How to Contact Reactive Systems:

**Web**  reactive-systems.com

**Email**  
help@reactive-systems.com  (Technical support)
sales@reactive-systems.com  (Sales)
info@reactive-systems.com  (General inquiries)

**Phone**  (+1) 919-324-3507

**Fax**  (+1) 919-324-3508

**Mail**  Reactive Systems, Inc.
341 Kilmayne Dr.
Suite 101
Cary, NC 27511
USA

---

Reactis User’s Guide

Copyright © 2000–2020 Reactive Systems, Inc. All rights reserved.

This document and the Reactis software are provided under the terms of the Reactis Software License Agreement. Tomorrow's Software Today and Reactis are registered trademarks of Reactive Systems, Inc. MATLAB, Simulink and Stateflow are registered trademarks of The MathWorks, Inc. TargetLink is a registered trademark of dSPACE GmbH. Other names are trademarks or registered trademarks of their respective holders.
Contents

1 Introduction .......................... 1

2 Installing and Starting Reactis .......................... 3
   2.1 Installing with Local License File ......................... 3
   2.2 Installing with Remote License Manager .................. 5
   2.3 Installing the Reactis License Manager ................. 5
   2.4 Performing a Silent Install ......................... 5
   2.5 Starting Reactis ......................... 7

3 Getting Started with Reactis .......................... 9
   3.1 A Note on Model Preparation ......................... 9
   3.2 Reactis Top Level .................................... 10
   3.3 The Info File Editor .................................... 12
   3.4 Simulator ............................................ 17
      3.4.1 Generating Random Inputs ................. 17
      3.4.2 Tracking Model Coverage ................. 18
      3.4.3 Reading Inputs from Tests ......................... 21
      3.4.4 Tracking Values of Data Items  .......... 22
      3.4.5 Querying the User for Inputs ................. 25
      3.4.6 Constructing a Functional Test with User Guided Simulation 26
      3.4.7 Other Simulator Features ................. 29
   3.5 Tester ......................... 29
   3.6 The Test-Suite Browser ......................... 31
   3.7 Validator ............................................ 33
      3.7.1 Manipulating Validator Objectives ................. 34
      3.7.2 Launching Validator ......................... 40

4 The Reactis Top-Level Window .......................... 41
   4.1 Labeled Window Items ......................... 42
   4.2 Menus ............................................ 46
   4.3 List of Matching From/Goto or Data Store Blocks ................. 51
   4.4 Configuration Variable Panel ......................... 51
   4.5 Extracting Subsystems ......................... 52
   4.6 Unpacking Top-Level Buses ......................... 53
   4.7 Printing Models ......................... 54
   4.8 Reactis Global Settings ......................... 55
      4.8.1 General Settings ......................... 56
# CONTENTS

4.8.2 Reactis for C Global Settings ........................................ 57
4.8.3 Reactis for EML Global Settings .................................... 57
4.8.4 MATLAB Global Settings ............................................. 57
4.8.5 Path Global Settings ................................................ 58
4.8.6 Files Global Settings ............................................... 60
4.8.7 User Info Settings .................................................. 60
4.8.8 License Settings .................................................... 60
4.9 Default Model-Specific Settings ........................................ 63

5 The Reactis Info File Editor .............................................. 65
5.1 Menus ................................................................. 66
5.2 Toolbar ............................................................... 68
5.3 Inport Types Pane ..................................................... 69
5.3.1 The Type Editor Dialog ................................................ 70
5.3.2 Type Constraint Syntax ............................................... 77
5.4 Configuration Variables ............................................... 79
5.4.1 Configuration Variable Sets ........................................... 81
5.5 Test Points .......................................................... 82
5.5.1 Adding and Removing Test Points .................................... 83
5.5.2 Importing Test Points from a Simulink model .................... 83
5.5.3 The Test Points Pane ................................................ 84
5.5.4 Configuring a Test Point with the Test Point Properties Editor ..................................................... 85
5.6 Outport Tolerances ..................................................... 88
5.6.1 Tolerance Methods ................................................... 89
5.7 The Tolerance Editor Dialog ............................................. 89
5.8 General .............................................................. 90
5.9 Error Checking ....................................................... 92
5.10 Coverage Metrics ..................................................... 97
5.11 Excluded Coverage Targets .......................................... 98
5.12 Validator Objectives ................................................ 99
5.13 C Code ............................................................. 100
5.14 External EML Functions .............................................. 100
5.15 Callbacks .......................................................... 100
5.16 Search Path .......................................................... 101
5.17 File Dependencies ................................................... 101

6 Reactis Coverage Metrics .................................................. 103
6.1 Simulink-Specific Metrics .............................................. 103
6.1.1 Conditional Subsystem Coverage .................................... 103
6.1.2 Branch Coverage .................................................. 103
6.1.3 Lookup Table Coverage ............................................. 108
6.2 Stateflow-Specific Metrics ............................................. 110
6.2.1 State Coverage .................................................... 111
6.2.2 Condition Action Coverage ........................................ 111
6.2.3 Transition Action Coverage ......................................... 111
6.2.4 Child State Exit via Parent Transition ............................ 112
6.3 Generic Metrics ....................................................... 113
6.3.1 Decision, Condition, MC/DC and MCC Metrics ................. 113
CONTENTS

6.3.2 Boundary Value Coverage ........................................... 116
6.4 Validator-Related Targets ............................................. 120
6.5 Excluding Coverage Targets ........................................... 121
   6.5.1 Disabling a Coverage Metric ..................................... 121
   6.5.2 Excluding a Subsystem ........................................... 122
   6.5.3 Excluding Individual Targets ................................... 122
6.6 Cumulative Subsystem Coverage ..................................... 125

7 Reactis Simulator ......................................................... 127
   7.1 Labeled Window Items ............................................... 127
   7.2 Menus ..................................................................... 132
   7.3 Creating Test Execution Reports ................................... 135
      7.3.1 Test Data Plots .................................................. 139
   7.4 Specifying the Simulation Input Mode ............................... 139
      7.4.1 User Input Mode ............................................... 141
      7.4.2 Test Input Mode ............................................... 149
   7.5 Tracking Data-Item Values .......................................... 149
      7.5.1 Selecting Bus Components ...................................... 150
      7.5.2 The Watched-Variable List .................................... 151
      7.5.3 Scopes ................................................................ 152
      7.5.4 Distribution Scopes ............................................. 154
      7.5.5 Difference Scopes ............................................... 154
   7.6 Tracking Model Coverage ............................................ 154
      7.6.1 The Coverage Summary Dialog ................................. 156
      7.6.2 Coverage Information in the Main Panel ....................... 156
      7.6.3 The Coverage Report Browser .................................. 159
   7.7 Exporting and Importing Test Suites ............................... 160
      7.7.1 Exporting Test Suites ........................................... 160
      7.7.2 Importing Test Suites .......................................... 162
   7.8 Updating Test Suites .................................................. 164
      7.8.1 Update Using Reactis ........................................... 164
      7.8.2 Update Outputs Using Simulink ............................... 165
   7.9 Model Highlighting ..................................................... 166

8 Reactis Tester ................................................................. 169
   8.1 The Tester Launch Dialog ............................................. 169
   8.2 The Progress Dialog ................................................... 172

9 Reactis Validator ............................................................ 175
   9.1 The Meaning of Validator Objectives ............................... 176
      9.1.1 Assertions ......................................................... 176
      9.1.2 User-Defined Coverage Targets ............................... 177
      9.1.3 Virtual Sources .................................................. 177
   9.2 Use Cases of Validator Objectives ................................... 177
      9.2.1 Checking a Requirement with an Expression Assertion and an Expression User-Defined Target ........................................... 177
      9.2.2 Checking a Requirement with a Diagram Assertion ......................... 178
      9.2.3 Creating a Functional Test Using a Virtual Source ......................... 179
9.3 Adding, Editing, and Removing Objectives ................................................. 180
  9.3.1 The Simulink Expression Objective Dialog ........................................ 181
  9.3.2 The Stateflow Expression Objective Dialog ....................................... 183
  9.3.3 The Timer Objective Dialog ............................................................... 185
  9.3.4 The Diagram Objective Dialog ............................................................ 187
  9.3.5 Accessing Values from Previous Simulation Steps .............................. 189
  9.3.6 Wiring Validator Objectives Within the Reactis Main Panel ................... 189
9.4 Linking to Requirements Documents ......................................................... 190
9.5 Running Reactis Validator ........................................................................... 193
  9.5.1 Labeled Window Items ......................................................................... 193
9.6 Validator Menus in the Reactis Top-Level Window ...................................... 197
9.7 Tracking Coverage Within Validator Objectives ........................................ 197

10 The Reactis Coverage-Report Browser ......................................................... 199
  10.1 Labeled Window Items ............................................................................ 200
  10.2 Menus ...................................................................................................... 201
  10.3 Exporting Coverage Reports .................................................................... 201
  10.4 Coverage Report Contents ...................................................................... 203

11 The Reactis Test-Suite Browser ..................................................................... 207
  11.1 Labeled Window Items ............................................................................ 207
  11.2 Menus ...................................................................................................... 210
  11.3 Test Step Filter Editor .............................................................................. 212

12 Executing Test Suites .................................................................................... 215
  12.1 Executing Test Suites Using Reactis Simulator ........................................ 215
  12.2 Executing Test Suites in Simulink ............................................................ 216
    12.2.1 Executing Tests in Simulink with the rsRunTests Utility ..................... 217
    12.2.2 Executing Tests in Simulink with the Simulation Parameters Dialog .. 219
    12.2.3 Running Tests in Simulink by Editing the .m file ............................... 219
    12.2.4 Running Tests in Simulink by Replacing Top-Level Inports with From-
        Workspace Blocks. .................................................................................. 219
  12.3 Executing Test Suites on Source Code ...................................................... 220

13 Maximizing Coverage ..................................................................................... 223
  13.1 Constraining Input Values ....................................................................... 223
  13.2 Modifying Test-Generation Parameters .................................................. 224
  13.3 Using Timer Targets ................................................................................ 224
  13.4 Interactively Tuning Test Suites .............................................................. 225
  13.5 Model Decomposition .............................................................................. 226

14 Preparing Models for Use with Reactis .......................................................... 227
  14.1 MATLAB .................................................................................................. 227
    14.1.1 Workspace Data Items ..................................................................... 227
    14.1.2 Unsupported MATLAB Features ...................................................... 228
  14.2 Simulink ................................................................................................... 229
    14.2.1 S-Functions ....................................................................................... 229
    14.2.2 Lookup Tables .................................................................................. 230
14.2.3 Table of Supported Blocks ................................................. 234
14.2.4 Simulink Extras ............................................................. 241
14.2.5 TargetLink 2.2.1 Library Blocks ......................................... 241
14.2.6 TargetLink 2.3.1 Library Blocks ......................................... 242
14.2.7 TargetLink 3.0 to 3.4 Library Blocks ..................................... 243
14.2.8 TargetLink 3.5 to 4.2 Library Blocks ..................................... 244
14.3 Stateflow ................................................................. 244
14.4 Embedded MATLAB ......................................................... 245

15 Model Semantics ............................................................. 247
15.1 Conditional Input Branch Execution ....................................... 247

16 Reactis for C Plugin ............................................................ 251
16.1 Quick Start ................................................................. 252
16.2 Configuring Reactis for C ................................................... 252
16.2.1 Global Settings ............................................................ 252
16.2.2 Model-Specific Settings ................................................... 253
16.3 S-Function C Code .......................................................... 258
16.3.1 Source Files .............................................................. 258
16.3.2 Include Search Path ....................................................... 260
16.3.3 Libraries ................................................................. 261
16.3.4 Defines ................................................................. 262
16.3.5 General ................................................................. 262
16.3.6 Pathnames .............................................................. 263
16.3.7 S-Function Requirements ................................................. 264
16.4 Stateflow Custom C Code ................................................... 264
16.4.1 Stateflow Custom Code Requirements .................................. 266
16.5 C Code Coverage Metrics .................................................... 266
16.5.1 Visualizing Coverage in C Code ......................................... 267
16.6 C Code in the Main Reactis Window ....................................... 267
16.7 C Code and Reactis Simulator ............................................... 267
16.8 C Code and Reactis Tester ................................................... 271
16.9 C Code and Reactis Validator ............................................... 272
16.10 Integer Overflows .......................................................... 272
16.11 Memory Errors ............................................................ 273
16.11.1 Invalid Pointer Creation ................................................. 274
16.12 Other Runtime Errors ....................................................... 274
16.13 Stub File Generation ......................................................... 275
16.14 Unsupported C Features ..................................................... 275
16.15 Supported C library functions .............................................. 277

17 Reactis for EML Plugin .......................................................... 279
17.1 Enabling the Reactis for EML Plugin ....................................... 280
17.2 External EML Functions ...................................................... 281
17.3 EML Coverage Metrics ....................................................... 281
17.3.1 Tracking Coverage in EML Code ....................................... 282
17.3.2 Debugging EML Code ..................................................... 283
17.4 Subset of Embedded MATLAB Supported by Reactis .................. 284
CONTENTS

17.4.1 Syntax ........................................... 284
17.4.2 Types .......................................... 285
17.4.3 Control flow ..................................... 285
17.4.4 User-defined functions, Subfunctions, and External Functions ............ 285
17.4.5 Array indexing .................................. 285
17.4.6 Logical operators ................................ 285
17.4.7 Relational operators ............................ 285
17.4.8 Math operators .................................. 285
17.4.9 Math functions .................................. 286
17.4.10 Statistical functions ............................ 286
17.4.11 Matrix operations ............................... 286
17.4.12 Element-wise array operations ................. 286
17.4.13 Vector operations ............................... 286
17.4.14 Bit operation ................................... 286
17.4.15 Cast operations ................................ 286
17.4.16 String operations ............................... 287
17.4.17 Miscellaneous functions ....................... 287

18 The Reactis API ..................................... 289
18.1 Using the Reactis API from MATLAB .................. 289
18.2 Using the Reactis API from C Programs ............... 289
  18.2.1 Quick Start ..................................... 289
  18.2.2 Compiling a C or C++ program with the Reactis API .................... 290
  18.2.3 Reactis API Files for C Programming .................. 291

19 The Reactis License Manager .......................... 293

A File Types and Extensions ............................. 295
  A.1 File Types Used or Created by Reactis .................. 295
  A.2 File Types Used or Created by the Reactis for C Plugin ............... 296

B Frequently Asked Questions ............................ 297

C Revision History ....................................... 299
  C.1 Patches Mailing List Archive ........................ 300
  C.2 V2020 (17 July 2020) ................................ 300
    C.2.1 Newly Supported Simulink Features ................. 300
    C.2.2 Other improvements ................................ 300
    C.2.3 Reactis for C Plugin .............................. 300
    C.2.4 Reactis for EML Plugin ............................ 300
  C.3 Previous Major Release Dates ........................ 301
Chapter 1

Introduction

Reactis® offers model-based testing, debugging, and validation for Simulink® / Stateflow® models. Reactis currently consists of three main components: Reactis Tester, Reactis Simulator, and Reactis Validator. An optional add-on product Reactis for C Plugin offers white-box analysis of C code that is incorporated into a model using S-Functions or Stateflow custom C code.

Reactis Tester automatically generates test suites from Simulink® / Stateflow® models of embedded control software. The test suites provide comprehensive yet concise coverage of different test-quality metrics. Each test in a test suite consists of a sequence of input vectors as well as the responses to those inputs generated by the model. These tests may be used for a variety of purposes, including:

**Implementation conformance.** The tests may be applied to implementations derived from models to ensure conformance with model behavior.

**Model testing and debugging.** The tests may be run on the models themselves to analyze model behavior and to detect runtime errors.

**Regression testing.** The tests may be run on a new version of a model to compare its behavior to an older version.

**Reverse engineering of models from source.** Tests may be generated from models derived from legacy code in order to check conformance between model and code.

Reactis enables you to maximize the effectiveness of your testing while reducing time and effort.

Reactis Simulator enables you to visualize model execution. Simulator’s user interface is similar to those of traditional debuggers from programming languages: it allows you to step through the execution of models by hand, set break points, and study values of intermediate data elements. By executing Tester-generated tests in Simulator and examining coverage, you can often identify modeling errors, redundant code, or dead code. Simulator also supports reverse execution and the fine-tuning of Tester-generated test suites.

Reactis Validator performs automated searches of models for violations of user-specified requirements. If Validator finds a violation of a requirement, it returns a test highlighting the problem. This test may then be executed in Reactis Simulator to gain an understanding of the

---

1 Reactis is a registered trademark of Reactive Systems, Inc.
2 Simulink, Stateflow and MATLAB® are registered trademarks of The MathWorks, Inc.
source of the error. Validator enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews. Some checks that may be performed with Validator include the following.

- Will a car’s anti-lock brakes always engage within a specified time period when the brake pedal is pressed?
- Will a plane’s thrust reversers ever engage while the aircraft is airborne?
- Will an x-ray machine ever deliver a dangerous dose of radiation?
- Will a cellular phone “hang” when moved from a non-serviced into a serviced area?

**Support and Feedback**
RSI welcomes user feedback and questions. To ask questions, make suggestions, or report suspected bugs, you may call RSI’s help line at (+1) 919-324-3507 or send e-mail to:

help@reactive-systems.com

When sending e-mail, you are encouraged to include the “System Info” information for your Reactis installation; this can be obtained by selecting the Help → About menu item from the top-level Reactis window, then clicking Copy To Clipboard and pasting the information into the e-mail message.
Chapter 2

Installing and Starting Reactis

Reactis is available on Windows® 10. To run Reactis, a system should satisfy the following minimum requirements:

- At least 1024 MB RAM (more is required for large models)
- At least 160 MB free disk space
- MATLAB, Simulink, and Stateflow installed
- An installed Ethernet card
- 64-bit Reactis requires 64-bit Windows and 64-bit MATLAB

There are different methods for installing Reactis, depending on how you will access the license(s) you have purchased:

1. via a Reactis License File residing on your computer, or
2. via a Reactis License Manager running on a remote server.

In the latter case, you (or someone in your organization) will also need to install the Reactis License Manager. The Reactis installers may be downloaded from the Reactis User Pages:

https://www.reactive-systems.com/login.msp

2.1 Installing with Local License File

To install Reactis, perform the following steps:

1. Execute the self-installing executable

   reactis-setup-V2020.exe

   and follow the instructions. The 64-bit Reactis installer is named

   reactis-setup-win64-V2020.exe

---

1Windows® is a registered trademark of Microsoft Corporation.
but otherwise installing 64-bit Reactis proceeds exactly as the 32-bit install. If you obtained your release on a DVD, inserting the DVD into your DVD drive should initiate the execution of this executable automatically. If it does not, manually run the program reactis-setup-V2020.exe that is included on the DVD. Alternatively, you may download the latest version of the installer from the Reactive Systems web site listed below. Note, that an updated version of the installer will have a name of the form reactis-setup-V2020.0.n.exe where n is a patch release number.

https://www.reactive-systems.com/login.msp

2. To obtain a Reactis license file, you will first need to locate the file rsilicense.dat, which is stored in the Reactis installation directory. If you installed in the default location, the license file should be found at one of the following two locations:

(64-bit Reactis)  C:\Program Files\Reactis V2020\rsilicense.dat
(32-bit Reactis)  C:\Program Files (x86)\Reactis V2020\rsilicense.dat

Once located, e-mail the (incomplete) rsilicense.dat to Reactive Systems at help@reactive-systems.com. You will receive a response containing the completed license as an attachment. When you receive this e-mail, replace rsilicense.dat in the Reactis installation directory with the completed license file.
2.2 Installing with Remote License Manager

If your organization already has a server running the Reactis License Manager, then you will need the name or IP address of the license server. The steps required to install Reactis on your computer (the client) are as follows:

1. On the client machine, run the installer (see step 1 of Section 2.1).
2. Invoke Reactis (see Section 2.5). The dialog shown in Figure 2.1 will appear. In this dialog, click the Add button and enter the IP address or name of a server running the Reactis License Manager.

Step 2 can be repeated multiple times if there is more than one server.

