAX: REAL := 1.0);
 PORT (
 SIGNAL INPUT: IN BIT_VECTOR(3 DOWNTO 0):=(OTHERS =>'0');
 QUANTITY VAL: OUT REAL := 0.0);
END ENTITY DAC;
```

A - Appendix

This appendix contains the following sections:

Twin Builder Glossary

Term	Description
AC simulation	Harmonic analysis of a model.
AC simulator	Twin Builder simulator for AC analysis.
Action type	Defines how an action in a state is processed.
.afa file	Analytical Frequency Analysis file containing data of an analytical frequency analysis.
Analytic frequency step response	Twin Builder module to compute frequency step response information for a given transfer function.
Animated Symbol	Symbol for a component or a macro that changes to reflect changes in values assigned to it. The symbol can be modified by the user with an interaction button or by a system value during the simulation. Animated symbols can be generated using the Symbol Editor.
.asmd file	Twin Builder file containing data for a model library.
.aws file	Simulator file containing initial values of capacitors and inductors.
Backplane	Common communication and control structure for all Twin Builder simulators.
Basics	Twin Builder library containing the basic modeling elements for the Twin Builder simulators.
Behavioral model	Smallest model unit that cannot be subdivided further.
Best representation	Function for scaling all output channels of a graph window to maximum size.
Bezier	Bezier curves are used in computer graphics to produce curves which appear reasonably smooth at all scales. The curves are constructed as a

Term	Description
	sequence of cubic segments, rather than linear ones. Bezier curves use a construction in which the interpolating polynomials depend on certain control points. The mathematics of these curves is classical, but it was a French automobile engineer Pierre Bezier who introduced their use in computer graphics.
Block	Linear or nonlinear transfer function, basic element of block diagrams.
Block diagram	Combination of blocks to describe the dynamic behavior of systems.
Block Diagram Module (BDM)	Twin Builder sub simulator to analyze simulation models described using block diagrams. Uses Euler formula and distributed integration algorithms.
Bookmark	Can be used in the Twin Builder editor to mark a position and find it again.
Characteristic	Function or data set to describe a nonlinear characteristic of a Twin Builder component, block or other model.
C-Interface	C/C++ programming interface for the integration of user defined nonlinear models or components into the simulation model.
Color scheme	Predefined or user defined color settings for screen outputs or printing.
Compiler	Program for the translation of the SML description of a simulation model into a simulator specific format.
Computation sequence	Sequence in which blocks in a simulation model are computed.
Connection rule	Rule determining which element can be connected to another under certain conditions.
Conservative node	Connection of two or more circuit component terminals.
Coordinate system	Reference system for the display of numerical values. It can be linear or logarithmic.
Crossing over	Exchange of genetic material between chromosomes in a genetic algorithm.
Cursor	Positioning element to determine the value of a quantity in a coordinate system.
Data cache	Saves the data of the last simulation run.
Data channel	Simulation data for a specific quantity stored in a file or transferred to an active element.
Data filter	Function to select data by user defined criteria.
Data format	Defines how data are stored or exchanged between programs.
Data set	Simulation data of a simulation run at a given time step.
DC simulation	DC operating point analysis of a model.

Term	Description
DC simulator	Twin Builder simulator for DC analysis.
DLL	Dynamic link library containing program components or models. It is loaded automatically upon request.
Dongle	Software protection device connected to the printer port of the computer.
Electric circuit	Combination of electric components, connected by ideal wires.
Electric Circuit Module (ECM)	Twin Builder sub simulator to analyze simulation models described using electric circuits. Uses Euler or Trapezoid algorithm and modified nodal approach.
Element	Any item that can be placed on a Schematic sheet, including models, text elements, display elements, drawing elements, subsheets, and so on.
Entity	VHDL-AMS term to describe the interface of a behavioral or structural model.
Euler formula	Numerical integration algorithm used inside Twin Builder.
Evaluation function	Function evaluating the quality of a solution compared to the defined optimum.
Export filter	Special program for the export of simulation data into other applications.
FFT	Fast Fourier Transformation.
File output	Saves a system quantity online during the simulation in a file.
Fitness function	Function describing the fitness of an individual (parameter constellation) in a genetic algorithm.
Formula Module (FML)	Twin Builder’s integrated expression evaluator.
Frequency step response analysis	Special mode in the Experiment tool, determining the frequency behavior of a simulation model.
Genetic Algorithm	Optimization method of the experiment tool with automatic parameter variation and target function determination.
Grid lines	Grid lines in a coordinate system.
HMAX	Simulation parameter, maximum step size for the integration algorithm.
HMIN	Simulation parameter, minimum step size for the integration algorithm.
ID	Numeric value for the distinction of sheet elements.
Information window	Display of warnings, errors, program status for the Twin Builder modules.
Initial condition	Defines the initial value for energy storing electrical components.
Initial state	A state that is active at the beginning of the simulation.

Term	Description
INT	Simulation parameter, defines the integration algorithm used for the simulation of electric circuits.
Interactivity pad	Part of an animated symbol, used to change the behavior and/or shape of a symbol (model component) by the user.
Iteration	Part of the integration process for the solution of nonlinear problems.
Jacobian Matrix	Coefficient matrix for the numerical integration algorithm.
Keyword	Can be used to do a search in a model database.
.krn file	Simulator file containing simulation status information. It can be used as a starting point for the next simulation.
Language concept	Settings to define the use of language for program (menus and dialogs) and libraries.
Library	Database containing a set of Twin Builder basic elements and/or macro models.
Local discretization error (LDF)	Simulation parameter, determines the accuracy of the computation according to the dynamics of the electric circuits.
.log file	Experiment tool file containing the protocol of an experiment.
Macro	Contains one or more elements of a model description that can be used as single elements.
Main skeleton (.skl)	Special file containing basic descriptions for the generation of the model description file. Must not be modified.
Maximum current error (IEMAX)	Simulation parameter that determines the accuracy of the right side computation of the differential equation system for current lines.
Maximum number of iterations (Iteratmax)	Simulation parameter that limits the number of iteration loops for the nonlinear iteration process.
Maximum voltage error (VEMAX)	Simulation parameter that determines the accuracy of the right side computation of the differential equation system for voltage lines.
.mda/.mdk file	Twin Builder data file of simulation results and characteristics (former format).
.mdx file	Twin Builder data file of simulation results and characteristics.
Model	All elements in the libraries and subsheets (except for display elements).
Model tree	Hierarchical list box containing the available elements for the graphical

Term	Description
	modeling.
Module	Twin Builder sub simulator or program.
Monitor	Special window for the display of simulation status and progress.
Monte Carlo Analysis	Optimization method of the experiment tool with automatic parameter generation and characteristic value determination.
Multi simulation	Analysis method of the experiment tool, batch mode computation of a simulation model with different, user defined parameter sets.
Mutation	Part of the genetic algorithm that generates new individuals (parameter sets) by randomly modified parameters.
Network installation	Installation process of a Twin Builder network version.
Newton-Raphson-Algorithm	Nonlinear iteration algorithm used in Twin Builder.
Non-conservative node	Connection of two or more non-circuit component terminals.
Object	Element of the graphic model description linked to another WINDOWS application (OLE).
Object browser	Special window for the display and browsing of the elements of a graphical model description.
Offline	When the simulation is not running in Twin Builder.
Online	During the simulation in Twin Builder.
Online graphic	Online display of simulation results during the simulation run.
Online graphic output	Display of a system quantity during the simulation in a Display element or the ViewTool.
Option	Optional Twin Builder application. It can be registered using the installation manager.
Output	Definition of a specific quantity of the simulation model to be used as an output.
Password	Letter and digit combination to access Twin Builder.
Petri net	Special form of a state machine.
Pipe	Data channel for the data transmission between Twin Builder modules.
Postprocessing	Data evaluation and processing after a simulation run has finished.
Preprocess	Specialized modules for information processing and parameter

Term	Description
	determination to define Twin Builder model components.
Preprocessor directive	Special commands for the SML compiler to include files, extract macros from the model library, etc.
Print colors	Colors used to print a system quantity from an active element or the ViewTool.
Project	Organizational structure containing all files and information belonging to a simulation task.
Qualifier	References a system quantity or parameter of a Twin Builder model component.
Quality criterion	Characteristic value of a system quantity used to determine the quality of an optimization run.
Queue	Display the active and all waiting simulation runs.
Recombination	Part of the genetic algorithm. It generates new individuals (parameter sets) by crossing of two parent individuals.
Roll back	Go back to a previous simulation step.
Sample time	Step size for digital controller.
Schematic	Twin Builder module for the graphical model definition.
Screen colors	Colors used to display a system quantity online during the simulation in an active element or the ViewTool.
Section	Part of the SML description containing model information for the Twin Builder sub simulators.
Selection	Part of the genetic algorithm. It selects individuals of the active generation to be transferred to the next generation by specific selection criteria.
Twin Builder settings	Contains the settings for the program environment, paths, etc.
Simulation backplane technology	Twin Builder software architecture for data exchange and program control of several simulators in one environment.
Simulation model	Graphical or text description of a real system using modeling capabilities of a simulation system.
Simulation parameter	Parameter used to control the simulation process.
Simulator	Software for the analysis of the behavior of a system using a simulation model.
Simulator coupling	Direct link of one or more simulators using the Twin Builder simulation backplane technology.

Term	Description
Simulator interface	Special software interface for the integration of external simulators into Twin Builder.
SML	Twin Builder Modeling Language.
.sml file	File containing the model description of a simulation run.
SML key word	Special terms used to determine the sections of an SML description.
Smooth	Analysis method in the DAY Post Processor.
Solver	Algorithm for the computation of model components without the capability to roll back steps.
.ssc file	Simplorer version 7 project file, containing all information about a project.
.ssh file	Simplorer version 7 schematic file, containing the graphical representation of a model and simulation data.
State	Basic element of state graphs that defines properties and activities in a certain system state.
State graph	Combination of states and transitions. A modeling language for discontinuous systems.
State Graph Module (SGM)	Twin Builder sub simulator to analyze simulation models described using state graphs basing on the PETRI net theory.
Status line	Displays the present state of a Twin Builder application.
Subsheet	A Twin Builder sheet embedded into another Twin Builder sheet, connected via pins, automatically creates a macro inside the .sml file.
Sub-simulator	Twin Builder internally coupled simulator.
Sub-skeleton	File containing rules for the creation of a non-Twin Builder description language.
Successive Approximation	Optimization method of the Experiment Tool with automatic parameter variation and target function determination.
Symbol editor	Twin Builder application for the creation or modification of a symbol.
Symbol level	Group of drawing elements of an animated symbol displayed together depending on the input value or the user activity on a interaction pad.