2.3 Installing the Reactis License Manager

Please see the Reactis License Manager User’s Guide.

2.4 Performing a Silent Install

The Reactis installer supports a silent install (installing Reactis from the DOS command line in an automated fashion). The following command line switches control a silent install.
• /SILENT
  Instructs the installer to be silent. When the installer is silent, the wizard and the background window are not displayed but the installation progress window and error messages are displayed. The default settings (which can be overridden using command line arguments) are used for the install.

• /VERYSILENT With this switch, the installer will display error messages, but not the wizard, background window, or progress window.

• /NOCANCEL Prevents the user from canceling during the installation process by disabling the Cancel button and ignoring clicks on the close button. This switch is useful in conjunction with /SILENT.

• /DIR="x:\dirname" Overrides the default directory name displayed on the Select Destination Location wizard page. A fully qualified path name must be specified.

• /GROUP="folder name" Overrides the default folder name displayed on the Select Start Menu Folder wizard page.

• /COMPONENTS="comma separated list of component names" Overrides the default components settings. By default, all components are installed. Possible components are:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>Reactis GUI</td>
</tr>
<tr>
<td>lm</td>
<td>License Manager</td>
</tr>
<tr>
<td>lmo</td>
<td>License Monitor</td>
</tr>
<tr>
<td>help</td>
<td>Help files</td>
</tr>
<tr>
<td>examples</td>
<td>Example files</td>
</tr>
<tr>
<td>api</td>
<td>Reactis API</td>
</tr>
</tbody>
</table>

• /MATLABROOT="x:\dirname" Overrides the default MATLAB root directory name displayed on the Locate MATLAB wizard page.

• /COPYSETTINGS Instructs the installer to copy the personal settings and license information to the new installation if running in SILENT or VERYSILENT mode and a previous existing Reactis installation is found. This is the default.

• /NOCOPYSETTINGS Do NOT copy personal settings and license information from a previous existing installation.

• /SAVEINF="x:\filename" Saves information entered during the install in a file. That file can later be used to specify default parameters via LOADINF. The following information is saved:

  - Install directory name
  - Program group name
  - Components
  - MATLAB root directory
  - Should settings and license information be copied from existing versions of Reactis?

• /LOADINF="x:\filename" Load default information from a file created with the SAVEINF option. This is useful in combination with a silent install, i.e.
2.5 Starting Reactis

Reactis can be started in several ways:

- Select Reactis V2020 from the Windows Start menu, or
- double-click on the desktop shortcut if you installed one, or
- call the reactis.exe executable from a DOS command prompt which can be invoked with zero, one, or two arguments:
  - If no arguments are given Reactis starts with no model loaded.
  - If one argument is given it can be either the name of a Simulink model (.mdl or .slx file) or the name of an .rsi file. Reactis will start and load the specified model.
  - If two arguments are given the first is assumed to be a model name and the second is an .rsi file and Reactis will start and load the specified model and .rsi file.
Chapter 3

Getting Started with Reactis

This chapter provides a quick overview of Reactis. It contains a brief description of each main component of the tool suite in the form of an extended “guided-tour” of the different features of the tool suite. Each “stop” in the tour is indicated by a §. The tour uses as a running example a simple Simulink / Stateflow model of an automobile cruise control that is included in the Reactis distribution.

3.1 A Note on Model Preparation

The cruise-control example included with the distribution does not require any special processing before you run Reactis on it; you may load the file immediately and start the guided tour. However, there is an important step that you should undertake when you are preparing models for use with Reactis. This section describes this preparatory step and discusses the Simulink operations needed to perform it.

Reactis supports a large portion of the discrete-time subset of Simulink and Stateflow. As it processes a Simulink / Stateflow model, it also interacts with MATLAB in order to evaluate MATLAB expressions, including references to workspace data items, that a model may contain.

In order for Reactis to process a model, Reactis must be able to automatically set up the MATLAB workspace data used by the model. For this reason, any workspace data items that a model uses must be initialized within one of the following locations (for more details see Section 14.1.1):

• Any Simulink model callback or block callback that is executed when loading or running the model (PreLoadFcn, PostLoadFcn, InitFcn, StartFcn).

• A “startup.m” file located in the folder where the model file is located.

• The Callbacks pane of the Reactis Info File Editor.

Below we describe how the workspace initialization was established in the cruise-control model file included in the Reactis release. In that example the MATLAB file cruise_constants.m defines two workspace variables that are used in the Simulink / Stateflow model file cruise.slx. The cruise_constants.m file was attached to cruise.slx using the following steps.

1. Load cruise.slx into Simulink.
2. In the Simulink window, select the File → Model Properties → Model Properties menu item\(^1\).

3. In the resulting dialog, select the Callbacks tab.

4. In the PreLoadFcn\(^*\) section, enter cruise\_constants; (note that the .m suffix is not included).

5. Save the model.

This saved cruise.slx file is distributed with Reactis, so you do not need to undertake the above steps yourself in order to load and process this file in Reactis. The above steps are cited only for illustrative purposes.

### 3.2 Reactis Top Level

![Reactis window](image)

**Figure 3.1: The Reactis window.**

The Reactis top-level window contains menus and a tool bar for launching and controlling verification and validation activities. Reactis is invoked as follows.

\[\text{Select Start → All Programs → Reactis V2020 → Reactis from the Windows Start menu (or double-click on the desktop shortcut if you installed one).}\]

You now see a Reactis window like the one as shown in Figure 3.1. A model may be selected for analysis as follows.

\[\text{Click the \(\text{\textcolor{white}{\text{'}}θ\text{\textcolor{white}{\text{'}}}\) \text{file open) button located at the leftmost position on the tool bar. Then use the resulting file-selection dialog to choose the file cruise.slx from the examples folder of the Reactis distribution.}^2}\]

\(^1\)Prior to R2012b, the dialog was invoked via menu item File → Model Properties...
3.2. REACTIS TOP LEVEL

Loading the model causes the top-level window to change as shown in Figure 3.2. The panel to the right shows the top-level Simulink diagram of the model; the panel to the left shows the model hierarchy. In addition, the title bar now reports the model currently loaded, namely cruise.slx, and the Reactis info file (.rsi file) cruise.rsi that contains testing information maintained by Reactis for the model. Info files are explained in more detail in the next section (Section 3.3).

It is also worth noting that if during installation you chose to associate the .rsi file extension with Reactis, then you can start Reactis and open cruise.slx in a single step by double-clicking on cruise.rsi in Windows Explorer.

![Figure 3.2: Reactis after loading cruise.slx.](image)

Subsystems in the hierarchy panel are tagged with icons indicating whether they are Simulink (SL), Stateflow (SF), Embedded MATLAB (ML) \(^3\), C source files (C), C libraries (LIB), S-Functions (FN) \(^4\), assertions (✓), or user-defined targets (☺).

Reactis provides a signal-tracing mechanism which allows the path taken by a signal to be quickly identified. To trace a signal, left-click on any part of the signal line.

---

\(^3\)Embedded MATLAB only shows up if you are using the Reactis for EML Plugin

\(^4\)C code subsystems only show up if you are using the Reactis for C Plugin
In the model hierarchy panel, left-click on the CruiseMDL subsystem, and then, in the main panel, left-click on the signal line emerging from the decelSet inport (inport 4). The result should be similar to Figure 3.3. Click the \( \uparrow \) button twice to see how to trace the signal through different levels of the model hierarchy.

As shown in Figure 3.3, signals are highlighted in yellow when left-clicked on. The route of the highlighted signal can then be easily identified. To turn off the highlighting, left-click on empty space in the main window.

![Figure 3.3: Highlighting a signal.](image)

### 3.3 The Info File Editor

Reactis does not modify the .slx file for a model. Instead the tool stores model-specific information that it requires in a .rsi file. The primary way for you to view and edit the data in these files is via the Reactis Info File Editor, which is described briefly in this section and in more detail in Chapter 5.

The next stop in the guided tour explains how this editor may be invoked on cruise.rsi.
§4 Select the Edit → Inport Types... menu item.

This starts the Reactis Info File Editor, as shown in Figure 3.4. Note that the contents of the .rsi file may only be modified when Simulator is disabled. When Simulator is running, the Info File Editor operates in read-only mode as indicated by “[read only]” in the editor window’s title bar.

Figure 3.4: The Reactis Info File Editor.

.rsil files contain directives partitioned among the following panes in the Info File Editor.

**Inport Types.** A type for each top-level inport is used by Reactis to constrain the set of values fed into top-level inports during simulation, test-case generation, and model validation. For example, you can specify an inport range by giving minimum and maximum values.

**Configuration Variables.** Configuration variables are user-selected workspace data items which can change in between tests in a test suite (but not during a test).

**Test Points.** Test points are internal signals which have been selected (by the user) to be logged in test suites.

**Outport Tolerances.** Each outport of a model may be assigned a tolerance. When executing a test suite on a model in Reactis Simulator, the tolerance specifies the maximum acceptable difference between the value computed by the model for the outport and the value stored in a test suite for the outport.

**General.** Various settings that determine how a model executes in Reactis.

**Error Checking.** Specify how Reactis should respond to various types of errors (e.g. overflow, NaN, etc.).

**Coverage Metrics.** Specify which coverage metrics will be tracked for the model and adjust the configurable metrics (CSEPT, MC/DC, MCC, Boundary).
Excluded Coverage Targets. Specify individual targets to be ignored when measuring the coverage of a model.

Validator Objectives. Validator objectives (including assertions, user-defined targets, and virtual sources) are instrumentation that you may provide to check if a model meets its requirements. You can double-click on an item in this list to highlight the location of the objective in the model.

C Code. If the Reactis for C Plugin is installed, this pane will contain information about C-coded S-functions, C code embedded within Stateflow charts and C library code called from either an S-function or Stateflow.

External EML Functions. If the Reactis for EML Plugin is installed, this pane let’s you specify any external Embedded MATLAB functions used by your model (i.e. functions stored in .m files outside your model).

Callbacks. These are fragments of MATLAB code that Reactis will run before or after loading a model in Simulink. Note, that these callbacks are distinct from those maintained by Simulink.

Search Path. The model-specific search path is prepended to the global search path to specify the list of folders (for the current model) in which Reactis will search for files such as Simulink model libraries (.slx), MATLAB scripts (.m), and S-Functions (.dll, .mexw32, .m).

File Dependencies. A list of files on which the current model depends. The file-dependency information enables Reactis to track changes in auxiliary model files so that information obtained from them for the purposes of processing the current model may be kept up to date. Typically, files listed here would include any .m files loaded as a result of executing the current model’s pre-load function. For example, a .m file mentioned in the pre-load function might itself load another .m file; this second .m file (along with the first) should be listed as a dependency in order to ensure that Reactis behaves properly should this .m file change. Dependencies on libraries (.slx files) are detected automatically and need not be listed here.

§5
To modify the type for an inport, select the appropriate row, then right-click on it and select Edit..., and make the desired change in the resulting dialog. Save the change and close the type editor by clicking Ok or close without saving by clicking Cancel.

The types that may be specified are the base Simulink / Stateflow types extended with notations to define ranges, subsets and resolutions, and to constrain the allowable changes in value from one simulation step to the next. More precisely, acceptable types can have the following basic forms.

Complete range of base type: By default the type associated with an inport is the Simulink / Stateflow base type inferred from the model. Allowed base types include: int8, int16, int32, uint8, uint16, uint32, boolean, single, double, sfix*, and ufix*.

Subrange of base type: $bt[i, j]$, where $bt$ is a Simulink / Stateflow base type, and $i$ and $j$ are elements of type $bt$, with $i$ being a lower bound and $j$ an upper bound.
### 3.3. THE INFO FILE EDITOR

<table>
<thead>
<tr>
<th>RSI Type</th>
<th>Values in Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double [0.0,4.0]</td>
<td>All double-precision floating-point numbers between 0.0 and 4.0, inclusive.</td>
</tr>
<tr>
<td>int16 [-1,1]</td>
<td>-1, 0, 1</td>
</tr>
<tr>
<td>int32 {0,4,7}</td>
<td>0, 4, 7</td>
</tr>
<tr>
<td>uint8 {0:1,1:3}</td>
<td>0, 1</td>
</tr>
<tr>
<td>double [0.0 : 0.5 : 2.0]</td>
<td>0.0, 0.5, 1.0, 1.5, 2.0</td>
</tr>
<tr>
<td>int16 [0,3] delta [-1, 1]</td>
<td>0, 1, 2, 3; inports of this type can increase or decrease by at most 1 in successive simulation steps.</td>
</tr>
<tr>
<td>if t == 0 then double { 0.0 } else double [0.0,10.0]</td>
<td>At simulation time zero, input has value 0.0; subsequently, input is between 0.0 and 10.0.</td>
</tr>
</tbody>
</table>

Table 3.1: Type examples.

**Subrange with resolution type:** $bt[i : j : k]$, where $bt$ is either double or single, $i$ is a lower bound, $j$ is a resolution, and $k$ is an upper bound; all of $i$, $j$ and $k$ must be of type $bt$. The allowed values that an inport of this type may have are of form $i + n \times j$, where $n$ is a positive integer such that $i + n \times j \leq k$. In other words, each value that an inport of this type may assume must fall between $i$ and $k$, inclusive, and differ from $i$ by some integer multiple of $j$.

**Set of specific values:** $bt\{e_1, \ldots , e_n\}$, where $bt$ is a base type and $e_1, \ldots , e_n$ are either elements of type $bt$, or expressions of form $v : w$, where $v$ is an element of type $bt$ and $w$, a positive integer, is a probability weight. These weights may be used to influence the relative likelihood that Reactis will select a particular value when it needs to select a value randomly to assign to an inport having an subset type. If the probability weight is omitted it is assumed to be “1”. For example, an inport having type uint\{0:1, 1:3\} would get the value 0 in 25\% of random simulation steps and the value 1 in 75\% of random simulation steps, on average.

**Delta type:** $tp\ delta [i, j]$, where $tp$ is either a base type, a range type, or a resolution type, and $i$ and $j$ are elements of the underlying base type of $tp$. Delta types allow bounds to be placed on the changes in value that variables may undergo from one simulation step to the next. The value $i$ specifies a lower bound, and $j$ specifies an upper bound, on the size of this change between any two simulation steps. More precisely, if a variable has this type, and $v$ and $v'$ are values assumed by this variable in successive simulation steps, with $v'$ the later value, then the following mathematical relationship holds: $i \leq v' - v \leq j$. Note that if $i$ is negative then values are allowed to decrease as simulation progresses.

**Conditional type:** if $expr$ then $tp_1$ else $tp_2$, where $expr$ is a Boolean expression and $tp_1$ and $tp_2$ are type constraints. If $expr$ is true then the input assumes a value specified by $tp_1$ otherwise it assumes a value specified by $tp_2$. Currently, the only variable allowed in $expr$ is the simulation time expressed as $t$.

Table 3.1 gives examples of types and the values they contain. For vector, matrix or bus inports, the above types can be specified for each element independently.

If any change is made to a port type, then “[modified]” appears in the title bars of the Info File Editor and the top-level Reactis window. You may save changes to disk by selecting File → Save from the Info File Editor or File → Save Info File from the top-level window.
If no .rsi file exists for a model, Reactis will create a default file the first time you open the Info File Editor, or start Simulator or Tester. The default type for each inport is the base type inferred for the port from the model. If you add or remove an inport to your model you can synchronize your .rsi file with the new version of the model by selecting Tools → Synchronize Inports, Outports, Test Points from the Info File Editor.

§6 Select the File → Exit menu item on the editor's tool bar to close the Reactis Info File Editor.
3.4 Simulator

Reactis Simulator provides an array of facilities for viewing the execution of models. To continue with the guided tour:

§7 Click the (enable Simulator) button in the tool bar to start Reactis Simulator.

This causes a number of the tool-bar buttons that were previously disabled to become enabled. Simulator performs simulations in a step-by-step manner: at each simulation step inputs are generated for each top-level inport, and resultant outputs reported on each outport. You can control how Simulator computes top-level inport values using the Source-of-Inputs dialog as follows. Simulator may be directed to:

1. generate values randomly (this is the default),
2. query the user,
3. read inputs from a Reactis test suite. Such tests may have been generated automatically by Reactis Tester, constructed manually in Reactis Simulator, or imported from a file storing test data in a comma-separated-value format.

To set the input source for an inport, use the Source-of-Inputs dialog located to the left of in the tool bar (see Figure 3.5). The next part of the guided tour illustrates the use of each of these input sources; interspersed with this discussion are asides on coverage tracking, data-value tracking, and other useful features.

Figure 3.5: The Source-of-Inputs dialog allows you to control how Reactis Simulator generates values for top-level inports when executing a model.

3.4.1 Generating Random Inputs

As the random input source is the default, no action needs to be taken to set this input mode.

§8 In the model hierarchy panel on the left of the Simulator window, click on CruiseMain/CruiseMDL/Mode in order to display the Stateflow diagram in the main window. Then, click the (Run/Pause Simulation) button.

During simulation, blocks, signals, states, and transitions in the diagram are highlighted in green as they are entered and executed. The simulation stops automatically when the number of simulation steps reaches the figure contained in the entry box to the left of the Source-of-Inputs dialog. Before then, you may pause the simulation by clicking the (Run/Pause
Simulation) button a second time. Note, that simulation will likely pause in the middle of a simulation step. You may then click:

- to continue executing a block at a time, or
- to complete the step, or
- to back up to the start of the step

### 3.4.2 Tracking Model Coverage

While Simulator is running you may also track coverage information regarding which parts of your model have been executed and which have not. These coverage-tracking features work for all input-source modes. The next portion of the guided tour illustrates how these features are used.

Make sure menu item Coverage → Show Details is selected (as indicated by a check mark in the menu). Select menu item Coverage → Show Summary.

![Figure 3.6](image)

Figure 3.6: The Reactis coverage tracking features convey which parts of a model have been exercised. Note that the coverage information you see at this point will be different than that shown above. This is because a different sequence of inputs (from the random ones you used) brought the simulation to the displayed state.

A dialog summarizing coverage (which blocks/states have been entered and which transitions have fired) now appears, with elements of the diagram not yet exercised drawn in red, as shown in Figure 3.6. Note that poor coverage is not uncommon with random simulation. You may hover over a covered element to determine the (1) test (in the case being considered here, the “test” is the current simulation run and is rendered as in the hovering information) and (2) step within the test that first exercised the item. This type of test and step coverage
information is displayed with a message of the form test/step. You may view detailed information for the Condition, Decision and MC/DC coverage metrics, which involve Simulink logic blocks and Stateflow transition segments whose label includes an event and/or condition, as follows.

§10 Perform the following.

1. In the model hierarchy panel on the left of the Reactis window, click on CruiseMain/CruiseMdl/Mode to display the Stateflow diagram in the main panel. Note, this was done in a previous step, so it may not be necessary here.

2. Right-click on the transition from state Init to state Active, and select View Coverage Details. A Coverage Details dialog will appear.

3. In the Coverage Details dialog, select the MCC tab to display and inspect information related to Multiple Condition Coverage.

4. Right-click on the bottom cell in the Coverage column of the MCC table and then left-click on Track Coverage in the pop-up menu. An Exclude Target dialog will appear.

5. In the Exclude Target dialog, select Exclude this target from coverage and click on Ok to close the dialog. The status of the MCC target will change to excluded.

6. Right-click on the bottom cell in the Coverage column of the MCC table and open the Exclude Target dialog a second time. Select Track coverage for this target and click on Ok to close the dialog. The status of the MCC target will change back to its original value.

The Coverage Details dialog, shown in Figure 3.7, has two tabs: Decision and MCC. The Decision tab displays details for decision coverage, condition coverage, and MC/DC. The table in this figure gives information for the decision:

\[
\text{set} == 1 \ \&\& \ \text{deactivate} == 0
\]

This decision contains two conditions:

- set == 1
- deactivate == 0

Conditions are the atomic boolean expressions that are used in decisions. The first two columns of the table list the test/step information for when the decision first evaluated to true and when it first evaluated to false. A value \(-/-\) indicates that the target has not yet been covered. The third column lists the conditions that make up the decision, while the forth and fifth columns give test/step information for when each condition was evaluated to true and to false.