Symbols	Graphic representations of elements on the Schematic, placed from the model library. They can be modified using the symbol editor.
Synchronization	Update of a schematic from an older Twin Builder version to the latest symbols.
Synchronization	Detection of events in a state graph.
System quantity	Any quantity computed by the simulator.
System	Simulation level, where models from different physical domains are

Term	Description
simulation	simulated at the same time.
System variable	Predefined variables inside Twin Builder that cannot be used as a variable name or a specifier.
Task	Analysis to be performed as part of an experiment. It can contain several simulation runs.
Template	Predefined structure for a Twin Builder application.
TEND	Simulation parameter that determines the simulation end time.
Text Editor	Twin Builder application for the definition of models in SML language.
Time Function Module (TFM)	Twin Builder sub simulator for the computation of time dependent functions.
Time limited	A license of the simulation software for a certain (limited) time period.
Time step	Present time step size used to compute the next results vector.
Total fitness	The total fitness of an optimization run.
TR simulation	Transient analysis of a model.
TR simulator	Twin Builder simulator for TR analysis.
Transition	Cross over condition between the input and output state(s) of a state graph, defined by a logical expression.
Transition component	Basic structure of a state machine. It comprises a transition and all of its input and output states.
Trapezoid Formula	Numerical algorithm used inside Twin Builder.
Trend Analysis	Analysis method of the Experiment Tool with automatic parameter variation and characteristic value determination.
UDMinit	Section of the C/C++ interface for the model initialization.
UDMMain	Section of the C/C++ interface containing the model computation algorithm.
User defined component (UDC)	Model description for electrical components using a user defined C/C++ program, with direct access to the solver matrix.
User defined model (UDM)	Model description for nonlinear characteristics using a user defined C/C++ program.
User management	Saves the workspace for individual users.
Version report	Special mode inside the Twin Builder help system to create a detailed information file about the present Twin Builder installation.
VHDL	Very high-speed integrated circuit Hardware Description Language for digital

Term	Description
	systems.
VHDL-AMS	Very high-speed integrated circuit Hardware Description Language - Analog Mixed Signal. Extension of VHDL, a hardware description language for digital and analog systems.
VHDL-AMS simulator	Simulator integrated with Twin Builder's backplane that calculates simulation models described in VHDL-AMS.
Window elements	Windows in the workspace containing various information. They can be turned on or off by the user.
workflow	Graphic representation of a sequence of activities that are performed on a certain data set.
Worst Case Analysis	Analysis method of the Experiment Tool with parameter combination of all extreme values for the parameters and characteristic value determination.
YMAX	Maximum value used for display in the online graphic using the View tool.
YMIN	Minimum value used for display in the online graphic using the View tool.

Table of Components Libraries

Components Libraries	Description
Basic Elements	Provides electric circuit components, blocks, states, measuring devices, signal characteristics (functions to evaluate characteristics online during simulation), modeling tools (time functions, characteristics, equations), and components of physical domains.
VHDL-AMS	Provides a Basic Elements VHDLAMS library with common basic functionality, circuit components and blocks, and a Digital Elements library in VHDL with common basic functionality used for simple digital circuits.
Basic Elements VHDLAMS	Provides electric circuit components, blocks, measuring devices, modeling tools (Time functions), and components of several physical domains modeled in VHDL-AMS.
Digital Elements	Provides components such as counters, flip-flops, latches, gates, analog-to-digital and digital-to-analog converters, and digital sources used for simple digital circuits.
Aircraft Electrical VHDL-AMS	Provides components that are mostly first principle mathematical system-level models: • Basic

Components Libraries	Description
	• Distribution • Engine • Generator • Load
HEV VHDLAMS	Provides data and control, electrical, and mechanical components for • Conventional Vehicle (CV) • Electrical Vehicle (EV) • Hybrid Electrical Vehicle (HEV) • Hybrid Electrical Vehicle (HEV) with Permanent Magnet Synchronous Motor (PMSM)
Manufacturers	Provides libraries with semiconductor device level models of different manufacturers.
Power Management ICs	Provides real components from manufacturers, consisting of the following types of components: • High and Low Side Drivers • High Side Switches • Linear Regulators • Low Side Switches • Power Factor Correction • PWM Controllers • Three Phase Bridge Drivers
Power Semiconductors	Provides real characterized power devices for Si and SiC from the manufacturers.
Ultracapacitors	Provides ultracapacitors components from Maxwell Technology.
Multiphysics	• Hydraulic Components Library • Mechanical System Library • Power System Library • Sensors Library • Switch Mode Power Supply (SMPS) Library
Tools	Provides a library of components that calculate coordinate transformations, connect conservative nodes from different natures, and connect different data types. A compatibility library that ensures backwards compatibility of models from previous versions is also available.
VDALibs	Provides component models from the open source library created by the

Components Libraries	Description
VHDLAMS	VDA/FAT working group number 30. This group promotes the relationship between car manufacturers and their suppliers concerning simulation of mixed systems and model exchange.
Personal Libraries	Provides a location to insert user-defined libraries.
Project Components	Provides libraries included in an opened project.

Common Twin Builder Design Conventions

The topics in this section describe the conventions that you must observe when working with Twin Builder designs.

Names of Components and Variables

User-defined names can be given to components, blocks, states, time functions, characteristics, nodes, ports, and standard variables. Names may contain any combination of uppercase and lowercase letters (A-Z, a-z), the numerals 0 through 9, and underscores – and can have a maximum of 50 characters.