MC/DC Coverage requires that each condition independently affect the outcome of the decision in which it resides. When a condition has met the MC/DC criterion in a set of tests,
Figure 3.7: The Coverage Details dialog includes two tabs: Decision and MCC. The Decision tab includes details of decision coverage, condition coverage, and MC/DC. The MCC tab shows multiple condition coverage.

The sixth and seventh columns of the table explain how. Each element of these two columns has the form $b_1b_2...b_n$:test/step, where $b_i$ reports the outcome of evaluating condition $i$ in the decision (as counted from left to right in the decision or top to bottom in column three) during the test and step specified. Each $b_i$ is either T to indicate the condition evaluated to true, F to indicate the condition evaluated to false, or x to mean the condition was not evaluated due to short-circuiting.

The MCC tab of the Coverage Details dialog displays details of multiple condition coverage (MCC) which tracks whether all possible combinations of condition outcomes for a decision have been exercised. The table includes a column for each condition in the decision. The column header is the condition and each subsequent row contains an outcome for the condition: True, False, or x (indicating the condition was not evaluated due to short-circuiting). Each row also contains the outcome of the decision (True or False) and, when covered, the test and step during which the combination was first exercised.

It should be noted that the previous scenario relied on randomly generated input data, and replaying the steps outlined above will yield different coverage information than that depicted in Figure 3.7.
An alternative way to query coverage information is to invoke the Coverage-Report Browser by selecting Coverage → Show Report. This is a tool for viewing or exporting coverage information that explains which model elements have been covered along with the test and step where they were first exercised. Yet another way to inspect test coverage is with Simulate → Fast Run With Report.... In addition to coverage data, these reports can contain other types of information such as plots showing expected outputs compared to actual outputs.

A simulation run, and associated coverage statistics, may be reset by clicking the Reset to First Step button (●) in the tool bar.

3.4.3 Reading Inputs from Tests

Simulation inputs may also be drawn from tests in a Reactis test suite. Such a test suite may be generated automatically by Reactis Tester, constructed manually in Reactis Simulator, or imported from a file storing test-data in a comma-separated-value format. By convention, files storing Reactis test suites have a .rst file-name extension. A Reactis test suite may be loaded into Simulator as follows.

§11 Click the button in the tool bar to the right of the Source-of-Inputs dialog and use the file-selection dialog to select cruise.rst in the examples folder of the Reactis distribution.

This causes cruise.rst, the name of a test suite file generated by Reactis Tester, to appear in the title bar and the contents of the Source-of-Inputs dialog to change; it now contains a list of tests that have been loaded. To view this list:

§12 Click on the Source-of-Inputs dialog (located to the left of in the tool bar).

Each test in the suite has a row in the dialog that contains a test number, a sequence number, a name, and the number of steps in the test. Clicking the “all” button in the lower left corner specifies that all tests in the suite should be executed one after another. To execute the longest test in the suite:

§13 Do the following.

1. Select the longest test in the test suite in the dialog.
2. Click the (Run Fast Simulation) button.

If you look at the bottom-right corner of the window, you can see that the test is being executed (or has completed), although the results of each execution step are not displayed graphically. When the test execution completes, the exercised parts of the model are drawn in black. If the Run Simulation button (●) is clicked instead, then the results of each simulation step are rendered graphically, with the consequence that simulation proceeds more slowly.

Whenever tests are executed in Simulator, the value computed by the model for each top-level outport and test point is compared against the corresponding value stored in the test suite. The tolerance for this comparison can be configured in the Info File Editor. An HTML report listing any differences (as well as any runtime errors encountered) can be generated by loading a test suite in Simulator and then selecting Simulate → Fast Run With Report.
3.4.4 Tracking Values of Data Items

When Simulator is paused\(^5\), you may view the current value of a given data item (Simulink block or signal line, Stateflow variable, Embedded MATLAB variable, or C variable) by hovering over the item with your mouse cursor. You may also select data items whose values you wish to track during simulation using the watched-variable and scope facilities of Simulator.

§14 In the model hierarchy panel on the left of the Reactis window, select the top-level of the model (“cruise”) for display. Then:

1. Right-click on the outport “active” and select Add to Watched from the resulting pop-up menu.
2. Hover over outport “active” with the mouse; a pop-up displays its current value.

The bottom of the Simulator window now changes to that indicated in Figure 3.8.

![Figure 3.8: The watched-variable panel in Simulator displays the values of data items, as does hovering over a data item with the mouse.](image)

The watched-variable panel shows the values of watched data items in the current simulation step, as does hovering over a data item with the mouse. Variables may be added to, and deleted from, the watched-variable panel by selecting them and right-clicking to obtain a menu. You may also toggle whether the watched-variable list is displayed or not by selecting the View → Show Watched Variables menu item.

Scopes display the values a given data item has assumed since the beginning of the current simulation run. To open a scope:

\(^5\)A simulation is paused if Simulator is not actively computing simulation step(s).
§15 Perform the following.

1. In the model hierarchy panel, select CruiseMain/CruiseMdl.

2. Right-click on inport “speed” and select Open Scope from the resulting pop-up menu. A new window like the one in Figure 3.9 (a) appears and plots the speed of the car during the simulation run.

3. Right-click on the “dSpeed” signal that exits the “DesiredSpeed” subsystem and select Add to Scope → CruiseMDL.speed. This causes the desired speed to be added to the actual speed scope previously opened, so that the scope now appears similar to Figure 3.9 (b).

4. To overlay the two signals to simplify comparing them, click the button in the scope tool bar, so that the scope now appears similar to Figure 3.9 (c).

Figure 3.9: Plotting actual speed (yellow) and desired speed (green). The third button from the left toggles whether the signals are overlaid (c) or not (b). Note that the plots you see will be different than that shown above. This is because a different sequence of inputs brought the simulation to the displayed state.

Scopes also have a zoom feature which is particularly useful for viewing the details of long signals. To zoom in, select a region of interest by clicking in the scope and dragging to specify a region. To scroll, hold down the control key, then click in the scope and drag. See Section 7.5.3 for more details on scopes.
§16 Perform the following.

1. Select a region of interest within the scope by left-clicking and dragging while holding the mouse button down. Release the button. This zooms the scope in to view the selected area.

2. Select a second region of interest within the scope by left-clicking and dragging. This zooms the scope in a second time.

3. Hold down the control (Ctrl) key and left-click within the scope. The cursor will change to a plus sign (+). Move the mouse while holding down the mouse button. This will shift the region displayed within the scope. Release the mouse button.

4. Right-click on the scope. This zooms the scope out so that the region selected during step 1 is now displayed.

5. Right-click on the scope. This zooms the scope out to view the area selected before step 1 was performed. In this case, the entire signal will be visible.

§17 Close the watched-variable panel by selecting the View → Show Watched Variables menu item.

Figure 3.10: The Next Input Values dialog enables you to hand-select values for top-level inputs at each simulation step. Values for configuration variables can be specified at the start of a test.
### 3.4.5 Querying the User for Inputs

The third way for Simulator to obtain values for inports is for you to provide them. To enter this mode of operation:

| §18 | Select User Guided Simulation from the Source-of-Inputs dialog. |

Upon selecting this input mode, a Next Input Values dialog appears (as shown in Figure 3.10). This dialog lets you to specify the input values for the next simulation step and see the values computed for outputs and test points. Initially, each top-level inport of the model has a row in the dialog. You can remove inputs from the dialog or add outputs, test points, and configuration variables by clicking the gear button (⚙️) in the toolbar of the Next Input Values dialog. The row for each inport has six columns that determine the next input value for the corresponding inport as follows.

1. The name of the item (inport, outport, test point, or configuration variable).

2. This checkbox toggles whether the item is included in a scope displaying a subset of the signals from the Next Input Values dialog.

3. This pull-down menu has several entries that determine how the next value for the inport is specified:
   - **Random**: Randomly select the next value for the inport from the type given for the inport in the `.rsi` file.
   - **Entry**: Specify the next value with the text-entry box in column four of the panel.
   - **Min**: Use the minimum value allowed by the inport’s type constraint.
   - **Max**: Use the maximum value allowed by the inport’s type constraint.
   - **Test**: Read data from an existing test suite.

4. If the pull-down menu in column three is set to “Entry”, then the next input value is taken from this text-entry box. The entry can be a concrete value (e.g. integer or floating point constant) or a simple expression that is evaluated to compute the next value. These expressions can reference the previous values of inputs or the simulation time. For example, a ramp can be specified by `pre(drag) + 0.0001` or a sine wave can be generated by `sin(t) * 0.001`. For the full description of the expression notation see Section 7.4.1.

5. If the pull-down menu in column three is set to “Entry”, then clicking the history button (labeled H) displays recent values the inport has assumed. Selecting a value from the list causes it to be placed in the text-entry box of column four.

6. The arrow buttons in this column enable you to scroll through the possible values for the port. The arrows are not available for ports of type double or single or ranges with a base type of double or single.

When Run Fast Simulation (onyms) is selected, the inport values specified are used for each subsequent simulation step until the simulation is paused. The toolbar at the top of the Next Input Values dialog includes buttons for stepping (mini-step, single-step, fast simulation, reverse step, etc.) that work the same as those in the top-level Simulator window. The text entry box to the left of the toolbar lets you enter a search string to cause Reactis to display only those items whose name contains the search string.
One use case for user guided simulation is for constructing functional tests. A test to ensure that the cruise control is enabled and disabled as expected can be created as follows.
§19 In the following “Take a step” means click ➤ in the Next Input Values toolbar. Take \( n \) steps means click ➤ \( n \) times.

1. If the Next Input Values dialog is not still open from the prior step, re-open it by selecting User Guided Simulation from the Source-of-Inputs dialog.

2. In the Next Input Values toolbar, click the ⚪ button to open the Select Signals dialog.

3. In the Input Ports tab that opens, click the header Entry to select that radio button for all inputs. For each of the last three inputs (gas, inactiveThrottleDelta and drag) uncheck the Panel checkbox and change the radio button to Random.

4. Select the Output Ports tab, then check the Panel checkbox for active.

5. Select the Configuration Variables tab, then for InitialSpeed select Entry and enter a value of 30.0. Click Ok.

6. In the Next Input Values toolbar, check the checkbox (second item from left) to open a scope to plot all signals in the dialog. This will help you visualize the test as you construct it. In particular, the bottom plot for active will be inspected to check that the model is properly enabling and disabling the cruise control. Figure 3.11 shows what the scope will look like when you complete construction of the test.

7. In the entry box for onOff enter 1.0, for all other inputs enter 0.0. The arrows to the far right can be used to toggle between 0.0 and 1.0.

8. Take six steps. The scopes hold the specified values for each step.

9. Set decelSet to 1.0 (indicating we want to turn the cruise control on) and take a step. Change decelSet back to 0.0 and take four steps. Scope for decelSet shows it going from 0.0 to 1.0, then back to 0.0. Cruise control becomes 1.0 indicating it is enabled.

10. Change cancel to 1.0 and take a step, then change cancel back to 0.0 and take four steps. Cruise control is disabled.

11. Change accelResume to 1.0 and take a step. Change it back to 0.0 and take four steps. Cruise control is enabled.

12. Change brake to 1.0 and take a step. Change it back to 0.0 and take four steps. Cruise control is disabled.

13. In the main Reactis window, select Test Suite → Add/Extend Test, then Test Suite → Save. This adds the newly constructed test to the current test suite and then saves the test suite. Now, whenever this test is run in Reactis Simulator, Reactis will check that the value computed by the model for active, matches the value stored in the test suite (which you just checked manually).
Figure 3.11: This scope shows a functional test constructed with user guided simulation that checks that the cruise control is enabled and disabled as expected.
3.4.7 Other Simulator Features

Simulator has several other noteworthy features. You may step both forward and backward through a simulation using toolbar buttons:

- The ▶ button executes a single execution step.
- The ▼ button executes the next “mini-step” (a block or statement at a time).
- The ◀ button, which causes the simulation to go back (undo) a single step.
- The ◀ button causes the simulation to go back multiple steps.
- If using the Reactis for C Plugin with a model that includes C code, additional buttons appear for stepping through C code: ◀, ▶, ◀, ◀, ◀, ◀. See Chapter 16, for a description of how these buttons work.

You may specify the number of steps taken when ▶, ◀, or ◀ are pressed by adjusting the number in the text-entry box to the right of ◀. When a simulation is paused at the end of a simulation step (as opposed to in the middle of a simulation step), the current simulation run may be added to the current test suite by selecting the menu item Test Suite → Add/Extend Test. After the test is added it appears in the Source-of-Inputs dialog. After saving the test suite with Test Suite → Save, the steps in the new test may be viewed by selecting Test Suite → Browse. A model (or portion thereof, including coverage information) may be printed by selecting File → Print...

Breakpoints may be set by either:

- right clicking on a subsystem or state in the hierarchy panel and selecting Toggle Breakpoint; or
- right clicking on a Simulink block, Stateflow state, or Stateflow transition in the main panel and selecting Toggle Breakpoint. Additionally, if you are using the Reactis for EML Plugin or the Reactis for C Plugin, then a breakpoint may be set on any line of C/EML code that contains a statement by right-clicking just to the right of the line number and selecting Toggle Breakpoint.

The • symbol is drawn on a model item when a break point is set. During a simulation run, whenever a breakpoint is hit, Simulator pauses immediately.

3.5 Tester

Tester may be used to generate a test suite (a set of tests) automatically from a Simulink / Stateflow model as shown in Figure 3.12. The tool identifies coverage targets in the model and aims to maximize the number of targets exercised by the generated tests.

To start Tester:

§20 Select the Test Suite → Create menu item.
This causes the window shown in Figure 3.13 to appear. If you specify existing test suites in the Preload Files section, Tester will execute those suites, then add additional test steps that exercise targets not covered by the preloaded suite(s). The second section (Run for) determines how long Tester should run. The third section (Coverage Objectives) lists the metrics which Tester will focus on. In the fourth section, you specify the name of the output file in which Tester will store the new test suite (Output File).

There are three options provided in the Run for section. The default option, as shown in Figure 3.13, is to run Tester for 20,000 steps. Alternatively, you may choose to run Tester for a fixed length of time by clicking on the top radio button in the Run for section, after which the steps entry box will be disabled and the hours and minutes entry boxes will be enabled. These values are added together to determine the total length of time for which Tester will run, so that entering a value of 1 for hours and 30 for minutes will cause Tester to run for 90 minutes. See Chapter 8 for usage details of the less-often used third radio button in the Run for section.

The Coverage Objectives section contains check boxes which are used to select the kinds of targets Tester will focus on while generating tests. Chapter 6 describes the different types of coverage tracked by Reactis.

To generate a test suite in the guided tour, retain the default settings and:

§21 Click the Create Suite button.

The Tester progress dialog, shown in Figure 3.14, is displayed during test-suite generation. When Tester terminates, a results dialog is shown, and a file cruise.rst containing the suite is produced. The results dialog includes buttons for loading the new test suite into the Test-Suite Browser (see below) or Reactis Simulator.
3.6 The Test-Suite Browser

The Test-Suite Browser is used to view the test suites created by Reactis. It may be invoked from either the Tester results dialog or the Reactis top-level window.

Select the Test Suite → Browse menu item and then cruise.rst from the file selection dialog that pops up.

A Test-Suite Browser window like the one shown in Figure 3.15 is then displayed. The Test Data tab of the browser displays the test selected in the button/dialog located near the center of the browser’s tool bar. The main panel in the browser window shows the indicated test as a matrix, in which the first column gives the names of input and output ports of the model and each subsequent column lists the values for each port for the corresponding simulation step. The simulation time is displayed in the output row labeled “___t___”. The buttons on the tool bar may be used to scroll forward and backward in the test. The Test History and Suite History tabs display history information recorded by Reactis for the test and the suite as a whole.

The Filter entry box on the right side of the toolbar lets you search for test steps that satisfy a given condition. You enter a boolean expression in the search box and then select the filter check box to search for test steps in the suite for which the expression evaluates to true. For
example, to see each step where the cruise control is active enter “active == 1” and then select the Filter check box.

The Test-Suite Browser may also be used to display the entire set of values passing over a port during a single test or set of tests.

§23 Perform the following in the Test-Suite Browser window.

1. Left click on the row for inport *drag*.
2. Right click on the row and select Open Distribution Scope (current test).
3.7. VALIDATOR

Reactis Validator is used for checking whether models behave correctly. It enables the early detection of design errors and inconsistencies and reduces the effort required for design reviews. The tool gives you three major capabilities.

A dialog similar to that shown in Figure 3.16 appears. In the figure, each value assumed by the inport `drag` is represented by a yellow dot.

![Figure 3.15: The Reactis Test-Suite Browser.](image)

![Figure 3.16: Values passing over inport `drag` during a test.](image)
CHAPTER 3. GETTING STARTED WITH REACTIS

Assertion checking. You can instrument your models with assertions, which monitor model behavior for erroneous scenarios. The instrumentation is maintained by Reactis; you need not alter your .slx file. If Validator detects an assertion violation, it returns a test highlighting the problem. This test may be executed in Reactis Simulator to uncover the source of the error.

Test scenario specifications. You can also instrument your models with user-defined targets, which may be used to define test scenarios to be considered in the analysis performed by Tester and Validator. Like assertions, user-defined targets also monitor model behavior; however, their purpose is to determine when a desired test case has been constructed (and to guide Reactis to construct it), so that the test may be included in a test suite.

Concrete test scenarios. Finally, you can place virtual sources at the top level of a model to control one or more top-level inports as a model executes in Reactis. That is, you can specify a sequence of values to be consumed by an import during simulation or test-generation. Virtual sources can be easily enabled and disabled. When enabled, the virtual source controls a set of inports and while disabled those inports are treated by Reactis just as normal top-level inports.

Conceptually, assertions may be seen as checking system behavior for potential errors, user-defined targets monitor system behavior in order to check for the presence of certain desirable executions (tests), and virtual sources generate sequences of values to be fed into model inports. Syntactically, Validator assertions, user-defined targets, and virtual sources have the same form and are collectively referred to as Validator objectives.

Validator objectives play key roles in checking a model against requirements given for its behavior. A typical requirements specification consists of a list of mandatory properties. For example, a requirements document for an automotive cruise control might stipulate that, among other things, whenever the brake pedal is depressed, the cruise control should disengage. Checking whether or not such a requirement holds of a model would require a method for monitoring system behavior to check this property, together with a collection of tests that would, among other things, engage the cruise control and then apply the brake. In Validator, the behavior monitor would be implemented as an assertion, while the test scenario involving engaging the cruise control and then stepping on the brake would be captured as a user-defined target.

3.7.1 Manipulating Validator Objectives

This section gives more information about Validator objectives.

§24 Do the following.

1. Disable Simulator by clicking the button in the top-level tool bar. (Validator objectives may only be modified or inserted when Simulator is disabled.)

2. In the model hierarchy panel, select the top-level of the model cruise for display in the main window.
After performing these operations, you now see a window like the one depicted in Figure 3.17. The three kinds of objectives are represented by different icons; assertions are denoted by a lightning bolt \( \mathbf{\text{\textbullet}} \), targets are marked by a cross-hair symbol \( \mathbf{\text{\textcircled{c}}} \), and virtual sources are represented by \( \mathbf{\text{\textcircled{v}}} \).

Validator objectives may take one of two forms.

**Expressions.** Validator supports a simple expression language for defining assertions, user-defined targets, and virtual sources.

**Simulink / Stateflow observer diagrams.** Simulink / Stateflow diagrams may also be used to define objectives. Such a Simulink subsystem defines a set of objectives, one for each outport of the subsystem. For assertion diagrams, a violation is flagged by a value of zero on an outport. For user-defined targets, coverage is indicated by the presence of a non-zero value on an outport. For virtual sources, each outport controls an inport of the model.

To see an example expression objective, do the following.