Note:

- In general, user-defined names are case-sensitive. However, names of components and variables of VHDL-AMS models are case-insensitive and all uppercase letters are converted to lowercase.
- Do not duplicate names (for example, **R1** and **r1**). Duplicate instance names result in netlist errors when attempting to compile a circuit prior to analysis.
- The first character of a name must always be a letter.
- Vowel mutations (e.g., umlauts) are not allowed.
- Spaces are not allowed.

The following predefined variable types are not allowed for names:

- SML notation keywords.
- Simulation parameters.
- System variables.

Parameter Qualifiers

Components are characterized by various physical quantities. A resistor, for example, is represented by current (I) and voltage (V) in the simulation. System variables may be accessed by reading (to use the actual quantity in an expression or to create an output) or writing (to influence quantities). Use the following syntax to access component variables:

ComponentName.Qualifier

Computations and outputs require access to system variables. The form and number of the qualifier depend on the corresponding component.

> **Note:**
>
> All qualifiers are case sensitive and must use capital letters (for example, **R.V**, *not***R.v**).

Qualifier Lists

The tables in this section display the most common Twin Builder qualifiers. The System Outputs table lists qualifiers that are read-only. The Component Parameters table lists qualifiers that can be both read and written.

System Outputs

Notation	Description
ComponentName.V	component voltage, node potential (read)
ComponentName.I	component current (read)
ComponentName.dV	derivative of the component voltage (read)
ComponentName.dI	derivative of the component current (read)
ComponentName.VAL	block/time function/characteristic output signal (read)
ComponentName.ST	yields the currently valid marking status of a state true-1-marked-active false-0-unmarked-inactive
ComponentName.P	component pressure, node pressure
ComponentName.Q	component flow rate
ComponentName.C	component hydraulic capacitance

Notation	Description
ComponentName.CHARGE	component charge
ComponentName.MMF	component magnetomotive force
ComponentName.FLUX	component magnetic flux
ComponentName.S	component displacement
ComponentName.F	component force
ComponentName.V	component velocity
ComponentName.PHI	component angle
ComponentName.TORQUE	component torque
ComponentName.OMEGA	component angular velocity
ComponentName.T	component temperature
ComponentName.H	component heat flow

Component Parameters

Notation	Description
ComponentName.EMF	component electromotive force
ComponentName.R	component resistance (read/write)
ComponentName.C	component capacitance (read/write)
ComponentName.L	component inductance (read/write)
ComponentName.G	component conductivity (read/write)
ComponentName.I0	component initial current (read/write only at simulation start)
ComponentName.V0	component initial voltage (read/write only at simulation start)
ComponentName.CTRL	control signal (read/write)
ComponentName.UL	upper limit (read/write)
ComponentName	lower limit (read/write)

Notation	Description
.LL	
ComponentName .FREQU	frequency of a function (read/write)
ComponentName .TPERIO	cycle duration of a function (read/write)
ComponentName .AMPL	amplitude of a function (read/write)
ComponentName .INPUT	block input signal (read/write)
ComponentName .TS	sampling time of the block sampling function (read/write)
ComponentName .QUANT	control quantity (read/write)
ComponentName .CH	component characteristic (read/write)
ComponentName .FILE	file name (read/write)
ComponentName .VALUE	source pressure, flow rate, magnetomotive force, magnetic flux, displacement, force, velocity, angle, torque, angular velocity, temperature, heat flow
ComponentName .K	component hydraulic conductance
ComponentName .VOL	component fluid volume
ComponentName .B	component bulk modulus
ComponentName .P0	component initial pressure (read/write only at simulation start)
ComponentName .RHO	component fluid density
ComponentName .DIA	component diameter
ComponentName .LEN	component length
ComponentName .Q0	component initial flow rate (read/write only at simulation start)

Notation	Description
ComponentName .K	component magnetic resistance
ComponentName .W	component number of turns
ComponentName .FLUX0	component initial magnetic flux (read/write only at simulation start)
ComponentName .C	component spring rate
ComponentName .DAMPING	component damping coefficient
ComponentName .M	component mass
ComponentName .S0	component initial position (read/write only at simulation start)
ComponentName .V0	component initial velocity (read/write only at simulation start)
ComponentName .SUL	component upper position limit
ComponentName .SLL	component lower position limit
ComponentName .J	component moment of inertia
ComponentName .PHI0	component initial angle (read/write only at simulation start)
ComponentName .OMEGA0	component initial angular velocity (read/write only at simulation start)
ComponentName .PHIUL	component upper angle limit
ComponentName .PHILL	component lower angle limit
ComponentName .K	component thermal resistance
ComponentName .C_TH	component thermal capacitance
ComponentName .T0	component initial temperature (read/write only at simulation start)

Parameter Types

The following table summarizes the parameter types used in Twin Builder.

Type	Description	Value Utilization	Accepted Value Format
Common Type (Name.X)	General parameters and quantities Example: E1.EMF	All expressions are interpreted with their actual value to define a parameter.	All numerical values (constant with or without unit suffix), simulation parameters, variables, component parameters, mathematical, or logical expressions. A logical expression provides only the value ‘1’ (TRUE) or ‘0’ (FALSE). Examples: 10k, 5K, -1E-3, 0.003, 20MEG Tend, Hmin, SECM.ITERAT var1, var2, _var3, var_4, Var_4 R23.I, C17.V, E4.EMF, GZ1.VAL 10*t+var1-INTEG(var2) delay>=2.5m*var1
Control Type (Name.CTRL)	Control input of switching devices. Example: S1.CTRL	The expression value tests if **>0**, **<0**, or **= 0**. See also **Common Type**.	All numerical values (constant with or without unit suffix), variables, component parameters, mathematical, or logical expressions. A logical expression provides only the value ‘1’ (TRUE) or ‘0’ (FALSE). Examples: See **Common Type**.