\$25\quad \text{Right click on the Brake assertion and select Edit Properties. When through with this step, click Cancel to dismiss the dialog.}$
A dialog like that shown in Figure 3.18 now appears. This dialog shows an expression intended to capture the following requirement for a the cruise control: “Activating the brake shall immediately cause the cruise control to become inactive.” Note that the dialog consists of five parts:

1. **Name.** This name labels the assertion in the model diagram.
2. **Enabled.** This check box enables and disables the expression.
3. **Expression.** This is a C-like boolean expression. The interpretation of such an assertion is that it should always remain “true”. If it becomes false, then an error has occurred and should be flagged. In this case, the expression evaluates to “true” provided that at least
one of brake and active is false (i.e. the conjunction of the two is false). Section 9.3.1 contains more detail on the expression notation.

4. **# steps to hold.** For assertions, this entry is an integer value that specifies the number of simulation steps that the expression must remain false before flagging an error. For user-defined targets, the entry specifies the number of steps that the expression must remain true before the target is considered covered.

5. **Inputs.** The entry boxes in the right column of this section list the variables used to construct the expression. These can be viewed as virtual input ports to the expression objective. The pull-down menus to the left of the section specify which data items from the model feed into the virtual inputs. Note that, although you can manually specify the inputs and wiring within this dialog, it is simpler to:

   (a) Leave this section blank when creating an objective.

   (b) After clicking Ok to dismiss the dialog, drag and drop signals from the main Reactis panel onto the objective.

To see an example diagram objective:

§26 Perform the following.

1. Using the model hierarchy panel, open the CruiseMDL subsystem located within the CruiseMain subsystem.

2. In the main panel, right-click on the assertion SpdCheck and select Edit Properties.

A dialog like the one in Figure 3.19 now appears. This dialog contains five sections.

1. **Name.** The name of the objective.

2. **Enabled.** This check box enables and disables the objective.

3. **Library.** The Simulink library containing the diagram for the objective. In the case of SpdCheck, the target resides in cruise_validator.slx.

4. **System.** The system in the library that is to be used as the objective. In this case, the system to be used is SpeedMaintenance.

5. **Parameters.** If the subsystem selected for use as an objective is a masked subsystem, then this panel is used to enter the relevant parameters. In the case of SpdCheck, no parameters are required because the indicated subsystem is not masked.

6. **Inputs.** The wiring panel is used to indicate which data items in the model should be connected to the inputs of the objective. In this case, SpdCheck has three inputs: speed, dSpeed, and active. The wiring information indicates that the first input should be connected to (the output of) Relational Operator1; the second should be connected to the dSpeed output of subsystem DesiredSpeed; and the third input should be connected to the speed inport of the current level of the model. In general, this panel contains pull-downs describing all valid data items in the current scope to which inputs of the objective may be connected.
Note, that, although you can manually specify the wiring within this dialog, it is simpler to:

(a) Leave the wiring unspecified after inserting an objective.

(b) After clicking Ok to dismiss the dialog, drag and drop signals from the main Reactis panel onto the objective.

Wiring information may be viewed by hovering over a diagram objective in the main Reactis model-viewer panel.

Diagram-based objectives may be viewed as monitors that read values from the model being observed and raise flags by setting their outport values appropriately (zero for false,
3.7. VALIDATOR

A diagram-based assertion in essence defines one “check” for each of its outports, with such an outport check being violated if it ever assumes the value zero. Similarly, a diagram-based target objective in essence defines one “target” for each outport; such a target is deemed covered if it becomes non-zero.

To view the diagram associated with SpdCheck:

§27 Perform the following.

1. Close the Properties dialog for SpdChk by clicking the Cancel button at the bottom of dialog box.
2. Double-click on the SpdChk icon.

These operations display the Stateflow diagram shown in Figure 3.20. This diagram encodes the following requirement: “While it is engaged, the cruise control shall not allow the desired and actual speeds to differ by more than 1 mile per hour for more than three time units.”

Figure 3.20: A Validator diagram assertion encoding a requirement that the cruise control properly maintain the speed of the car when engaged.

To understand how this diagram captures this requirement, note that the SpdCheck diagram has two top-level states, one corresponding to when the cruise control is active and one when it is inactive. The active state has three child states:

OkDiff Control resides here when the difference between actual and desired speed is within the accepted tolerance.

BigDiff Control resides here when the difference between actual and desired speed is greater than the accepted tolerance, but three time units have not yet elapsed.

Error Control resides here when the difference between actual and desired speed has been greater than the accepted tolerance for more than three consecutive time units.

Note that the transition action executed as state Error is entered sets the outport ok to 0. This is how an assertion violation is flagged.

Diagram objectives give you the full power of Simulink / Stateflow to formulate assertions and targets. The objectives may use any Simulink blocks supported by Reactis, including full Stateflow. The diagrams are created using Simulink and Stateflow in the same way standard models are built; they are stored in a separate .slx file from the model under validation.
Diagram wiring is managed by Reactis, so the model under validation need not be modified at all. As this information is stored in the \texttt{.rsi} file associated with the model, it also persists from one Reactis session to the next. After adding a diagram objective to a model, the diagram is included in the model’s hierarchy tree, just as library links are. See Chapter 9 for more details on using Reactis Validator.

### 3.7.2 Launching Validator

To use Validator to check for assertion violations:

\textbf{§28} Select the 	extit{Validate} \rightarrow \textit{Check Assertions...} \textbf{menu entry.}

![Validator interface](image)

Figure 3.21: Launching Validator brings up this screen.

A dialog like the one in Figure 3.21 now appears. The dialog is similar to the Tester launch screen in Figure 3.13 because the algorithms underlying the two tools are very similar. Conceptually, Validator works by generating thorough test suites from the instrumented model using the Tester test-generation algorithm and then applying these tests to the instrumented model to see if any violations occur. Note that when a model is instrumented with Validator objectives, the test-generation algorithm uses the objectives to direct the test construction process. In other words, Reactis actively attempts to find tests that violate assertions and cover user-defined targets. Validator stores the test suite it creates in the file specified in this dialog. The tests may then be used to diagnose any assertion violations that arise.

Note that if a model is instrumented with Validator objectives Reactis Tester also aims to violate the assertions and cover the user-defined targets.
Chapter 4

The Reactis Top-Level Window

The next several chapters of this manual contain detailed descriptions of different components of Reactis. This chapter concentrates on the functionality available in the top-level window when Simulator is turned off, or disabled. Simulator is always disabled when the Reactis top-level window first appears; it may also be explicitly disabled by clicking button ☺ (window item 12 in Figure 4.1). Clicking button 🎨 (window item 13) turns Simulator on (the functionality of the enabled mode is described in Chapter 7).

Figure 4.1: The Reactis top-level window.
4.1 Labeled Window Items

An annotated screenshot of the Reactis top-level window appears in Figure 4.1. This section describes the functionality of the numbered items in Figure 4.1, while Section 4.2 explains the workings of the pull-down menus.

The numbers below refer to the labels in Figure 4.1.

1. The *model hierarchy panel* shows the subsystems in the model and how they are related. Clicking on a + to the left of an item displays the subsystems of the item, while clicking on a − to the left of an item hides the subsystems. Clicking on the name or icon of an item causes the diagram for the item to be displayed in the main panel (window item 2). Pressing the “F2” key causes the parent of the currently displayed system to be displayed. Hovering in the hierarchy panel over a child of the currently displayed system causes the child to be highlighted in the main panel.

Right-clicking on an item in the hierarchy panel causes a pop-up menu to be displayed with two enabled entries: *Coverage Tracking* and *Copy System Path* and several disabled entries. The disabled menu items become enabled when Simulator is enabled and are described in Chapter 7.

The *Coverage Tracking* entry lets you modify how coverage is tracked for the given subsystem. Coverage for a subsystem can either be tracked cumulatively, or disabled altogether. If you disable coverage tracking for a subsystem, Reactis Tester will not attempt to exercise targets within the subsystem when generating tests and Simulator will not display coverage information for targets in the subsystem. See Section 6.5.2 for details on disabling coverage tracking. See Section 6.6 for details on cumulative coverage tracking.

The *Copy System Path* entry causes the path of the selected subsystem to be copied to the clipboard. When Reactis uncovers an issue in your model, the copied system path can help you easily navigate to the appropriate subsystem in the Simulink editor to resolve the problem. To do so:

(a) In the hierarchy panel, right-click on the subsystem of interest and select *Copy System Path*

(b) Start MATLAB and open your model in Simulink

(c) At the MATLAB prompt type `open_system('`

(d) Paste in the system path from the clipboard

(e) Type `');` to complete the command. At this point, the command you entered should look something like this: `open_system('cruise/CruiseMain/CruiseMDL');`

(f) Hit Enter. The subsystem of interest will be displayed in the Simulink editor.

2. The *main panel* displays the currently selected Simulink subsystem or Stateflow diagram. C code may also be displayed in the main panel if you are using Reactis for C Plugin. You can use the mouse to interact with the diagram in a number of different ways, including hovering over model items, single- or double-clicking on items, and right-clicking in various parts of the panel. Note that some operations in the main panel are only available when Simulator is enabled. See Chapter 7 for a description of those operations. The following mouse operations are available when Simulator is disabled:
### 4.1. LABELED WINDOW ITEMS

**Hovering...**
- over a From or Goto block will cause it and its associated block(s) to be highlighted in yellow.
- over a Data Store Read, Data Store Write or Data Store block will cause it and its associated block(s) to be highlighted in yellow.
- over a Validator objective will cause its wiring information to be drawn in blue arrows from the data items monitored to the objective.
- over a top-level input port that is controlled by a virtual source (see Section 9.1.3) will show a blue arrow indicating the virtual source block that is controlling this input port.
- over any block or port will cause the name and a brief description to appear.
- over a Stateflow transition segment will cause the segment and its label to be highlighted in yellow.

**Clicking...**
- on a signal line in a Simulink system highlights the signal in yellow. This makes it easy to trace a signal back to the point where it was generated and forwards to the point where it is consumed (i.e., the signal enters a block which uses it to compute a new value). The highlighting flows through (backward and forward) the following blocks which do not modify the signal’s value: Inport, Outport, Subsystem, From, Goto, Data Store Write, Data Store Read.
- in empty space removes signal highlighting.

**Double Clicking...**
- on a Simulink block will display the block’s parameters.
- on a Simulink subsystem will cause the subsystem diagram to be displayed in the main panel.
- on a Stateflow state will cause the state’s diagram to be displayed in the main panel.
- on a Simulink S-Function block with an associated .rsm file (for more details about .rsm files see Chapter 16) will cause the .rsm file to be displayed in the main panel.
- on a From, Goto or Goto Tag Visibility block will open a dialog listing all matching From and Goto blocks in your model (see Section 4.3).
- on a Data Store Read/Write/Memory block will open a dialog listing all matching blocks in your model (see Section 4.3).
- on a top-level input port will bring up a type editor dialog to modify that port’s type (see Section 5.3.1).
- on a configuration variable in the Configuration Variable Panel (see Section 4.4) will bring up a type editor dialog to modify that variable’s type.
- on a Validator expression objective will open the parameter dialog for that objective.

**Right Clicking...**  
Causes different pop-up menus to be displayed. The contents of the menu vary based on where the click occurs and whether or not Simulator is enabled. A summary of the menu items available when Simulator is disabled follows. For descriptions of the menu entries available when Simulator is enabled, see Section 7.1.
### CHAPTER 4. THE REACTIS TOP-LEVEL WINDOW

<table>
<thead>
<tr>
<th>Right-Click Location</th>
<th>Menu Entries (when Simulator is disabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink signals, Simulink blocks, Stateflow variables</td>
<td><strong>Add To Test Points</strong> Add the data item to the list of test points which may be associated with a model, stored in the model’s <code>.rsi</code> file, and viewed from the Reactis Info File Editor. Test points are internal data items (Simulink signals or Stateflow variables) that Reactis subjects to user-specified checks during simulation and test generation. For example, Reactis can be configured to record values for test points in test suites and have Reactis Simulator flag any differences between the values computed by a model for a test point and those stored in a test suite. Additionally, a type constraint can be assigned to a test point and Reactis can be configured to monitor if a test point is ever assigned a value outside the set of expected values specified by the constraint. For more details on test points please see Section 5.5.</td>
</tr>
<tr>
<td>White space in Stateflow state or Simulink subsystem</td>
<td><strong>Paste</strong> Insert the user-defined target or assertion from the clipboard at the current mouse location. <strong>Add User-Defined Target</strong> Create new user defined target (see section 9.3). <strong>Add Assertion</strong> Create new Assertion (see section 9.3).</td>
</tr>
<tr>
<td>White space at top-level of model</td>
<td><strong>Paste</strong> Insert the user-defined target, assertion, or virtual source from the clipboard at the current mouse location. <strong>Add User Defined Target</strong> Create new user defined target (see section 9.3). <strong>Add Assertion</strong> Create new Assertion (see section 9.3). <strong>Add Virtual Source</strong> Create new virtual source (see section 9.3). <strong>Unpack Bus Inports</strong> Create a wrapper model that contains a separate scalar inport for each element of a top-level bus input (see section 4.6).</td>
</tr>
<tr>
<td>Validator objective (i.e. user-defined target, assertion, or virtual source)</td>
<td><strong>Cut</strong> Cut the objective and copy it to the clipboard. <strong>Copy</strong> Copy the objective to the clipboard. <strong>Edit Properties...</strong> Edit properties of the objective. (see section 9.3). <strong>Remove</strong> Remove the objective from the model. <strong>Enable/Disable</strong> Enable or disable the objective.</td>
</tr>
<tr>
<td>Simulink top-level input port that is wired to virtual source</td>
<td><strong>Enable/Disable control by virtual source</strong> Enable or disable whether this port should be controlled by its associated virtual source. If control is disabled, Reactis will control the input values to the port directly. If control is enabled, the input values generated by the associated virtual source will be used.</td>
</tr>
</tbody>
</table>
4.1. LABELED WINDOW ITEMS

<table>
<thead>
<tr>
<th>Right-Click Location</th>
<th>Menu Entries (when Simulator is disabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink S-Function block</td>
<td>These items are visible only if “Enable white-box analysis of C code” is enabled in the “Reactis for C” pane of the Reactis global settings dialog. Note that you will need a license for the Reactis for C Plugin in order to use this functionality. For more information see Chapter 16.</td>
</tr>
<tr>
<td>Assign RSM File...</td>
<td>Create a new RSM file or assign an existing RSM file to this S-Function.</td>
</tr>
<tr>
<td>Edit RSM File...</td>
<td>Edit the RSM file assigned to this S-Function.</td>
</tr>
<tr>
<td>Remove RSM File...</td>
<td>Remove the assignment of an RSM file to this S-Function.</td>
</tr>
<tr>
<td>Simulink block</td>
<td>View Block Parameters... Display Simulink block parameters.</td>
</tr>
<tr>
<td>Top-level input port or configuration variable in Configuration Variable Panel (see Section 4.4)</td>
<td>Edit Type... Modify type of top-level input port or configuration variable (see Section 5.3.1).</td>
</tr>
<tr>
<td>Simulink subsystem or Stateflow diagram</td>
<td>Extract Subsystem... Extract a subsystem and save it in a separate model file (see section 4.5). Selecting this item causes the selected subsystem and any components related to the triggering of the subsystem to be extracted from the model and inserted into a new model. Reactis may then be applied to the newly extracted subsystem. Usage of this feature is described in more detail below in Section 4.5.</td>
</tr>
<tr>
<td>View Block Parameters...</td>
<td>Display Simulink block parameters.</td>
</tr>
<tr>
<td>Model Reference block</td>
<td>Open Model Load the referenced model in a separate Reactis window and using the referenced model’s .rsi file.</td>
</tr>
</tbody>
</table>

When a diagram is too big to display completely in the main panel, scroll bars appear for repositioning the diagram. Alternatively the diagram may be repositioned by left-clicking and dragging in the panel or by using the cursor keys. If your mouse includes a scroll wheel, then you may scroll vertically by clicking in the main panel and then using the scroll wheel.

Left-clicking and dragging in the panel while holding down the control key defines a “print region” that can be used for printing parts of a model. The print region is represented as a shaded blue box; it may be removed using the File → Remove Print Region menu entry or pressing the escape key.

Finally, pressing the F2 key within this panel causes the parent of the currently displayed subsystem to be displayed, pressing F3 zooms in, and pressing F4 zooms out.

3. Load a new model into Reactis.

4. Undo an operation (add, edit, remove, move) on a Validator objective. See Chapter 9 for a description of Validator objectives.

5. Redo last undone operation (add, edit, remove, move) on a Validator objective. See Chapter 9 for a description of Validator objectives.
6. Zoom in.
7. Zoom out.
8. Fit to page.
9. Go back in the history of displayed subsystems.
10. Go forward in the history of displayed subsystems.
11. Display the parent of the currently displayed subsystem.
12. Disable Reactis Simulator.
13. Enable Reactis Simulator.

4.2 Menus

This section describes the top-level menu items available when Simulator is disabled. The menu items available when Simulator is enabled are described in Chapter 7.

Some menu entries also have keyboard shortcuts that enable the relevant operations to be invoked from the keyboard. These shortcuts are displayed to the right of the relevant entries in the menus.

**File menu.** The file menu contains the following entries.

- **Open Model...** Load a new model into Reactis.
- **Close Model.** Close the currently displayed model.
- **Reload Model.** Reload the currently displayed model.
- **Select Info File...** Specify a Reactis Info File (.rsi file) to be used with the current model. See Chapter 5 for a discussion of editing .rsi files with the Reactis Info File Editor. .rsi files store information that Reactis associates with a model including import constraints, configuration variable settings, Validator objectives, outport tolerances, and other settings.
- **Extract Info File...** Extract an .rsi file from an .rtp file. Reactis Tester may be configured to store launch parameters and the .rsi file used for a given run in a Reactis Tester Parameter file (.rtp file). Selecting this menu item retrieves the .rsi file from the .rtp file.
- **Save Info File.** Save the current .rsi file.
- **Save Info File As...** Rename and save the current .rsi file.
- **Print...** Open a print dialog for model printing. Section 4.7 explains this feature in more detail.
- **Remove Print Region.** Clear the selected printing region in the main panel. You may select a region of a model for printing by left-clicking and dragging in the panel while holding down the control key. The resulting selection is highlighted within a blue box. Selecting this menu item removes the blue box.
- **Global Settings...** Opens dialog to adjust Reactis global settings. Section 4.8 describes the use of this dialog.
4.2. MENUS

Default Model-Specific Settings... Opens dialog to specify the default model-specific settings to be used when creating new .rsi files. The dialog includes settings that specify:

- how a model executes (e.g. conditional input branch execution and short circuiting),
- error checking (e.g. flagging overflows or NaN values),
- coverage settings that specify which coverage metrics are used for a model and how the coverage metrics are configured (e.g. multi-block versus traditional MC/DC),
- C code settings.

Whenever a new .rsi file is created, these settings are imported into the model-specific settings of the new .rsi file. Subsequently the model-specific settings for a model are modified using the Info File Editor. See Section 4.9 for more details on the Default Model-Specific Settings dialog.

Exit Reactis. Exit Reactis.

Edit menu. This menu includes entries used to manipulate .rsi files and an entry to launch a model search function. .rsi files contain constraints on the values assumed by top-level inports, details related to Validator objectives, and other model information maintained by Reactis. Note that .rsi files may be modified only when Simulator is disabled. Therefore, when Simulator is enabled the first five menu items are disabled, and the last ten launch the Info File Editor in a read-only mode (the information may be viewed but not changed).

Undo. Undo an operation (add, edit, remove, move) on a Validator objective.

Redo. Redo last undone operation (add, edit, remove, move) on a Validator objective.

Cut. Cut the currently selected Validator objective and place it in the clipboard.

Copy. Copy the currently selected Validator objective to the clipboard.

Paste. Paste a Validator objective from the clipboard to the current subsystem. To paste an objective to a specific position, right-click on that position in your model and select Paste from the context menu.

Find... Perform a text search of your model for strings matching a pattern you specify. Two types of patterns are currently supported. If the search pattern includes a colon (:), then the text before the colon corresponds to a Simulink block parameter name and the text after the colon corresponds to a value for that parameter. For example the pattern BlockType: Inport will initiate a search for all input ports in the model. If the search string contains no colon, then the search will examine Simulink block names, Stateflow state names and actions, Validator objective names, configuration variable names, and C code (if you are using the Reactis for C Plugin).

The scope of the search (the parts of the model examined for a match) is determined by the subsystem displayed when the search is launched. The scope consists of the currently displayed subsystem and all of its descendants (child subsystems, their children and so forth). Note that the search scope does not change if the search pattern is modified. The scope changes only when the search dialog is dismissed and a new search is launched. To search the entire model select the top-level and launch a search.
The following twelve entries open the Info File Editor with the pane specified by the menu entry pre-selected. The Info File Editor is described in more detail in Chapter 5.

**Inport Types...** Constrain the values generated for top-level inports during test generation.

**Configuration Variables...** Specify workspace data items that may change in between tests but not during a test.

**Test Points...** Manipulate test points. Test points are internal data items (Simulink signals or Stateflow variables) that Reactis subjects to user-specified checks during simulation and test generation. For example, Reactis can be configured to record values for test points in test suites and have Reactis Simulator flag any differences between the values computed by a model for a test point and those stored in a test suite. Additionally, a type constraint can be assigned to a test point and Reactis can be configured to monitor if a test point is ever assigned a value outside the set of expected values specified by the constraint. For more details on test points please see Section 5.5.

**Outport Tolerances...** Specify a tolerance for each outport of a model. When executing a test suite on the model in Reactis Simulator, an outport tolerance specifies the maximum acceptable difference between the value computed by the model for the outport and the value stored in a test suite for the outport.

**General...** Specify model-specific settings related to how a model executes (e.g. conditional input branch execution and short circuiting).

**Error Checking...** Specify the set of error checks Reactis will employ (e.g. flagging overflows or NaN values).

**Coverage Metrics...** Specify the set of coverage metrics to be used when working with a model in Reactis. If a metric is disabled:

- the metric will not be targeted by Tester when generating tests, and
- Simulator will not include the targets from the criterion in the Coverage Summary dialog, the Coverage Report Browser, and the highlighting in the main panel.

**Excluded Coverage Targets...** View and/or modify the set of coverage targets which have been excluded from coverage tracking. When a target is excluded from coverage, it will not be targeted by Tester when generating tests, and Simulator will not include the target in the Coverage Summary dialog or the Coverage Report Browser. Targets which have been excluded from coverage are colored blue in the main panel.

**Validator Objectives...** Displays a centralized list of all Validator objectives in your model and allows you to monitor, edit, and remove them.

**C Code...** Displays a list of all locations in your model where C code is used. For more information, see Chapter 16.

**External EML Functions...** Allows you to add external .m files called from EML functions for coverage tracking and stepping through. For more information, see Section 17.2.

**Callbacks...** Specify fragments of MATLAB code to execute before and/or after a model is loaded. Note that these operations are distinct from the similar Simulink callbacks.
4.2. MENUS

Search Path... Specify the model-specific search path.
Dependencies... Specify files on which a model depends.

View menu. The following are the menu entries that are enabled when Simulator is disabled. The other elements of the menu are only enabled when Simulator is enabled. See Section 7.2 for descriptions of these items.

Back. Go back in the history of displayed subsystems.
Forward. Go forward in the history of displayed subsystems.
Go to Parent. Cause the parent of the currently displayed subsystem to be displayed in the main panel.
Zoom In. Zoom in the display of the model in the main panel.
Zoom Out. Zoom out the display of the model in the main panel.
Zoom to Fit. Fit to page.
Expand Tree. Causes the entire tree in the model hierarchy panel to be expanded.
Collapse Tree. Causes the entire tree in the model hierarchy panel to be collapsed.
Hide Empty Subsystems. If checked, subsystems that do not contain any blocks, lines or annotations will not be included in the hierarchy tree.
Get Model Statistics Causes Reactis to invoke MATLAB, collect various statistics about the current model’s size and display them.
Select Label Font... Select font for labels in Simulink / Stateflow diagrams.
Increase Label Font Size. Increase size of font for labels in Simulink / Stateflow diagrams.
Decrease Label Font Size. Decrease size of font for labels in Simulink / Stateflow diagrams.
Select C Source Font... Select font for displaying C source code in the main panel when using Reactis for C Plugin.
Select Line Styles... Select styles and colors for drawing various Simulink / Stateflow diagram items.
Show Recent Errors... Display recent error messages encountered for the current model.
Clear Recent Errors. Remove all entries from the list of recent errors.

Simulate menu. Only one menu entry is enabled when Simulator is disabled, and it is described below. The remaining elements of the menu are only enabled when Simulator is enabled, and are described in Section 7.2.

Simulator on/off. Toggles the state of Simulator between enabled and disabled. Implements the same behavior as labeled window items 12 and 13 in Figure 4.1.

Test Suite menu. The following menu entries are enabled when Simulator is disabled. The remaining elements of the menu are only enabled when Simulator is enabled, and are described in Section 7.2.

Create... Launch Reactis Tester. See Chapter 8 for details.
Browse... Launches a file-selection dialog which is used to select a test suite to be viewed with the Test-Suite Browser. See Chapter 11 for details.

Validate menu. Since Validator objectives may only be modified when Simulator is disabled, the first six entries are disabled when Simulator is active. Depending on the state of the main panel, these entries might be enabled when Simulator is disabled. See Chapter 9 for more details.

Add Assertion Add an assertion to the system currently displayed in the main panel. The sub-menu enables you to specify whether the assertion should be an expression, a diagram, or a timer. This menu item is enabled when the subsystem currently displayed in the main panel is a Simulink diagram, but disabled when it is Stateflow diagram. To add assertions into Stateflow diagrams, right-click in the diagram.

Add User-Defined Target Add a user-defined target to the system currently displayed in the main panel. The sub-menu enables you to specify whether the target should be an expression, a diagram, or a timer. This menu item is enabled when the subsystem currently displayed in the main panel is a Simulink diagram, but disabled when it is a Stateflow diagram. To add user-defined targets into Stateflow diagrams, right-click in the diagram.

Add Virtual Source Add a virtual source. The sub-menu enables you to specify whether the target should be an expression or a diagram. This menu item is enabled when the subsystem currently displayed is the top level system, but disabled when the focus is at a deeper level.

Edit Objective... Edit the currently selected Validator objective. This menu item is disabled if no objective is selected.

Remove Objective. Remove the currently selected Validator objective. This menu item is disabled if no objective is selected.

Disable/Enable Objective. Disable or enable the currently selected Validator objective. This menu item is disabled if no objective is selected.

Check Assertions... Launch Reactis Validator to search for assertion violations. See Chapter 9 for details.

Coverage menu. No items are enabled when Simulator is disabled.

Window menu. Lets you switch to any model currently loaded in Reactis.

Help menu. The Help menu contains the following entries.

Contents. Display the table of contents for the documentation.

Index. Display the index of topics covered in the documentation.

Frequently Asked Questions. Go to the list of frequently asked questions.

Release Notes. Display the release notes for the current Reactis version.

Top Level Window. Go to the section of the documentation that describes the Reactis top-level window.

Check for Updates... Check the Reactive Systems website to see if a newer version of Reactis is available.
4.3 List of Matching From/Goto or Data Store Blocks

This dialog comes up after double-clicking on a From, Goto, Goto Tag Visibility or Data Store Read/Write/Memory block in Reactis. It lists all other such blocks in your model that match the double-clicked block.

In this dialog you can:

- Double-click on a row to have Reactis highlight the block in the main panel. If the block is located in a different subsystem than the one currently displayed, Reactis will switch to that subsystem.
- Click on a column header to re-sort the table by the values of that column.
- Click on the Close button to close the dialog.

4.4 Configuration Variable Panel

If you have defined configuration variables for your model (see Section 5.4), Reactis will include an entry “Configuration Variables” in the top-level of the hierarchy panel. Clicking on this entry displays a block in the main panel for each currently defined configuration variable. This panel allows you to view and manipulate configuration-variable information.

Double-clicking on a configuration variable when Simulator is disabled opens the type editor dialog to specify a constraint for the configuration variable. When Simulator is enabled and at the start of a simulation run (no steps have been taken), double-clicking on the configuration variable opens a dialog to specify a value for the variable.

When Coverage → Show Details is selected in Simulator, information on boundary value coverage for configuration variables is conveyed in the panel. A block is highlighted in red if it has any uncovered boundary value targets. See Section 6.3.2.1 for a description of the boundary value targets associated with a configuration variable. Hovering over the variable in this panel when Simulator is enabled displays test and step in which exercised targets were covered.
4.5 Extracting Subsystems

It is sometimes useful to apply Reactis to individual subsystems within a larger model. For example, if a given model is closed-loop, with a controller connected to a plant, then isolating the controller subsystem may be needed in order to generate tests for the controller. In other cases, very large models might require subsystem-at-a-time analysis. Reactis enables such analysis by providing a facility to extract subsystems from models. This feature isolates a subsystem, optionally along with portions of the model involved in triggering the subsystem, and stores the result as a new model in a separate .slx file.

When extracting a subsystem, Reactis retains the hierarchical structure of the original model. That is, the original subsystem interfaces enclosing the extracted subsystem are retained, although the input and output ports connected to these interfaces are altered to coincide with those of the extracted subsystems. Retaining the model hierarchy in this fashion facilitates the inclusion of triggering mechanisms and Data Store Memory blocks in the extracted model.

The subsystem extraction utility is invoked by loading a model into Reactis, then right-clicking on the Simulink subsystem or Stateflow chart to be extracted and selecting the Extract Subsystem entry in the pop-up menu. This causes the Extract Subsystem dialog shown in Figure 4.3 to appear. Using the dialog you can specify the name of the file in which to store the extracted model and whether the triggering mechanism should be included.

The Port settings section lets you specify if information (type and sample rate) from the original model should be propagated to the top-level ports of the extracted subsystem. Note that when this information is not propagated, the types and sample rates inferred for the ports of the extracted subsystem may differ from those inferred in the original model (since type and sample rate information has been removed in the extracted model). When the propagation is enabled, for each port \( P \) of the subsystem being extracted, the extraction routine:

1. Traces the signal connected to \( P \) up one level in the hierarchy to a port \( P' \)
2. Records the type/rate inferred for \( P' \) in the original model
3. Explicitly sets the type/rate for \( P' \) in the extracted model

The following limitations apply:

- If the type of \( P' \) is a virtual bus then the type will not propagate.
- If \( P' \) has multiple sample rates or is triggered by an irregular mechanism such as a State-flow chart then the sample rate will not propagate.

When you click **Extract**, Reactis will extract the subsystem, save it under its new name, open a new Reactis window, and load the extracted model. Note that the extracted subsystem is saved as a standard .slx file, making it easy to edit the extracted model using Simulink if changes to new model are necessary.

If you select **Extract subsystem and triggering mechanism** then portions of the original model outside the subsystem of interest, but involved in the triggering of that subsystem will be included in the extracted model. More precisely, if the extracted subsystem

- is a triggered subsystem,
- is located within a triggered subsystem, or
- contains one or more triggered subsystems whose triggers are connected to something outside the extracted subsystem

then Reactis determines which portions of the model residing outside of the extracted subsystem should be retained in order to trigger the extracted subsystem properly. This ensures that the simulation times during which the extracted subsystem is executed match those of the subsystem before it was extracted.

If the extracted subsystem references Data Store Memory blocks located outside the extracted subsystem, Reactis will keep those Data Store Memory blocks. However, Reactis will not keep any Data Store Write blocks outside the extracted subsystem.

Note that the extraction tool can extract parts of a model even if Reactis reports errors when trying to run it. This enables Reactis to be used on models that contain Simulink features that are unsupported by Reactis, as long as those features are not used in the extracted subsystem. For example, one may extract a discrete-time controller subsystem from a model that includes a continuous-time plant.

---

### 4.6 Unpacking Top-Level Buses

If a model has bus inputs at the top level, this feature lets you generate a wrapper model that has no bus inputs, but has scalar inputs that feed into Bus Creator blocks that construct buses suitable for input to the original model which is referenced by a Model Reference block. The operation can be invoked by right-clicking in white space at the top level of the model and selecting **Unpack Bus Inports** to open the dialog shown in Figure 4.4. The generated wrapper model will be stored in a new .slx file named in the entry box of the dialog. Two checkboxes let you configure how the wrapper model is generated:

**Stub unused elements in top-level input buses** When checked, if an element of a bus is unused, no input port will be created for that element. Instead, the signal for that element will be connected to a Ground block.
**Hide stubbed elements in wrapper subsystem** When checked, create a wrapper subsystem around the Inport/Ground blocks so that the top-level system looks cleaner (since it contains only the scalar input ports and not the stubbed signals).

![Image of the dialog to unpack top-level buses into scalar inputs.](image)

Figure 4.4: The dialog to unpack top-level buses into scalar inputs.

---

### 4.7 Printing Models

![Image of the Print dialog.](image)

Figure 4.5: The Print dialog.

Reactis includes a flexible facility for printing models. Upon selecting menu item **File → Print...** the print dialog shown in Figure 4.5 appears. The radio buttons and check-boxes in the **Print Range** section of the dialog specify which portions of the model should be printed as follows.

**Selection.** Prints only the portion of the model located within the current print region in the main panel. If no print region is defined, this entry is disabled. A print region is selected by left-clicking and dragging in the main panel while holding down the control key. The print region is cleared by selecting menu item **File → Remove Print Region.**

**Current system.** Prints only the system currently displayed in the main panel.

**Current system and above.** Prints the system currently displayed in the main panel and all systems between the current system and the top-level system in the current model. Each such system is printed on a separate sheet of paper.
4.8. REACTIS GLOBAL SETTINGS

Current system and below. Prints the system currently displayed together with all its sub-systems, sub-subsystems, etc. Each system is printed on a separate sheet of paper.

Whole model. Prints the whole model.

Expand unique library links. When checked, library blocks referenced by the model will be printed. Note that each library block is printed only once, even though some blocks might be referenced multiple times by a model.

Look under masks. When checked, masked subsystems will be printed.

The following radio buttons and check-boxes in the Frame section specify whether a frame should be printed on each page, and if so what content should be included in the frame.

No frame. When checked, no frame is printed.

Frame. When checked, a frame is printed.

Include system name. When checked, the name of the system is printed in the upper left part of the page frame.

Include print date. When checked, the date the model was printed is included in the bottom left corner of the frame on each page.

Include page number. When checked, the page number is printed in the bottom right corner of the frame on each page.

Include file name. When checked, the name of the .slx file containing the model and the folder containing the file is printed in the bottom left corner of each page.

The remaining buttons in the print dialog work as follows.

Help. Display print dialog help.

Page Setup... Invokes a dialog that allows the user to specify paper size and margins, and whether printing should be portrait or landscape.

Preview. Open a viewer to display what will be printed.

Print... Begin printing.

Close. Close the dialog (and cancel printing).

---

4.8 Reactis Global Settings

Selecting File → Global Settings... invokes the Reactis Global Settings dialog \(^1\), which allows you to adjust the behavior of Reactis. The global settings are partitioned into seven tabbed panes each described in detail below: General, Reactis for C, Reactis for EML, MATLAB, Path, Files, User Info, and License.
4.8.1 General Settings

The General global settings pane shown in Figure 4.6 includes the following items.

**GUI Language.** Language used in the Reactis GUI.

**Documentation Language.** Language used for the in-tool Reactis help.

**Automatically check for updates (once a day).** Instructs Reactis to check once per day whether updates to Reactis are available for download. If updates are found you will be asked if you would like to download and install the patch. *Note: this feature can be disabled at install time; in which case this checkbox will not appear in the dialog.*

**Use Z3 solver.** Checking this setting improves the coverage of Tester-generated tests for some models and also improves the static analysis that identifies unreachable coverage targets.

**Enable logging.** Enable logging, specify a log level, and indicate the file to which the log should be written. Note that logging degrades performance and can create very large log files; therefore, it is typically only used to diagnose problems. The log level string will be provided by the Reactis support team if you are asked to create a log file.

---

Prior to V2012, the Global Settings dialog was called the Settings dialog and included a number of additional settings that are now model-specific settings manipulated via the Reactis Info File Editor.
4.8.2 Reactis for C Global Settings

This pane is used to enable and configure Reactis for C Plugin. Please see Section 16.2 for details.

4.8.3 Reactis for EML Global Settings

This pane is used to enable and configure Reactis for EML Plugin. Please see Chapter 17 for details.

4.8.4 MATLAB Global Settings

Reactis sometimes invokes MATLAB during simulation, test-generation, and validation. The MATLAB global settings pane, shown in Figure 4.7, enables you to configure some aspects of how Reactis invokes MATLAB.

![Image of MATLAB global settings pane]

Figure 4.7: The Global Settings dialog with the MATLAB pane selected.

*Use MATLAB version.* If you have multiple versions of MATLAB installed, this pull-down menu may be used to specify the version of MATLAB that Reactis will invoke. Each installation of MATLAB that Reactis automatically detects will have an entry in the menu. The menu includes two additional entries:
None specified (use Windows path) When this entry is selected, Reactis will invoke the version of MATLAB that appears first in the Windows path.

Custom When this entry is selected, Reactis will invoke the version of MATLAB that appears in the MATLAB root text entry box (just below this menu in the MATLAB pane of the global settings dialog).

Use MATLAB root. This text entry box becomes enabled when the Custom entry of the MATLAB version menu is selected. By entering a folder name (directly or by clicking the button to the right and using the file-selection dialog), you specify a folder containing the MATLAB installation which Reactis should use. This configuration is typically only used for a custom MATLAB installation that Reactis is unable to detect automatically.

Invoke MATLAB via: MATLAB supports several different mechanisms for third party applications to invoke it. Each method has advantages and disadvantages. This menu lets you configure which mechanism will be used by Reactis to invoke MATLAB. The options are:

MATLAB engine interface via C API This option works with older versions of MATLAB, but requires administrator privileges to switch between MATLAB versions.

Windows COM interface This option does not require administrator privileges to switch between MATLAB versions, but:

- does not work with R2006b or earlier,
- is partially compatible with R2007b through R2011b (you can switch between only one MATLAB installation in this range and newer versions),
- is fully compatible with R2012a and later.

Full MATLAB window This option will start a full MATLAB session to communicate with. It is a useful option if other methods of invoking MATLAB fail but will require more time for MATLAB to start up.

Full MATLAB window (hidden) Similar to the previous option but hides the MATLAB window it is communicating with. This is less distracting but for some models MATLAB will not work properly if the window is hidden.

MATLAB start up timeout (seconds). When Reactis invokes MATLAB, it will wait this long for a response indicating a successful invocation before assuming that MATLAB will not start properly. This setting is unavailable when using the Windows COM interface.

Reuse existing MATLAB command windows. When Reactis invokes MATLAB, it will use this option to determine whether to reuse to existing MATLAB instances, or invoke new ones. This may help compatibility in some situations. This setting not available if “Full MATLAB window” is selected. In that case Reactis always invokes a new MATLAB instance.

4.8.5 Path Global Settings

The Path pane of the Global Settings dialog, shown in Figure 4.8, enables you to specify the list of folders in which Reactis will search for files such as Simulink model libraries (.mdl), MATLAB scripts (.m), and S-Functions (.dll, .mexw32, .m). The order in which folders are listed in the dialog specifies the search order (from top to bottom).

Note that Reactis also gives users the capability to define model-specific search paths which consist of a list of folders to be searched when loading a given model. The model-specific path is set using the Reactis Info File Editor as described in Chapter 5. When searching
for files, the complete search path is constructed by prepending the model-specific path to the global path.

The buttons labeled in the figure work as follows.

1. Add a new folder to the list.
2. Open a dialog to edit the currently selected folder.
3. Remove the currently selected folder(s) from the list.
4. Select all folders in the list.
5. Copy the currently selected folder(s) to the clipboard.
6. Paste from the clipboard to the list.
7. Invoke MATLAB, query the MATLAB path, and add each folder in the MATLAB path to the list.
8. Move the currently selected folder up one spot in the list.
9. Move the currently selected folder down one spot in the list.
10. Move the currently selected folder to the top of the list.
11. Move the currently selected folder to the bottom of the list.
12. Produce warning if multiple instances of a library are found in the search path.