Initial Value Type (Name.X0)	General initial parameters and quantities. Example: IM1.IA10	Initial values are set only once at simulation start. See also **Common Type**.	All numerical values (constant with or without unit suffix), simulation parameters, variables, component parameters, mathematical, or logical expressions. A logical expression provides only the value ‘1’ (TRUE) or ‘0’ (FALSE). Examples: See **Common Type**.

Type	Description	Value Utilization	Accepted Value Format
Logical Expression Type	Parameters for switching conditions. Example: TRANS1.TRC	Logical expressions are interpreted as **TRUE** if the provided value is '1'. Otherwise, the value is **FALSE**.	Boolean expressions. Examples: (V1.EMF >=0), (AM1.I = -3), (RHYD1.Q < 2.4)
Quantity Type (Name.QUANT)	Control quantity of controlled sources. Examples: ExNL.CTRL IxNL.CTRL EPOLY.CTRL	All quantities are interpreted with their actual value to define a parameter.	The type accepts only voltage and current of voltmeters, ammeters, or wattmeters. The values cannot be assigned by a variable. Examples: VM1.V, AM2.I, WM3.V, WM3.I
Characteristic Type (Name.CH)	Characteristics of non-linear elements. Examples: RNL.CH EINL.CH	For a given X value, the Y value is determined. The LOOKUP function provides characteristic values in equations.	The type accepts only the output value of a characteristic component. The values cannot be assigned by a variable. Examples: EQUL1.VAL, XY1.VAL
File Type (Name.FILE)	File name for dataset-based characteristics. Example: XY1.FILE	For a given X value, the Y value is selected.	The type accepts only a name referring to an .mdx file (.xls 4.0, .csv, .txt, .mdb). The value cannot be assigned by a variable. Example: C:\release\diode.mdx

Note:

This table does not apply to user-defined models (UMODEL) if users choose to integrate values directly.

Predefined Variables

The simulator uses intrinsic variables for internal computation. All predefined variables are case insensitive.

> **Warning:**
>
> Do not use predefined variables for names (including port names) in a model description. If these variables are used in a model description, unexpected effects or an error message result.

System constants (read)	F, TIME, H, PI, TRUE, FALSE, SECM.ITERAT, FSTEP
General simulation parameters (read/write)	TEND, HMIN, HMAX, TEMP, FSTART, FEND

Predefined Constants

The simulator provides natural and mathematical constants that can be used in mathematical expressions within component dialog boxes or SML descriptions.

The following table shows the available constants and their corresponding symbols:

Constant	Legacy Name[1]	Value	Unit	Description	Symbol
pi	MATH_PI	3.1415926535898	[/]	Pi	π
mathE	MATH_E	2.718281828	[/]	Euler number	E
e0	PHYS_E	8.85419 10^{-12}	$C^2 \cdot Jm$	Permittivity of vacuum	ε_0
u0	PHYS_MU0	1.25664 10^{-06}	T^2m^3/J	Permeability of vacuum	μ_0
boltz	PHYS_K	1.38066 10^{-23}	J/K	Boltzmann constant	k_B
elecq	PHYS_Q	1.60217733 10^{-19}	C	Elementary charge	e
c0	PHYS_C	299792458	m/s	Speed of light	c
g0	PHYS_G	9.80665	m/s^2	Acceleration due to gravity	g
planck	PHYS_H	6.6260755 10^{-34}	Js	Planck constant	h
abs0	PHYS_T0	-273.15	°C	Absolute Zero	ϑ
eta	N/A	376.730313461	Ohm	Impedance of vacuum	η

1. For backward compatibility, #defines of these names will be replaced by their values during translation of a model definition - unless the #define line is still present in the model definition.

Equations, Expressions, and Variables

Equations consist of *operands* and *operators*. An *operand* can be any number or variable name. An *operator* compares or assigns a value.

In *expressions* you can create and use *variables* as often as you want. A *variable* is defined when the *variable* name is used in an *expression* or for a parameter value within a component dialog box. You do not need to define the *variable* in a specific assignment unless you want it to have a defined initial value. For example, in the following equation:

Z:=Y+X

- **X**, **Y**, and **Z** are the *operands*
- **:=** and **+** are the *operators*.

If *operands* are complex numbers (for example, in an AC simulation), the comparison *operators* (**<**, **>**, **<=**, **>=**) consider only the real part.

Operators

Assignment operators	:=	Assignment
	##	Delay operator combined with the action type DEL
	$$	Schedule operator combined with the action type SCHED
Arithmetic operators	*	Multiplication
	/	Division
	+	Addition
	–	Subtraction
	**	Power 7**4 = 7^4 = 2401
Comparison operators without synchronization	<	Less than
	>	Greater than
	!=	Not equal to
Comparison operators with synchronization	This operator type forces the simulator to synchronize on the condition with the minimum step width.	