4.8.6 Files Global Settings

The Files pane, shown in Figure 4.9, lets you specify the folder where Reactis should store files that it creates. These files include Reactis Info (.rsi), Reactis Profile (.rsp), and Reactis Cache (.mwi) files. The setting also determines the default location (which you can override) for Reactis test suite (.rst) and coverage report (.html) files.

If you choose to specify a subfolder, then the folder name entered in the textbox is relative to the model folder. The special string %MODELNAME% may be included in the subfolder name and will be replaced by the base name of the model file.

For example, for a model cruise.mdl located in a folder \models, a subfolder string of reactis\%MODELNAME% will cause the files associated with cruise.mdl to be stored in

\models\reactis\cruise

4.8.7 User Info Settings

The User Info pane is shown in Figure 4.10. When Reactis is configured to use a remote license server as described in the next section, information contained in this panel is submitted to the server when Reactis is started, and available to all users who have access to the server.

The list of users occupying licenses at a given time may be obtained using the License pane of the Global Settings dialog as described below, or using the standalone License Monitor utility included in the Reactis distribution. This utility may be invoked by selecting Reactis V2020 → License Manager → License Monitor from the Windows Start menu.

4.8.8 License Settings

The License pane, shown in Figure 4.11, enables you to query and specify license configuration information. The first two sections display the MAC address of the machine on which Reactis is running and the location of a local license file if one is in use.

The third section of the pane displays a list of servers running the Reactis License Manager. When Reactis is invoked, this list will be searched from top to bottom for an available license. The lowest portion of the pane displays a list of users currently using licenses for the License Manager/product currently selected License servers list.

Each of the window items labeled in Figure 4.11 is interpreted and used as follows.

1. Information about the contents of the local license file. If there is a problem with the license file, then a description of the error condition is listed here. If no problem exists, then a list of licensed products and their expiration dates is given.

2. This is the list of servers running the Reactis License Manager. Each entry in the list includes the following:
4.8. REACTIS GLOBAL SETTINGS

Figure 4.9: The Files pane of the Global Settings dialog lets you specify the folder where Reactis should store any files that it creates.

Host  The name or IP address of the server running the License Manager.

Status  The status of the connection to the License Manager.

For each product managed by the server:

Product  Name of the product (Reactis, Reactis for C Plugin, or Reactis for C).

Total  The total number of licenses for the product.

In Use  The number of currently occupied licenses for the product.

3. Add a new License Manager to the list.

4. Remove the currently selected License Manager from the list.

5. Move the currently selected License Manager up one spot in the list, down one spot in the list, to the top of the list, or to the bottom of the list. These buttons are only enabled when the list contains more than one License Manager.

6. Information regarding the currently selected License Manager is displayed here. If there is a problem with the connection to the License Manager, then a description of the error condition is listed here. If no problem exists, then for each license currently occupied, this section lists:
CHAPTER 4. THE REACTIS TOP-LEVEL WINDOW

Figure 4.10: The Global Settings dialog with the User Info pane selected.

**IP Address.** The IP address of the computer on which the Reactis application occupying the license is running.

**Name.** The contents of the Name field in the User Info pane of the person occupying the license.

**Phone.** The contents of the Phone field in the User Info pane of the person occupying the license.

**Duration.** The length of the time this computer has been holding the license.

### 4.9 Default Model-Specific Settings

The Default Model-Specific Settings dialog lets you specify default settings to be used when creating new .rsi files. Whenever a new .rsi file is created, these settings are imported into the new .rsi file. Subsequently the settings for a model are modified using the Info File Editor. The dialog includes the following tabbed panes:

**General** various settings primarily related to how a model executes (e.g. conditional input branch execution, short circuiting, etc.)

**Error Checking** what types of errors should be flagged (overflows, NaN values, etc.)
4.9. DEFAULT MODEL-SPECIFIC SETTINGS

Figure 4.11: The Global Settings dialog with the License pane selected.

Coverage settings that specify which coverage metrics are used for a model and how the coverage metrics are configured (e.g. multi-block versus traditional MC/DC).

C Code settings to configure the Reactis for C Plugin.

See Chapter 5 for a description of each setting.
Chapter 5

The Reactis Info File Editor

Reactis does not modify the .slx file for a model. Instead it stores any model-specific information it requires in a .rsi file. The primary way to view and edit the data in an .rsi file is via the Reactis Info File Editor, as described in this chapter.

The information stored in the .rsi file is grouped into twelve categories. Each category is controlled from a different pane of the Info File Editor. These categories (described in detail in subsequent sections) are named as follows:

**Inport Types.** This panel contains constraints on the values that the inports may assume during simulation, test generation, and validation.

**Configuration Variables.** This panel lists which workspace or data dictionary data items will be used as configuration variables.

**Test Points.** This panel lists model data items which will be used test points during testing.

**Outport Tolerances.** This panel specifies the tolerance for Simulator to use when comparing an outport value computed by the model against the corresponding value in a test suite.

**General.** This panel contains parameters which control how a model executes in Reactis (e.g. conditional input branch execution, short circuiting, etc.).

**Error Checking.** This panel specifies how Reactis will respond to various types of errors (e.g. overflow, NaN, etc.).

**Coverage Metrics.** This panel specifies the coverage metrics to be used when working with a model.

**Excluded Coverage Targets.** This panel lists targets to be ignored when measuring the coverage of a model.

**Validator Objectives.** This panel lists all Validator objectives (assertions, user-defined targets, virtual sources).

**C Code.** If using the Reactis for C Plugin, this panel will contain a list of places C code is used in the model under test (S-Functions and Stateflow custom code) and settings which control the white-box analysis of each S-Function.

**External EML Functions.** This panel lists .m files which contain Embedded MATLAB functions called from the model (e.g. from MATLAB Function blocks, Stateflow, Truth Tables).
Callbacks. This panel specifies callbacks to be executed before and/or after a model is loaded.

Search Path. This panel contains the model-specific search path.

File Dependencies. This panel lists files on which the model depends.

Note that the contents of an .rsi file may only be modified when Simulator is disabled. When Simulator is running, the Info File Editor operates in read-only mode as indicated by [read only] in the editor window’s title bar.

Although not necessary, the default naming convention assumes that the .slx file and .rsi file of a model share the same base name; for example, if the model file is named cruise.slx, then the name of the associated .rsi file is assumed to be cruise.rsi. An rsi-file named differently may be associated with a model by loading the model in Reactis and selecting File → Select Info File...

5.1 Menus

The menu entries of the Info File Editor are described in this section. Note, that the set of enabled entries and the meaning of some entries vary depending on which pane is currently displayed.

File menu. The File menu contains the following entries.

Save  Save the current .rsi file.

Save As... Rename and save the current .rsi file.

Exit. Exit Info File Editor.

Edit menu. The Edit menu contains the following entries. Entries are enabled or disabled according to the currently selected pane.

Undo. Undo the last edit operation.

Redo. Redo the last undone edit operation.

Cut. Delete the currently selected item and place it into the clipboard.

Copy. Copy the selected item into the clipboard.

Paste. Paste the item in the clipboard into the selected location.

Add... When the Configuration Variables pane is selected, this menu item is enabled and allows you to add a new configuration variable. When the menu item is selected you are presented with a list of variables from which to chose. The elements of the list are:

- Variables initialized within the Simulink or Reactis pre-load callback of the model, e.g. by executing .m file script(s) or loading .mat file(s).
- Variables defined in a Simulink data dictionary attached to the model. Data dictionary variables are shown as [dictionaryname].[variablename].

When the External EML Functions pane is selected, this menu item is enabled and lets you add an external EML function that resides in an .m file and is called by EML code within the model (e.g a MATLAB Function block).
When the Search Path pane is selected, this menu item is enabled and allows you to add an entry to the model-specific search path.

When the File Dependencies pane is selected, this menu item is enabled and allows you to add a new file to the list of files on which the model depends. When the menu item is selected, a file selection dialog appears, allowing you to specify a file.

**Edit...** Edit the currently selected item. The panes for which this menu entry is enabled include Inport Types, Configuration Variables, Test Points, Outport Tolerances, Validator Objectives, C Code, and Search Path. In the case of inport types and configuration variables the dialog shown in Figure 5.3 is used for editing. The edit dialogs for Validator objectives and C code are described in Chapters 9 and 16 respectively. The entry is disabled when one of the other panes is selected.

**Edit Text...** Edit the type of the currently selected port or configuration variable by directly entering a textual type specification according to the syntax defined in Section 5.3.2. The entry is disabled when one of the other panes is selected.

**Remove** Remove the currently selected item.

**Tools.** The availability and contents of this menu vary depending on the currently displayed pane. The panes that offer a Tools menu are below along with the entries available with each pane.

**Inport Types Pane** Synchronize, Inports, Outports and Test Points The current .rsi file is modified as follows. Inports, outports, or test points deleted from the .slx file will have their entry in the .rsi file removed; new inports, outports, or test points in the .slx file will have entries introduced in the .rsi file with the full range of the base data type. If an item exists in both the .slx file and .rsi file and the base data type differs between the two, Reactis will display a dialog to resolve the type conflict as shown in Figure 5.1. If minimum or maximum values for the inport are given in the model then those are also imported. Reactis recognizes the following settings:
- Minimum/Maximum set in the input port’s properties dialog.
- Minimum/Maximum set via a Simulink.Signal object attached to the signal line connected to the input port.
- Minimum/Maximum set for bus elements in a Simulink.Bus object that is set as the data type for the input port.

Synchronize Inport Ranges with TargetLink Data Dictionary Each inport type in the .rsi file is constrained to match the corresponding TargetLink production code type and Min/Max limits. If the TargetLink type is numeric, the range of the inport type is constrained to match the TargetLink Min/Max limits. If the TargetLink type is integer or fixed-point, the resolution of the inport type is set to match the resolution of the TargetLink type. If the TargetLink type is Boolean, the inport type is constrained to the set \{0, 1\}.

Reset to Defaults Revert the type of each port to the type inferred for it in the .slx file (i.e. drop any added constraints such as ranges, enumerated sets, probabilities, delta constraints, etc.).

**Test Points Pane** Synchronize, Inports, Outports and Test Points See description above of this menu item for Inport Types Pane.
CHAPTER 5. THE REACTIS INFO FILE EDITOR

Import from Simulink  If the model includes any signals specified as test points within Simulink, then import these Simulink test points as Reactis test points. It is also possible to import Simulink signals with a Simulink.Signal object attached as Reactis test points.

Outport Tolerances Pane  Synchronize, Inports, Outports and Test Points  See description above of this menu item for Inport Types Pane.

Reset Outports to Defaults  Revert each outport tolerance to Inherit.

General, Error Checking, Coverage Metrics, C Code Panes  Reset to Defaults  Reset the model-specific settings in the current pane to the values specified in the Default Model-Specific Settings dialog (invoked via File → Default Model-Specific Settings).

Help menu.

Contents. Go to the table of contents in the documentation.

Index. Go to the index in the documentation.

Info File Editor. Display Info File Editor help.

![Figure 5.1: The type conflict resolution dialog.](image)

5.2 Toolbar

The following buttons are available in the Info File Editor toolbar. Note that whether a button is enabled or not may depend on which pane is currently active.

- Save any changes to the currently loaded .rsi file.
- Equivalent to menu item Edit → Add....
- Equivalent to menu item Edit → Edit....
- Equivalent to menu item Edit → Remove.
- Cut the selected item and copy it to the clipboard.
- Copy the selected item into the clipboard.
- Paste the item in the clipboard into the selected row.
- Undo last editing operation.
- Redo last undone edit operation.
- Get help.
5.3 Inport Types Pane

Inport type constraints are used by Reactis to constrain the set of values fed into top-level inports during simulation, test generation, and model validation. The types that may be used are: base Simulink/Stateflow types extended by notation to specify ranges and subsets within a base type, rate-of-change (or delta) constraints, resolution constraints, and constraints on vector and bus elements.

The Info File Editor with the Inport Types pane selected is shown in Figure 5.2. The pane displays a row for each top-level inport of your model. Double-clicking on a row opens the type editor dialog for the item in the row. As detailed in the following section, the dialog offers a user-friendly interface to construct a type constraint for the input.

Figure 5.2: The Reactis Info File Editor with the Inport Types pane selected.
5.3.1 The Type Editor Dialog

The Type Editor Dialog, shown in Figure 5.3, helps you construct type constraints for top-level inports (as well as for configuration variables and test points). These types constrain the set of values Reactis will assign to inports during testing. In the following discussion, we use TUC to denote the *type under construction*, i.e. the type being defined with the dialog.

![Figure 5.3: The dialog for specifying type constraints for inports, configuration variables, or test points. As shown here a default type of `double {0.0, 1.0}` is specified for port `onOff`, meaning that during random simulation, after time zero, `onOff` will be assigned the values 0.0 and 1.0 with equal probability. The list of conditions (item 1) lets you specify different constraints be used at different times. As shown, when simulation time is zero (indicated by the first condition `t == 0`), a constraint is specified that is different from the currently displayed default.](image)

5.3.1.1 Invoking the Type Editor

The Type Editor may be invoked for an inport in several different ways:

- From the main panel of the Reactis window, by either:
  - double-clicking on a top-level inport, or
  - right-clicking on a top-level inport and selecting *Edit Type*, or...
5.3. **IMPORT TYPES PAN**E

- From the **Import Types** pane of the Reactis Info File Editor, by either:
  - double clicking on an import row, or
  - selecting an import row with a single click, and then right-clicking in the body of the Info File Editor and selecting **Edit Type**, or selecting menu item **Edit → Edit...**

### 5.3.1.2 Using the Type Editor

A type may be constructed with the dialog by performing the following steps. Window items refer to the labels in Figure 5.3. Recall that TUC denotes *type under construction* (for the currently selected condition), i.e. the type being defined with the dialog.

1. Should the type constraint stay the same throughout a simulation or should different constraints apply at different simulation times? The condition list (window item 1) lets you specify this using a list of conditions. Each condition is a simple boolean expression that can reference the current simulation time denoted as \( t \). To determine the constraint to be used at a given simulation time, Reactis evaluates the condition expressions in the list from top to bottom. The constraint associated with the first expression that evaluates to true will be used. The final element of the list default will be used if no other condition evaluates to true. Add, edit, remove, or reorder the conditions using the buttons under the list (window item 2). Clicking on an a condition in the list will cause the constraint associated with that condition to be displayed in the panel to the right of the condition list.

2. Should the type include all values in the base type, only a subrange of the base type, or only a set of specific values in the base type?
   - **All values in base type.** Select the Complete range of base type radio button (window item 5) and proceed to step 3.
   - **Only a subrange of the base type.** Select the Subrange of base type radio button (window item 6) and specify a lower and upper bound for the range using window items 7 and 8 respectively. If the port has a base type of double or single, then you may specify a *resolution* for the values in the TUC. To do so click the Resolution check box (window item 9) and enter a resolution value in the text box to the right (window item 10). When a resolution is specified, a variable of the TUC will only assume values that differ from the lower bound specified in window item 7 by some multiple of the resolution value in window item 10. Proceed to step 3.
   - **A specific set of values in the base type.** Select the Set of specific values radio button (window item 11) and use the Add, Edit, and Remove buttons (window item 14) to create a list of values. Each value must be an element of the base type of the TUC and may be assigned a probability weight. See Section 5.3.2 for a description of probability weights. Proceed to step 3.

3. Do you wish to constrain how the values arriving over the inport may change from one step to the next during simulation?
   - **No.** Select the No limitation radio button (window item 16) and proceed to step 4.
   - **Yes.** Select the Limits radio button (window item 17) and enter values in the Minimum and Maximum text entry boxes (window items 18 and 19). If \( \text{min} \) and \( \text{max} \) are the
values specified here, then values arriving over ports with the TUC will be con-
strained as follows. If \( v_1 \) and \( v_2 \) are values for such a port in consecutive simulation
steps, then \( v_1 + \min \leq v_2 \leq v_1 + \max \). Proceed to step 4.

4. When you are satisfied with the type you have constructed click the Ok button to update
type and dismiss the Type Editor dialog.

5. Select File → Save to save your changes to the .rsi file.

5.3.1.3 Labeled Window Items

This section describes the labeled items in Figure 5.3. Recall that TUC denotes type under
construction (for the currently selected condition), i.e. the type being defined with the dialog.

1. Condition list that lets you specify different constraints for different simulation times.
Each condition in the list is a simple boolean expression that can reference the current
simulation time (denoted as \( t \)). When a condition is selected, the constraint associated
with that condition is displayed in the right portion of the dialog.

To determine the constraint to be used at a given simulation time, Reactis evaluates the
condition expressions in the list from top to bottom. The constraint associated with the
first expression that evaluates to true will be used. The final element of the list default
will be used if no other condition in the list evaluates to true.

Conditions are constructed from relational operators (\(<, \leq, =, \neq, >, \geq\) ), logical
operators (\(\&\&\), ||, !), the current simulation time \( t \), and time values (non-negative
double values). The complete syntax of conditions is defined by the \( timeExpr \) variable
in the grammar in Section 5.3.2. The following table gives some sample conditions:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t == 0 )</td>
<td>Simulation time is zero.</td>
</tr>
<tr>
<td>((t &gt; 1) &amp;&amp; (t &lt; 1.5))</td>
<td>Simulation time is greater than 1 and less than 1.5.</td>
</tr>
</tbody>
</table>

2. Buttons used to add, edit, remove or reorder elements of the condition list.

3. The textual version (as described in Section 5.3.2) of the TUC is displayed here. The
string here may not be edited; it is read-only. It is updated by Reactis when you click the
Apply button.

4. The radio buttons in this group (window items 5, 6, and 11) enable you to specify the
values from the base type to be include in the TUC.

5. Selecting this radio button indicates that all values from the base type should be included
in the TUC.

6. Selecting this radio button indicates that only a subrange of values from the base type
should be included in the TUC.

7. When the Subrange of base type radio button is selected, this entry box enables you to
specify a lower bound for the subrange.

8. When the Subrange of base type radio button is selected, this entry box enables you to
specify an upper bound for the subrange.
9. If the Subrange of base type radio button is selected and the TUC has a base type of double or single, then this check box may be checked in order to specify a resolution for the TUC.

10. When the Resolution check box (window item 7) is checked, a resolution value may be entered in this text box. When a resolution is specified, a variable of the TUC will only assume values that differ from the lower bound specified in window item 5 by some multiple of the resolution value here.

11. Selecting this radio button indicates that only a specified set of values from the base type should be included in the TUC. The buttons in window item 14 are used to specify the set of values.

12. The entries in this column specify the values in the TUC.

13. The entries in this column associate a probability weight with each value in the TUC. See Section 5.3.2 for a description of probability weights.

14. Buttons used to add, edit, or remove values to/from a set of specific values. Items are added using the dialog shown in Figure 5.4.

![Figure 5.4: Dialog for adding one or more values (with optional probability weight) to an enumerated set of values. The entry shown will add the values 0 and 1; with 0 having a probability weight of 1 and 1 having a probability weight of 3.](image)

15. The radio buttons in this group enable you to specify whether or not the type has a delta constraint. For an inport with the TUC, such a constraint limits the allowable changes in value from one simulation step to the next.

16. The TUC has no delta constraint.

17. The TUC has the delta constraint indicated by window items 18 and 19.

18. The value for an inport with the TUC may change by no less than the value specified here. If \( v_1 \) and \( v_2 \) are the values of the inport in consecutive simulation steps then \( v_2 - v_1 \geq \min \).

19. The value for an inport with the TUC may change by no more than the value specified here. If \( v_1 \) and \( v_2 \) are the values of the inport in consecutive simulation steps then \( v_2 - v_1 \leq \max \).

20. View help on the type editor dialog.
21. Update the TUC and dismiss the dialog. The revised TUC will appear in the appropriate row of the Info File Editor and [modified] will appear in the title bar. Selecting File → Save will write the .rsi file to disk and cause the [modified] tag in the title bar to disappear.