	<=	Less than or equal to

	>=	Greater than or equal to
	==	Equal to (compares two values for equality)
Logic operators (must use a space before and after the operator)	&&	Logical AND (conjunction)
	||	Logical OR (disjunction)
	!	Logical NOT (negation)

Note:

For backward compatibility, the operators: **AND**, **OR**, **NOT**, **<>**, **><**, **=** are still recognized in pure SML text.

Standard Mathematical Functions

Mathematical functions consist of the function name and one or two arguments. An argument can be any number or variable name. A mathematical function applies the function, which it represents, to the arguments.

r:=FCT(x,y),r:=FCT(z)

In the example above:

- **x**, **y**, and **z** are arguments.
- **z** is a complex number.
- **FCT** is the function name.
- **r** is the result.

If the arguments are complex numbers (for example, in an AC simulation), the functions RAD, DEG, DEGEL, INT, REM, and LOOKUP consider only the real part.

Warning:

When entering these functions, do not leave spaces between the function arguments and the open parenthesis mark. For example: **SIN(x)** *not* **SIN (x)**.

Note:

When defining arguments for trigonometric functions, you must consider poles to avoid potential errors during a simulation.

Notation	Description	Example
SIN(x)	Sine, x[rad]	SIN(PI/6)=0.5
COS(x)	Cosine, x[rad]	COS(2•PI/3)=-0.5
TAN(x)	Tangent, x[rad]	TAN(PI/4)=1
ASIN(x)	Arc sine [rad]	ASIN(0.5)=0.524=PI/6
ACOS(x)	Arc cosine [rad]	ACOS(0.5)=1.0471=PI/3
ATAN(x)	Arc tangent [rad]	ATAN(1)=0.785=PI/4
ATAN2(x,y) ATAN2=ATAN(y/x)	Arc tangent2 [rad] r=0 if x=0 and y=0; $-\pi \leq r \leq \pi$	ATAN2(.25,1)= ATAN(4)=1.325
SINH(x)	Hyperbolic sine	SINH(1)=1.175201
COSH(x)	Hyperbolic cosine	COSH(1)=1.54308
TANH(x)	Hyperbolic tangent	TANH(1)=0.761594
ASINH	Hyperbolic Arcsine	ASINH(1.175201)=1
ACOSH	Hyperbolic Arccosine	ACOSH(1.54308)=1
ATANH	Hyperbolic Arctangent	ATANH(0.761594)=1

Arithmetic Functions

Notation	Description	Example
SQR(x)	Square.	SQR(16)=16^2=256
SQRT(x)	Square root.	SQRT(9)=$\sqrt[2]{9}$=3
ROOT (x,y)	n-th Root.	ROOT(27,3)=$\sqrt[3]{27}$=3
SDT(x [,y]), y=0	Integration of a variable from the function call until to the simulation end. y=integration method	SDT(var1)=$\int$var1 dt **Integration methods:** 0=Euler (explicit) 1=Euler (implicit) 2=Trapezoidal (implicit) 3=Auto mode (setting taken from transient options dialog box)
DDT(x)	Derivative of a variable in time	DDT(var1)=dvar1/dt
MAX (x1,x2)	Returns the greater of two values	MAX(1,5)=5
MIN	Returns the lesser of two values	MIN(1,5)=1

Notation	Description	Example
(x1,x2)		

Exponential Functions

Notation	Description	Example
EXP(x)	Exponential function.	EXP(5)=e^5=148.41
LN(x)	Natural logarithm.	LN(3)=$\log_e 3$=1.099
LOG(x[,y]); y=10	Common logarithm.	LOG(7,4)=$\log_4 7$=1.403

Complex Functions

Notation	Description	Example
ABS(x)	Absolute value.	ABS(-8.5)=|-8.5|=8.5
RE(z)	Real part	RE(z)=5
IM(z)	Imaginary part	IM(z)=3
ARG(z)	Argument of a complex number in radians. For example, given: $z=a+bi=r(\cos\varphi+i\cdot\sin\varphi)=r\cdot e^{i\varphi}$ $z=5+3i=5.83(\cos 30.96°+i\cdot\sin 30.96°)$	ARG(z)=0.53

Conversion Functions

Notation	Description	Example
ANG_RAD(x)	Conversion from degrees to radians.	ANG_RAD (30)=PI/6=0.524
ANG_DEG(x)	Conversion from radians to degrees.	ANG_DEG(PI/2)=90°
DEGEL(x[,y]); y=1	Conversion from degrees electrical to seconds with respect to Hz.	DEGEL (180,50)=10ms

Rounding Functions

Notation	Description	Example
SGN(x)	Sign dependent value (-1, 0, 1). r=0 if z=0, 1 if Re(z)>0 or (Re(z)=0 and Im(z)>0), -1 otherwise.	SGN(3)=1; SIGN (0)=0; SIGN(-3)=-1
MOD(x,y)	Modulus.	MOD(370,60)=10
INT(x)	Integer part of a value.	INT(2.5)=2
REM(x)	Fractional part (remainder) of a decimal number, such that rem(x) = x-int(x).	REM(2.5)=0.5
LOOKUP(x,y) x=Characteristic name y=X value	Access function to a characteristic.	LOOKUP (XY1.VAL,5)= Y value of the characteristic XY1 for the X value 5

Conditional Functions

Notation	Description	Example

IF (condition1)
{ var:=1; }
[ELSE IF (condition2)
{ var:=2; }
ELSE
{ var:=3; }]
	IF-ELSE returns equations (which are valid for FML and FML_INIT components).
	If-Else function to perform operations dependent on conditions.
	The ELSE IF and ELSE statements can be omitted.
		IF (t>=1)
		{ var:=1; }
		[ELSE IF (t>=2)
		{ var:=2; }
		ELSE
		{ var:=3; }

IF (condition, True-expression, False-expression)	The implicit IF statement is an expression and can only be used on the right side of an equation or in value expressions.	

Unit Suffixes of Numeric Data

Enter numeric data in component dialog boxes and in Twin Builder‘s editors, using the following unit extensions:

Suffix	Value		SML		Examples
tera	10^{12}	E12	t	TER	*5e12, 5t, 5ter*
giga	10^{9}	E9	g	GIG	*1.4e9, 1.4g, 1.4gig*

Suffix	Value		SML		Examples
mega	10^6	E6		MEG	*-1.4E6, -0.3meg, -0.3MEG*
kilo	10^3	E3	k	KIL	*1000, 1e3, 1k, 1kil*
milli	10^{-3}	E-3	m	MIL	*0.0105, 1.05E-2, 10.5M, 10.5MIL*
micro	10^{-6}	E-6	u	MIC	*0.000005, 5e-6, 5u, 5mic*
nano	10^{-9}	E-9	n	NAN	*40E-9, 40n, 40nan*
pico	10^{-12}	E-12	p	PIC	*100E-12, 100P, 100PIC*
femto	10^{-15}	E-15	f	FEM	*9E-15, 9F, 9FEM*

Note:

- The **comma** is reserved for separating parameters in lists.
- The **period** (**dot**) is reserved as a decimal point.
- "**M**" is interpreted as 10^{-3}, *not* as 10^6.

SI Units

All units used in Twin Builder are derived from the SI system of units.

Quantity	Unit Name	Symbol
Length	Meters	m
Mass	Kilograms	kg
Time	Seconds	s
Electrical current intensity	Amperes	A
Temperature	Kelvins	K
Voltage (derived SI unit)	Volts	V

However, the unit handling feature of Twin Builder allows you to use non-SI units for designs. Non-SI units are automatically converted to the expected SI units.

Unit Handling

The unit handling feature of Twin Builder allows you to enter component parameter values in multiples of standard SI units such as *millivolts*, *nanoamperes*, and *kilometers*, as well as in non-SI units such as *pounds-per-square-inch*, *degrees Fahrenheit*, and *feet*. This eliminates the

need for error-prone unit conversions and reduces the time needed to calculate component parameters.

Each component parameter that is a physical quantity can be assigned an expected unit of measure, which is the unit of measure for the parameter value used during simulation.

> **Note:**
>
> - Twin Builder *internal* models have predefined expected units for each physical parameter.
> - The parameters of *user-defined* models such as C-Models can be assigned expected units when the user-defined models are created.

Each parameter also is provided with a set of additional units that can be applied to the same physical quantity.

For example, Twin Builder's force source component has an expected unit of *Newtons* and an associated set of additional units that includes *milli-Newtons*, *PoundsForce*, and *dynes*.

When an instance of a component such as the force source is placed on a schematic, its parameter values can be edited in its **Properties** dialog box. The units associated with the parameters also can be changed. The units are located in combo boxes next to the text fields containing each parameter's value. Changing the unit for a parameter is as simple as selecting a new unit in the combo box for the parameter's unit.

When the component is simulated, Twin Builder automatically converts quantities expressed in the newly chosen units to equivalent quantities of the expected units. Thus, you can express component parameters in units from different systems of measurement without affecting the accuracy of the simulation results.

Network Configurations

In Twin Builder only ammeters are allowed as controlling components for current controlled elements. These must be inserted properly in the controlling branch. If sources are part of mutual controlling sources in the circuit, stability problems may occur if the total gain of the loop is greater than or equal to one.

The following types of network configurations are invalid:

- Series connection of ideal current sources
- Series connection of inductors and ideal current sources
- Series connection of inductors with different initial values of current, $I0_1 \neq I0_2$
- Series connection of an inductor with an initial current value and an opened ideal switch or nonconducting system level semiconductor
- Parallel connection of ideal voltage sources
- Parallel connection of capacitors with ideal voltage sources

- Parallel connection of capacitors with different initial values of voltage, $V0_1 \neq V0_2$
- Meshes which consist only of ideal sources (short-circuit)
- Open-ended branches

Actions in States

Action	Description	Syntax in Value
CALC	The variable is calculated at each simulation step and each transition from one state to another.	*var1***:=***100*t*
STEP	The variable is calculated at each valid simulation step.	*var2***:=***2*t*
CATI	The variable is calculated outside the state graph and before the calculation of the electric circuit.	*var3***:=***sqrt(t)*
SET	The variable is calculated only once at the moment of activation of the state.	*var4***:=***2.5*
DEL	Sets a delay. The variable is set to false at the moment of activation and set to true after the delay time.	var##time [s] *delation****##****10m*
DELRES	Deletes a defined delay variable.	del4
DIS	The variable value (and moment) is displayed in the simulator status window.	Name.Qualifier *dc.n*
TXT	The given text string is displayed in the simulator status window.	“Text String” *“State waiting”*
KEY	Sets a mark in the state graph by pressing a key.	**<***A***>**
STOP	Interrupts the simulation (can be continued).	No Parameters
BREAK	Finishes the simulation.	No Parameters
SAVE	Saves the active simulation status in a status file.	No Parameters

Basic Rules for Specifying Time Steps

Correct simulation processing and results depend on the proper choice of minimum and maximum values for the integration step size. The smaller the maximum integration step size, the more correct the results, but the longer the processing time.