22. Discard any changes to the TUC and dismiss the dialog. Note that if the Apply button was previously clicked, any changes prior to the click will not be discarded.

23. Apply any changes made to the TUC and update the text in window item 3. Any changes prior to the click will be retained even if the dialog is dismissed with the Cancel button or clicking the x in the upper right corner of the dialog. After the dialog is closed, the revised TUC will appear in the appropriate row of the Info File Editor and [modified] will appear in the title bar. Selecting File → Save will write the .rsi file to disk and cause the [modified] tag in the title bar to disappear.

5.3.1.4 Editing Vector Types

As shown in Figure 5.5, a left panel appears in the type editor dialog when editing an inport with a vector type. The left panel includes a matrix such that each element of the matrix specifies a type for one element of the vector signal. In the case of one-dimensional vectors the matrix has a single column. To specify the type for an element of the vector, select it with the mouse and then use the controls in the right portion of the dialog to specify a type in the same manner described above for scalar types. Multiple elements may be set to the same type as follows:

1. Select one element and configure it as desired.
2. Click the Copy button.
3. Select some number of other elements by one of the following methods:
   - **Consecutive elements** Select the first element in the desired range of elements, then hold down the shift key and left-click on the last element of the desired range.
   - **Non-consecutive elements** Hold down the control key and select any number of elements by left-clicking on them.
4. Click the Paste button.

5.3.1.5 Editing Bus Types

As shown in Figure 5.6, a left panel appears in the type editor dialog when editing an inport with a bus type. The left panel includes a hierarchical display of the bus elements. To specify the type for a leaf element of the bus, select it with the mouse and then use the controls in the right portion of the dialog to specify a type in the same manner described above for scalar types.
5.3. **IMPORT TYPES PANE**

Figure 5.5: When an import has a vector type, the left portion of the type editor includes a matrix in which each element specifies the type for an element of the vector.
Figure 5.6: When an inport has a bus type, the left portion of the type editor includes a hierarchical display of the bus elements. When a leaf bus element is selected in the left panel, a type may be configured for the element using the right panel.
5.3. IMPORT TYPES PANE

5.3.2 Type Constraint Syntax

Although you typically construct type constraints with the graphical editor, a textual language formally defines the set of valid type constraints. This language is defined by the following grammar.

\[
\begin{align*}
\text{intType} & : \text{int8, int16, int32, uint8, uint16, uint32} \\
\text{floatType} & : \text{double, single} \\
\text{fixedPointType} & : \text{sfixedBitsSlopeBias | ufixedBitsSlopeBias} \\
\text{numberType} & : \text{intType | floatType | fixedPointType} \\
\text{baseType} & : \text{numberType | boolean | Enum \text{enumTypeName}} \\
\text{rsiType} & : \text{baseType | baseType \{ val_1, val_2, ..., val_n \} | numberType [ min, max ] | numberType delta [ min, max ] | numberType [ min, max ] delta [ min, max ] | floatType [ min : step : max ] | floatType [ min : step : max ] delta [ min, max ] | vectorType | busType | if timeExpr then rsiType_1 else rsiType_2} \\
\text{vectorType} & : < | rsiType_1 * rsiType_2 * ... * rsiType_n | > | < | rsiType ^ \text{int} | > \\
\text{busType} & : \{ \text{fieldName}_1 : rsiType_1, ..., fieldName_m : rsiType_m \} | \text{busName -> \{ fieldName}_1 : rsiType_1, ..., fieldName_m : rsiType_m \} \\
\text{val} & : \text{value | value : weight} \\
\text{timeExpr} & : t | \text{timeVal} | \text{timeExpr}_1 \text{relOp} \text{timeExpr}_2 \text{ | timeExpr}_1 \text{logOp} \text{timeExpr}_2 \text{ | ! timeExpr} \text{ | ( timeExpr )} \\
\end{align*}
\]

The variables/symbols in this grammar are defined as follows:

- \text{min, max, step, and value} are elements of the given base type.
• **weight** is an integer greater than zero and less than one thousand.

• t is the current simulation time.

• **timeVal** is a non-negative double value.

• **relOp** is a relational operator: `<`, `=<`, `==`, `!=`, `>`, `>=`.

• **logOp** is a logical operator: `&&`, `||`.

For a vector type `<| rsiType_1 * rsiType_2 * ... * rsiType_n |>`, the base types of all rsiType_i must be identical. For a bus type, the base types of the different field types rsiType_i may be different.

The shorthand vector type syntax `<| rsiType ~ int |>` defines a vector type with “int” repetitions of type rsiType. For example, `<| int8 ~ 4 |>` is equivalent to `<| int8 * int8 * int8 * int8 |>`.

Fixed point types are strings that specify a set of parameters for the generalized slope-bias encoding scheme for representing real numbers using integers. These strings consist of a prefix of either `sfix` (signed) or `ufix` (unsigned) followed by a suffix of the form `bitsSlopeBias`. The suffix is further divided into one required substring `bits` and two optional substrings `slope` and `bias`. These three substrings are defined as described below. Note that `fpDecimal` is a string that encodes a decimal value by replacing a negative sign “-” with “n” and the decimal point “.” with “p”. For example, “-4.9” is represented as “n4p9” and “2.3” as “2p3”. Similarly `fpInt` is a string that encodes an integer value by replacing a negative sign “-” with “n”.

**bits** is a positive integer indicating the number of bits to be used to represent values of the type.

**slope** is a decimal specified in one of two ways

- as `_SfpDecimal` such that `slope = fpDecimal`
- as `_SfpDecimal_EfpInt` such that `slope = fpDecimal × 2^fpInt`

**bias** is a decimal specified as `_BfpDecimal`.

If the `slope` substring is omitted a default value of 1 is used. If the `bias` substring is omitted a default value of 0 is used.

The following examples show some legal types and the sets of values they specify.

<table>
<thead>
<tr>
<th>RSI Type</th>
<th>Values in Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>double [0.0,4.0]</td>
<td>All double-precision floating-point numbers x, where 0.0 (\leq x \leq 4.0)</td>
</tr>
<tr>
<td>double [-10.0:0.25:10.0]</td>
<td>-10.0, -9.75, -9.5, ..., 9.5, 9.75, 10.0</td>
</tr>
<tr>
<td>int16 [-1,1]</td>
<td>-1, 0, 1</td>
</tr>
<tr>
<td>int32 {0, 4, 7}</td>
<td>0, 4, 7</td>
</tr>
<tr>
<td>uint8 {0:1, 1:3}</td>
<td>0, 1</td>
</tr>
<tr>
<td>double delta [-1.0,1.0]</td>
<td>All double-precision floating-point numbers, but input values of this type can increase or decrease by at most 1.0 during successive simulation steps.</td>
</tr>
<tr>
<td>`&lt;</td>
<td>double {1.0,2.0,3.0} * double {10.0}</td>
</tr>
<tr>
<td><code>{ x : uint8 {1, 2}, y : unit8 {3, 4} }</code></td>
<td><code>{ x = 1, y = 3 }, </code>{ x = 1, y = 4 } <code>{ x = 2, y = 3 }, </code>{ x = 2, y = 4 }`</td>
</tr>
</tbody>
</table>
In range types (i.e. those of form $t[l, u]$), the lower and upper bounds are inclusive. In subset types (i.e. those of form $t\{u_1, \ldots, u_n\}$) the probability weights in the $u_i$ enable you to influence the probabilities of the different values in the type being selected during simulation, test generation, and model validation. For example, an import having type `uint8 {0:1, 1:3}` would be assigned the value 0 in 25% of random simulation steps and the value 1 in 75% of random simulation steps, on average. In general, if import $x$ is assigned the type $t\{v_1 : w_1, v_2 : w_2, \ldots, v_n : w_n\}$, then the probability of each value in the type being assigned to $x$ is computed as follows. Let $W = w_1 + w_2 + \ldots + w_n$, where $w_i$ denotes the probability weight assigned to $v_i$ (a weight of 1 is assumed if no explicit weight is given). During random simulation or test generation, the probability of an import $x$ being assigned the value $v_i$ is $w_i / W$.

Using delta types (i.e. those of form $t\delta[l, u]$) you may constrain how the values fed into imports change from one simulation step to the next. For example, an import representing a sensor measuring temperature might have the type `double $\delta[-1.0,1.0]$` to specify that temperature can increase or decrease by at most 1 degree between any consecutive simulation steps. In general, if import $x$ has type $t\delta[l, u]$ and the current value of $x$ is $v_1$ then in the next simulation step the new value, $v_2$, of $x$ must satisfy: $v_1 + l \leq v_2 \leq v_1 + u$.

Using resolution types (i.e. those of form `double $[i : j : k]$`) where $i$ is a lower bound, $j$ is a resolution, and $k$ is an upper bound) users may specify that each value that a variable of this type may assume will differ from $i$ by some multiple of $j$. A delta constraint may also be added to a resolution type.

---

### 5.4 Configuration Variables

If you tag a workspace or data dictionary variable as a configuration variable, you indicate that Reactis may change its value in between tests, but not during a test. In Simulink / Stateflow, expressions involving such variables can be used to initialize blocks, for example Constant blocks; these expressions are evaluated before simulation begins. When these expressions are workspace or data dictionary variables, they may be declared to be configuration variables. Note, that variables appearing in larger expressions may not be declared configuration variables. For example, if `MINTEMP` is a workspace or data dictionary variable then:

- If a Constant block is initialized by the expression `MINTEMP`, then `MINTEMP` may be tagged a configuration variable.

- If a Constant block is initialized by the expression `MINTEMP - 1`, then `MINTEMP` may not be tagged a configuration variable.

A configuration variable is created by clicking on Edit → Configuration Variables... in the main Reactis window, then selecting Edit → Add... in the menu bar of the Info File Editor window that comes up. Reactis will display a list of all candidate configuration variables from which users may select. Data dictionary variables in this list are shown as [dictionary-name].[variable-name], for example `cruiseDict.InitialSpeed` specifies variable `InitialSpeed` within dictionary `cruiseDict`. The types associated with these configuration variables in `.rsi` files are the same as those that may be assigned to imports and are manipulated in the same way using the Type Editor dialog as describe in Section 5.3.1. Note, however, that configuration variables may not be given delta types, since the variables are not allowed to change during a test or simulation run.

The Type Editor may be invoked for a configuration variable in several different ways:
• From the main panel of the Reactis window, by either:
  – selecting “Configuration Variables” in the hierarchy panel to display the configuration variables panel, and then either:
    * right-clicking on a configuration variable and selecting Edit Type, or
    * double-clicking on a configuration variable

• From the Configuration Variable pane of the Reactis Info File Editor, by either:
  – double clicking on a configuration variable row, or
  – selecting a configuration variable row with a single click, and then right-clicking in the body of the Info File Editor and selecting Edit..., or selecting menu item Edit → Edit...

Note that if you intend for the changes of a configuration variable to propagate into an S-Function via a parameter, then you must ensure that your S-Function defines the mdlProcessParameters function as described in Section 14.2.1.

The specification of configuration variables of type array or structure may be done using the text version of the type editor. To edit the type for such a variable, right-click on the row and select “Edit Type Text”. A text entry dialog will come up in which the array or structure type may be specified. The syntax is as follows:

• One-Dimensional Arrays
  \[ \text{basetype array } [\text{numElts}] \{ \text{instance}_1, \ldots, \text{instance}_n \} \] where \(\text{numElts}\) is a positive integer and each \(\text{instance}_i\) has the form \([\text{elt}_1, \ldots, \text{elt}_{\text{numElts}}]\).

• Two-Dimensional Arrays
  \[ \text{basetype array } [\text{numRows},\text{numCols}] \{ \text{instance}_1, \ldots, \text{instance}_n \} \] where \(\text{numRows}\) and \(\text{numCols}\) are positive integers and each \(\text{instance}_i\) has the form \([\text{row}_1; \ldots; \text{row}_{\text{numRows}}]\) and each \(\text{row}_i\) has the form \(\text{elt}_1; \ldots; \text{elt}_{\text{numCols}}\).

• Structures
  \[ \{ \text{fieldName}_1 : \text{fieldType}_1, \ldots, \text{fieldName}_n : \text{fieldType}_n \} \{ \text{instance}_1, \ldots, \text{instance}_m \} \] where each \(\text{instance}_i\) has the form \(\{\text{fieldName}_1 = \text{value}_1, \ldots, \text{fieldName}_n = \text{value}_n\}\).

The following examples demonstrate the notation.

• Three instances of a four-element array of doubles:

\[
\text{double array [4]}
\quad \{ \\
\quad \quad [1,2,3,4], \\
\quad \quad [11,12,13,14], \\
\quad \quad [21,22,23,24] \\
\quad \}
\]

• Two instances of a structure for initializing a 2-D Lookup Table:
Configuration variable sets let you specify multiple constraints for each configuration variable in your model. This feature offers a convenient way to capture dependencies among configuration variables. For example, in a configurable model of an automotive transmission the maximum expected RPMs might depend on the number of gears. In such a case, different configuration variable sets can be created to capture each allowed combination.

Figure 5.7: The Configuration Variables pane of the Reactis Info File Editor.
Figure 5.7 shows the Info File Editor with the Configuration Variables pane selected. Within the Configuration Variables pane, the Configuration variable set pull-down (item 1 in Figure 5.7) is used to manipulate configuration variable sets. The menu contains the following items:

\[ \text{ConfigVarSet}_1 \ldots \text{ConfigVarSet}_n \]. Selecting \( \text{ConfigVarSet}_i \) switches to the configuration variable set named \( \text{ConfigVarSet}_i \). In Figure 5.7 the configuration variable sets are named \text{Slow Initial Speed}, \text{Fast Initial Speed}, and \text{All Initial Speeds}. The constraints for \( \text{ConfigVarSet}_i \) will appear in the main configuration variable pane and can be modified.

Create a copy of current set... Creates a copy of the current configuration variable set. You will be queried for the name of the new set and the constraints for each configuration variable will be copied to the new set.

Create new set... Create a new configuration variable set. You will be queried for the name of the new set and the constraint for each configuration variable will simply be the type of the underlying workspace or data dictionary variable.

Delete current set... Delete the current configuration variable set.

Rename current set... Change the name of the current configuration variable set.

Note that the name of the current configuration variable set is always displayed as the value of the Configuration variable set pull-down. This lets you know which set you are currently editing.

To the right of the Configuration variable set pull-down is the Enabled check box (item 2 of Figure 5.7). When Enabled is checked, the current configuration variable set is eligible for use by Reactis Tester.

Note that all configuration variable sets contain the same member variables. Hence, adding or removing a variable will change the membership of all sets, not just the current set. What does differ between sets is the type constraint of each configuration variable. For example, in Figure 5.7, each constraint variable set contains a single variable \( \text{InitialSpeed} \). The value of \( \text{InitialSpeed} \) is constrained to a different range in each set:

\[
\begin{align*}
0 & \leq \text{InitialSpeed} \leq 10 & \text{in } \text{Slow Initial Speed}, \\
40 & \leq \text{InitialSpeed} \leq 50 & \text{in } \text{Fast Initial Speed}, \text{ and} \\
0 & \leq \text{InitialSpeed} \leq 50 & \text{in } \text{Any Initial Speed}.
\end{align*}
\]

When generating tests at the start of each new test, Reactis Tester will randomly select one of the enabled configuration variable sets and then select a value for each configuration variable that is allowed by the constraints in the selected set. If no set is enabled, then Tester works as if no configuration variables are defined (the values for workspace variable will come from the workspace).

5.5 Test Points

This pane displays a list of test points currently associated with the model. Test points are internal data items (Simulink signals or Stateflow variables) that Reactis subjects to user-specified checks during simulation and test generation. For example, Reactis can be configured to record values for test points in test suites and have Reactis Simulator flag any differences between the values computed by a model for a test point and those stored in a test suite.
Additionally, a type constraint can be assigned to a test point and Reactis can be configured to monitor if a test point is ever assigned a value outside the set of expected values specified by the constraint.

5.5. TEST POINTS

5.5.1 Adding and Removing Test Points

To add or remove a test point, Simulator must be turned off.

A test point can be added by right-clicking on a data item in the Reactis main panel and selecting **Add To Test Points**. After a test point is added, a visual indicator appears in the Reactis main panel as shown in Figure 5.8. If the test point is in Simulink, the indicator is a pair of eyeglasses appearing above a signal. If the test point is in Stateflow, the indicator is a box at the bottom of the Stateflow diagram containing the eyeglasses symbol and the test point name.

A test point can be removed by right clicking on the test point in the main panel (when Simulator is off) and selecting **Remove Test Point**. Alternatively, a test point can be removed from the **Test Points** pane of the Info File Editor by:

1. selecting a row in the pane, and

2. selecting **Edit → Remove** from the Info File Editor menu or clicking the **Remove ( )** button in the Info File Editor toolbar.

5.5.2 Importing Test Points from a Simulink model

Reactis allows you to import test points defined in the Simulink model by selecting **Tools → Import from Simulink** from within the **Test Points** pane on the Info File Editor. Selecting that item brings up the dialog in Figure 5.9. The labeled items of Figure 5.9 configure the import parameters as follows:

1. When checked, import test points defined in Simulink by right-clicking on a signal line, selecting “Signal Properties” and checking the “Test Point” box.
2. When checked, import test points defined in Stateflow by checking the “Test Point” box in the variable’s properties dialog in Stateflow.

3. When checked, import a Simulink signal as a Reactis test point if “Signal name must resolve to Simulink signal object” is checked in the “Signal Properties” dialog.

4. When checked, import a Stateflow variable as a Reactis test point if “Data name must resolve to Simulink signal object” is checked in the variable’s properties dialog in Stateflow.

5. If not checked, Reactis uses the name of the block from which the signal emerges as the test point name. If checked, Reactis uses the name of the Simulink.signal object as the test point name.

6. If either of the “must resolve to Simulink.signal object” options is checked, checking this box will assign the minimum and maximum ranges from the Simulink.signal object to the imported test point.

7. If the “Import Min/Max range...” option is checked, checking this box will set up the imported test point to track boundary coverage for the imported minimum and maximum.

8. If the “Import Min/Max range...” option is checked, checking this box will set up the imported test point to create an assertion that will be violated if the value of the test point exceeds be imported maximum or minimum.

9. If the “Import Min/Max range...” option is checked, checking this box will set up the imported test point to produce either a warning or error message if its value exceeds the imported minimum or maximum.

10. If the “Check test point values against range” option is checked, selecting this will produce a warning message and continue test case generation or simulation if the test point value exceeds the imported minimum or maximum.

11. If the “Check test point values against range” option is checked, selecting this will produce an error message and stop test case generation or simulation if the test point value exceeds the imported minimum or maximum.

**5.5.3 The Test Points Pane**

The Test Points pane of the Info File Editor lists each test point created for a model as shown in Figure 5.10. Test points may be removed using this pane as described above. Double-clicking on the name of a test point in this pane causes the test point to be displayed and highlighted (with yellow flashing) in the main Reactis panel. The Tolerance and Type Constraint columns list aspects of how each test point is configured as described in the next section. Double-clicking in the Tolerance or Type Constraint column for a test point opens the Test Point Properties Editor for the test point.
5.5.4 Configuring a Test Point with the Test Point Properties Editor

The Test Point Properties Editor (Figure 5.11) lets you configure the checks that Reactis will perform on a test point during test generation and simulation. It can be opened from both the Reactis main panel and the Reactis Info File editor.

- From the Reactis main panel:
– Double-click on a test point symbol, or
– Right-click on a test point symbol and then select Edit Test Point Properties...

• From the Test Points pane of the Reactis Info File editor:
  – Double-click on the Tolerance or Type Constraint column for a test point, or
  – Select a test point row, then click the Edit ( ) button in the toolbar, or
  – Select a test point row, then selecting menu item Edit → Edit....

Figure 5.11: The Test Point Properties Editor for configuring test point behavior.

The labeled items of Figure 5.11 configure a test point as follows:

1. The name of the test point. This need not match the underlying signal or variable name.

2. Clicking the Highlight button will cause the test point to be displayed in the main Reactis panel and be highlighted by flashing yellow.
3. The location of the test point within the model hierarchy.