This means that when specifying these minimum and maximum time step values, there must be a compromise between accuracy and simulation time. The basic rule of measurement “Not as precise as possible, but as precise as required” is also valid for a simulation. The following guidelines should help with the proper of integration step width:

Model properties	**Recommended**
What is the smallest time constant (τ_{min}) of the electric circuit (R*C or L/R) or of the block diagram (PTn-elements)	$Hmin < \frac{\tau_{min}}{10}$
What is the largest time constant (τ_{max}) of the electric circuit (R*C or L/R) or of the block diagram (PTn-elements)	$Hmax < \frac{\tau_{max}}{10}$
Which is the smallest cycle (T_{min}) of oscillations that can be expected (natural frequencies of the system or oscillating time functions)	$Hmin < \frac{T_{min}}{20}$
Which is the largest cycle (T_{max}) of oscillations that can be expected (natural frequencies of the system or oscillating time functions)	$Hmax < \frac{T_{max}}{20}$
What is the smallest controller sampling (TS_{min})	$Hmin < \frac{TS_{min}}{5}$ $Hmax = TS_{min}$
What is the fastest transient occurrence (TU_{min}) (edge changes of time functions)	$Hmin < \frac{TU_{min}}{20}$
What is the time interval to be simulated (Tend)	$Hmax < \frac{TEND}{50}$

- Select the smallest of each estimated maximum and minimum time step for the simulation model.
- All recommended values are based on numeric requirements and experience and do not guarantee a successful simulation. Please consider the algorithm as a guideline.
- In case of doubt, decrease the maximum and minimum step size by dividing by 10, repeat the simulation and compare the results. If the second set of results (with the step size decreased) shows conformity with the first results, then the step sizes chosen for the first simulation were appropriate (remember that smaller values increase the simulation time).

> **Note:**
>
> If the number of iterations is identical with the defined maximum value (Maximum Number of Iterations, or Iteratmax) during the simulation, the model may be incorrect. The simulation monitor displays the actual number of iterations for each step of the simulation.

Troubleshooting

This section contains the following information:

Modeling

- **Project must be copied before you can work with it**

  Please remove write protection from the files.

- **Cannot connect elements on a sheet**

  Twin Builder schematic checks that the types of two pins to be connected are correct, e.g.,. it is not possible to connect an electrical pin to a signal pin. So please check that the proper pin types are being connected. If the connection is still impossible, please place the same element again.

- **Connections are incorrect**

  Select the improperly connected wires, click the right mouse button and select **«Disconnect»** from the pop-up menu.

- **SML syntax errors**

  Using a comma as a decimal point is not allowed.

  Logic operators must be surrounded by spaces.

Predefined variables cannot be used for names in a model description.

When using functions, there should be no space between the function argument and the open parenthesis mark.

- **Modeling errors**

  Each independent circuit has to be connected to the ground node at least once. Only electrically reasonable circuits lead to correct simulation results. Voltage sources, current sources and switches are ideal components.

Display and Simulation

- **No graphic is displayed**

  It is possible that there are no outputs defined for a component (right-click a schematic sheet and select **Output Dialog**) or no values were selected for display in a report.

- **Simulation does not start**

  Check the simulation queue (**Tools** > **Show Queued Simulations**) to see all active jobs. All simulations are placed in this queue and processed according the start time.

- **Unexpected simulation results**

  If the controlled sources are part of closed control loops in the circuit, stability problems can occur if the total gain of the loop is greater than or equal to one. In this case, (linear or nonlinear) controlled sources which directly access variables from the set or circuit equations should be used.

- **The simulator computes always with the maximum step width (HMAX)**

  If the simulation model only consist of blocks, the block diagram is computed (to each time step) with the maximum step width (HMAX).

Literature References

- The examples used in this document were derived from benchmarks developed by the working group 'VHDL-AMS' of the MSR project. The MSR Consortium was founded by German Car Manufacturers and Suppliers to define and establish standards for information/data exchange between manufacturer and supplier. MSR has granted permission to use the examples found in this tutorial. More information on the MSR Consortium and their activities can be found at http://www.msr-wg.de/msr.html.
- Web site of IEEE 1076.1 Working Group at http://www.eda-twiki.org/vhdl-ams-old/wwwpages/welcome.htm.
- Vahe Caliskan: *Modeling and Simulation of a Claw-Pole Alternator, Detailed and Averaged Models*, LEES Technical Report TR-00-009, Laboratory for Electromagnetic

and Electronic Systems, Massachusetts Institute of Technology, Cambridge, MA, October 2000

- Woodson and Melcher: *Electromechanical Dynamics, Part 1*, 1968
- Bai, H.; Pekarek, S.; Techenor, J.; Eversman, W.; Buening D.; Holbrook, G.; Hull, M.; Krefta, R.; Shields, S.: *Analytical Derivation of a Coupled-Circuit Model of a Claw-Pole Alternator with Concentrated Stator Winding*, IEEE Transaction on Energy Conversion, March 2002, pp 32-38

Index

W