4. When checked, Reactis will record a value for the test point whenever it constructs a test (in Tester, Simulator, Validator) in the same manner as is done for computed top-level output values.

5. When checked, before storing a test point value in a test suite, Reactis will check whether the computed value is within the set of acceptable values specified by the type constraint for the test point. If not it will convert the computed value to the nearest value within the set specified by the type constraint and then store the converted value in the test suite.

6. If checked, whenever a test is executed in Reactis Simulator, at each simulation step the value computed by the model for a test point will be compared against the value stored in the test step for the test point. If the difference exceeds the tolerance specified by item 7, then the difference will be flagged.

7. If item 6 is checked, this section specifies the tolerance to be used when comparing a computed test point value against the value in a test suite. The pull-down lets you select one of five methods for computing a tolerance. How the methods work is described in Section 5.6.1.

8. When checked, Reactis will create coverage targets for the boundary values of the test point’s type constraint. For example, targets will be created for the minimum and maximum values if a range constraint is given for a test point. For a description of the boundary value targets inferred from a type constraint in general see Section 6.3.2.1.

9. When checked, Reactis will create and track an assertion that will be violated if the test point is assigned a value outside the set of values specified by its type constraint. For example, if a range is specified, the assertion will be violated if the test point ever assumes a value outside the given range.

10. When checked, Reactis will generate either a warning or error if a test point is assigned a value outside the set specified by its type constraint.

11. Clicking this button opens a dialog for specifying the type constraint for the test point. See Section 5.3.1 for instructions on operating the type editor dialog.

12. Displays the type constraint for the test point. Note, that specifying a constraint is only necessary if one of the operations using type constraints is enabled. When no type constraint is specified, this box will display <not set>.

13. Display help for the Test Point Properties Editor.

14. Save any changes and close the editor.

15. Discard any changes and close the editor.

When exporting a test suite to CSV format, users have the option to export test point data in addition to import and output values.

Simulink Test Points in a model can be imported into Reactis by selecting Info File Editor menu item Tools → Import from Simulink.
5.6 Outport Tolerances

When executing a test suite in Reactis Simulator, at each simulation step, for each outport, Reactis compares the value computed by the model for the outport against the value stored in the test suite for the outport and step. The Outport Tolerances pane, shown in Figure 5.12, lets you specify a tolerance for each outport to be used in these comparisons. The Outport Tolerances pane displays five properties for each outport, labeled one through five in Figure 5.12:

1. The port number.
2. The port name.
3. The tolerance method (see Section 5.6.1).
4. The relative tolerance value. If the method does not use relative tolerance, this will be blank.
5. The absolute tolerance value. If the method does not use absolute tolerance, this will be blank.

The tolerance for a port is changed by either (1) double-clicking on the row for the port, or (2) left-clicking on the row for the port and then right-clicking and selecting Edit.... Either of these actions will open the tolerance editor dialog for the selected port. Note that changing the port-specific tolerance will override the global tolerance for the model set within the Error Checking pane of the Info File Editor. The tolerance editor dialog is explained in Section 5.7.
5.6.1 Tolerance Methods

There are five methods for computing the tolerance: Absolute, Relative, Max, and Min, and Inherit.

The **absolute tolerance** method flags a difference between a computed value and a test value if the absolute value of the difference exceeds a given amount. More precisely, given two values $x$ and $y$ and an absolute tolerance $\epsilon$, the difference between $x$ and $y$ is flagged if and only if $|x - y| > \epsilon$.

The **relative tolerance** method flags a difference between a computed value and a test value if it exceeds an amount determined by multiplying the tolerance by either the computed or test value. More precisely, given two values $x$ and $y$ and a relative tolerance $\epsilon$, the difference between $x$ and $y$ is flagged if and only if any of the following conditions are true:

- $|x - y| > \epsilon \times |x|$ when $x \neq 0$
- $|x - y| > \epsilon \times |y|$ when $y \neq 0$
- $|x - y| > \epsilon^2$ when $x = 0$ or $y = 0$

A relative or absolute tolerance of zero will require that the model and test values always be the same, a relative tolerance of 0.1 will allow a difference of 10%, and a relative tolerance of 1 will allow the model value to be at most twice the test value.

The **max tolerance** method uses both a relative and absolute tolerance. A difference is flagged only if it is flagged by both the absolute and relative tolerance methods.

The **min tolerance** method uses both a relative and absolute tolerance. A difference is flagged if it is flagged by either the absolute or relative tolerance method.

If the tolerance method is set to Inherit, then the tolerance method is inherited from the model’s global tolerance setting (set in the Error Checking pane of the Info File Editor).

In addition to their use in flagging differences when running a test suite in Simulator, the tolerances specified here are also used in the following places:

- When highlighting tolerances and differences within a Difference Scope (see Section 7.5.5)
- When executing a test suite generated by Reactis within Simulink using the rsRunTests utility (see Section 12.2.1)

5.7 The Tolerance Editor Dialog

Tolerances are specified via the tolerance editor dialog, which is shown in Figure 5.13. The labeled items in Figure 5.13 are explained as follows:

1. This pulldown menu lets you choose the tolerance method. The possible choices are Inherit, Relative, Absolute, Min, or Max. These are explained in Section 5.6.1.

2. This data entry box contains the relative tolerance value used by the selected method, which must be non-negative. If the method does not use relative tolerance, it will be disabled.

3. This data entry box contains the absolute tolerance value used by the selected method, which must be non-negative. If the method does not use absolute tolerance, it will be disabled.
CHAPTER 5. THE REACTIS INFO FILE EDITOR

Figure 5.13: The dialog for specifying the tolerance to use when comparing computed and test suite values. As shown, an absolute tolerance of 0.1 is specified for port speed, which means that differences which are less than or equal to 0.1 will be ignored.

4. This button opens the help dialog for the tolerance editor dialog.

5. This button accepts the displayed tolerance value and closes the tolerance editor dialog.

6. This button ignores any changes made to the tolerance and closes the tolerance editor dialog.

5.8 General

The General pane lets you configure the following model-specific settings. When a new .rsi file is created, the values for these settings are taken from the Default Model-Specific Settings dialog (invoked by File → Default Model-Specific Settings...).

Model file encoding. Character encoding used by the model.

Conditional input branch execution. Conditional input branch execution is an optimization that causes some blocks in a model not to execute when they are not necessary to compute the outputs of the model. Please see Section 15.1 for a description of this optimization. The options for this menu are:

- On Use conditional input branch execution.
- Off Do not use conditional input branch execution.
- Inherit from model settings Enable or disable conditional input branch execution according to the settings in the .slx file.

Bus element name visibility. Reactis V2012 introduced substantial enhancements for models that use bus types. One of the central improvements includes the visibility of bus element names in many new places (e.g. when editing type constraints for top-level bus inports using a new user-friendly graphical editor, when adding a watched variable, or when opening a scope).

This setting lets you disable the new bus-related feature and configure Reactis to internally treat buses as it did prior to V2012. Due to the substantial loss of new capabilities
that occurs, we suggest you do not disable the feature unless the Reactive Systems support team suggests you do so to work around an issue that arises when your model uses the new functionality. The settings are as follows:

**Show element names for all buses** the recommended setting that enables the new set of bus-related features.

**Do not show element names (legacy)** disable the new bus-related features.

*Create and use cache files (with extension '.mwi').* Instructs Reactis to create a file containing intermediate information it has computed about a model. These files can significantly improve subsequent load times. If you modify your model, Reactis will detect the changes and re-generate the .mwi file. Clicking the **Flush Cache** button causes the cache to be emptied. Subsequently intermediate information will be recomputed as needed. Note, however, that .mwi files are not deleted on disk.

*When creating a test suite, also create a parameters file (extension .rtp).* Instructs Reactis Tester to store launch parameters and the .rsi file used for a given run in a Reactis Tester Parameter file (.rtp file). Subsequently:

- the .rsi file may be extracted from the .rtp file by selecting File → Extract Info File...;
- the Tester parameters may be loaded into the Tester launch dialog by clicking the **Load** button.

Re-running Tester with the same .rsi file and same parameters on the same model will regenerate the test suite produced by the original Tester run.

*Propagate set_param changes by saving the model to a temporary file.* Models using self-modification commands (e.g. **set_param**, **add_block**, etc.) could previously cause problems because Reactis would not see the changes made by them. This setting has been added to deal with this scenario. If this setting is enabled, Reactis will cause Simulink to apply all the changes to the model, then automatically save them to a temporary file. Reactis then imports the temporary model file, allowing Reactis to see the applied changes.

*Use new fixed-step sample time computation.* Enabling this option will slightly change the way Reactis computes time values for models configured to use the fixed-step solver, especially those with sample times that have poor floating point representations such as 0.1, 0.01 or similar. The new behavior makes the rounding errors in the Reactis sample time computations line up better with Simulink’s rounding errors for such models. Note that after enabling this option, test suites created with the legacy approach employed prior to Reactis V2010 can show different behavior in cases where a small change in sample time can cause the model to significantly change its execution.

*Ignore ToFile and ToWorkspace blocks.* Checking this box enables Reactis to process models that include these blocks by ignoring them. If not checked, Reactis reports an error whenever it simulates a model containing such blocks.

*Short-circuit Simulink boolean operators.* Checking this box indicates that evaluation of logic blocks should be short-circuited, i.e. should be halted as soon as its output value has been determined. For example, once one input to an AND block has been found to be false, it
is known that the output of the AND block should be false; if this box is checked, then the remaining inputs are not examined. Short-circuiting affects coverage analysis and reporting for Condition, Decision and MC/DC coverage metrics, as unexamined inputs are treated as “don’t care” points. More information about coverage may be found in Chapter 6.

**Short-circuit Stateflow boolean operators.** Checking this box indicates that boolean operators in Stateflow conditions should be short-circuited. See the previous item for a discussion of short-circuiting.

**Prepend Reactis search path to MATLAB path.** When this box is checked, every time Reactis launches MATLAB it will prepend the Reactis search path to the MATLAB path.

**Execute startup.m and pathdef.m scripts in model folder.** When this box is checked, every time Reactis launches MATLAB it will look for pathdef.m and startup.m files residing in the same folder as the model. If a pathdef.m file is found, Reactis will set the MATLAB path accordingly to its contents. If a startup.m file is found, Reactis will direct MATLAB to execute it prior to loading the model.

---

### 5.9 Error Checking

The Error Checking pane lets you configure the following model-specific settings related to which types of errors Reactis will flag. When a new .rsi file is created, the values for these settings are taken from the Default Model-Specific Settings dialog (invoked by File → Default Model-Specific Settings...).

**General:**

**If top-level input value does not match constraint** Reactis lets you associate a constraint with each top-level inport of a model from the Edit → Inport Types pane of the Info File Editor. For example, you can specify a range for a given input, and then, when Reactis Tester generates tests it will only select values within the range for the tests it constructs. However, if external test data is brought into Reactis (e.g. via importing a .csv file), it is possible for the data to contain input values that violate the constraints set for the corresponding input in Reactis. The If top-level input value does not match constraint setting allows you to set the desired behavior for the case that an input value is encountered during simulation that does not match the input’s constraint. The possible settings are:

- **Use value.** Proceed and produce no error or warning.
- **Use value and produce warning.** Proceed but produce a warning.
- **Produce error.** Produce an error and stop test generation or simulation.

Note that ‘Produce error’ is the recommended setting because it allows Reactis to use top-level input constraints to improve both static reachability analysis and constraint solving during test case generation.

**If configuration variable value does not match constraint** Each configuration variable can be given a constraint from the Edit → Configuration Variables pane of the Info File Editor. For
example, a range constraint can be specified and then Reactis Tester will only select values within that range. However, if external test data is brought into Reactis (e.g. via importing a .csv file), it is possible for the data to contain configuration variable values that violate the constraints set for the configuration variable. The *If configuration variable value does not match constraint* setting allows you to set the desired behavior for the case that an configuration variable value is encountered during simulation that does not match the variable’s constraint. The possible settings are:

**Use value.** Proceed and produce no error or warning.

**Use value and produce warning.** Proceed but produce a warning.

**Produce error.** Produce an error and stop test generation or simulation.

Note that ‘Produce error’ is the recommended setting because it allows Reactis to use top-level input constraints to improve both static reachability analysis and constraint solving during test case generation.

**If delta constraint exceeds type range.** A delta constraint for a top-level import limits how much values on the import can change from one simulation step to the next. A delta constraint that sets the maximum allowed change to be bigger than the entire range of the type most likely indicates an erroneous delta constraint. This setting lets you configure whether and how Reactis should flag such constraints with a size error.

To define how the setting works assume a delta constraint:

\[
\text{double } [\text{min}, \text{max} ] \text{ delta } [\text{dmin}, \text{dmax}] 
\]

and let \( \text{range} = \text{max} - \text{min} \). Then a delta constraint size error occurs if \( \text{dmin} < -\text{range} \) or \( \text{dmax} > \text{range} \). When such an error occurs, Reactis clips the \( \text{dmin} \) and/or \( \text{dmax} \) values so no change is bigger than \( \text{range} \). If you start Simulator or Tester using an .rsi file containing a delta constraint size error, then Reactis does the following according the the setting in this dialog:

**Clip delta.** Proceed and produce no error or warning.

**Clip delta and produce warning.** Proceed but produce a warning.

**Produce error.** Produce an error and stop test generation or simulation.

**Tolerance for output and test point differences** When running a test suite, Reactis Simulator compares the top-level outputs and values for test points produced by the model to those stored in the test suite and produces a warning if a difference is detected. Since minor differences for floating point values due to small rounding errors are common, a tolerance can be specified here that will be used when comparing floating point values. To require an exact match, the tolerance can be set to absolute or relative zero. Note that the global tolerance specified here can be overridden for a particular outport or test point as described in Sections 5.6 and 5.5.

**In Simulink and Stateflow:**
On integer overflow. Integer overflow can occur during the evaluation of some blocks. These blocks typically have a SaturateOnIntegerOverflow parameter. When set to ‘on’, this parameter specifies that the block should evaluate to the maximum value for the type on overflow and the minimum value on underflow. When SaturateOnIntegerOverflow is set to ‘off’ Simulink uses the model parameter IntegerOverflowMsg to determine how overflows are handled. There are three possible settings:

- **None.** Return the wrapped around value.
- **Warning.** Return the wrapped around value but also issue a warning.
- **Error.** Raise an error and halt simulation.

Reactis gives users a way to override the IntegerOverflowMsg parameter with the ‘On Integer Overflow’ setting which may have one of three possible values:

- **Wrap over.** Return the wrapped around value.
- **Wrap over and produce warning.** Return the wrapped around value but also issue a warning.
- **Produce error.** Raise an error and halt simulation.

Reactis gives the user a way to override the Simulink Inf/NaN behavior. The Reactis **When detecting Inf or NaN values** setting may have one of three possible values:

- **No action.** Continue simulation or test-generation.

---

1To set the IntegerOverflowMsg parameter, from the Simulink main menu select Simulation → Configuration Parameters. In the resulting dialog, select the ‘Data Validity’ item from group ‘Diagnostics’ and then set via ‘detect overflow’.

2Wrapping around works in the usual fashion. Incrementing the maximum value in a type by one yields the minimum value in the type. Decrementing the minimum value in a type yields the maximum value in the type.

3To set the Inf/Nan behavior in Simulink, from the Simulink main window select Simulation → Configuration Parameters. In the resulting dialog, select the ‘Data Validity’ item from group ‘Diagnostics’ and then set behavior via ‘Inf or NaN block output’.
5.9. ERROR CHECKING

**Produce warning.** Continue simulation or test-generation, but also issue a warning.

**Produce error.** Raise an error and halt simulation or test-generation. In the case of interrupted test-generation, Reactis Tester returns a test suite that includes a test leading to the Inf/NaN.

**Inherit from model settings.** Use the Simulink setting from the model; however, Reactis does not currently generate warnings. When the Simulink setting is ‘warning’, Reactis behaves as if it is set to ‘none’.

*When detecting conversion of Inf or NaN values to integer.* Configure what happens if a model attempts to convert an Inf or NaN value to an integer as follows:

**No action.** Emulate the Simulink/Stateflow behavior: for fixed-point values convert to the value that has a stored integer of zero, otherwise convert to zero.

**Produce warning.** Emulate the Simulink/Stateflow behavior, but also produce a warning.

**Produce error.** Raise an error if such a conversion is attempted.

**Inherit from “When detecting Inf or NaN values”**. Inherit from the previous setting.

*When shifting by more than word size.* Simulink includes a Shift block and Stateflow includes a shift function. If either operation results in a shift larger than the word size you have specified for your model, most likely this indicates a modeling error. This setting lets you direct Reactis to flag such illegal shift operations as follows:

**No action.** Do not flag shifts larger than the word size.

**Produce warning.** Generate a warning if a shift larger than the word size is detected.

**Produce error.** Raise an error if a shift larger than the word size is detected.

**Inherit from model (error otherwise).** For Simulink Shift blocks respond to a shift larger than the word size as configured in the block. For Stateflow raise an error.

*When comparing enumerated values of different types.* In Stateflow, comparing two expressions of different enumerated types is allowed. However, allowing such comparisons can result in unexpected behavior. Stateflow compares the underlying integer value of the enumerated types in that case, which may be different from the literal enumeration names. In many cases such comparisons are the result of typos or copy/paste errors and are not intentional. Reactis by default flags such comparisons as errors but in cases where they are intentional this setting allows you to specify the desired behavior:

**No action.** Do not flag comparisons of different enumerated types as errors.

**Produce warning.** Generate a warning if a comparison of different enumerated types is detected.

**Produce error.** Raise an error if a comparison of different enumerated types is detected.

*When Detecting Stateflow outport merge.* This setting configures Reactis to issue a warning or error when a model contains a particular configuration of Stateflow charts and merge blocks that can yield unexpected behavior. Note that even for models with this unexpected behavior, Reactis implements the same semantics as the MathWorks Simulink / Stateflow engine.
We now describe how the unexpected behavior can occur. If an outport of a Stateflow chart is directly connected to a merge block and the outport is read within the chart, then an unexpected back-propagation of values from outside the chart into the chart via the outport can occur. This setting enables you to configure Reactis to produce a warning or error when such a construct is detected. The example in Figure 5.14 shows a model in which this back-propagation occurs.

**When detecting undirected local event broadcast in Stateflow.** Undirected local event broadcasts can cause infinite recursion in a Stateflow chart. This setting configures Reactis to produce a warning or error when an undirected local event broadcast occurs.

**When detecting multiple trigger event outputs.** If an edge-triggered output event is triggered multiple times in a single step then the events are queued and output during subsequent simulation steps. This can cause the chart’s output to change in a step where no action occurs within the chart. The *When detecting multiple trigger event outputs* setting con-
figures Reactis to produce a warning or error when an edge-triggered output event is triggered multiple times during a single step.

*Stateflow infinite loop timeout (seconds).* Maximum time for Reactis to compute a simulation step. If the computation of a simulation step takes longer, Reactis assumes there is an infinite loop; the simulation is then terminated and an error message printed.

**In C Code:**
The settings in this section are enabled when using the Reactis for C Plugin (which is enabled from the Reactis for C pane of the Global Settings dialog). For a description of these settings, see Section 16.2.

## 5.10 Coverage Metrics

The different coverage metrics supported by Reactis are defined in Chapter 6. The model-specific settings in the coverage pane let you enable or disable each metric. When a new `.rsi` file is created, the values for these settings are taken from the Default Model-Specific Settings dialog (invoked by File → Default Model-Specific Settings...). If a metric is disabled:

- the metric will not be targeted by Tester when generating tests, and
- Simulator will not include the targets from the metric in the Coverage Summary dialog, the Coverage Report Browser, and the highlighting in the main panel.

The following dependencies exist between the coverage metrics:

- Enabling Condition coverage requires that Decision coverage be enabled
- Enabling MC/DC or MCC require that Condition coverage be enabled

Additionally, the meaning of several of the metrics (CSEPT, Decision, and Boundary Value) can be adjusted in this dialog as follows.

**Child State Exit via Parent Transition (CSEPT)** See Section 6.2.4 for a description of how these settings configure what constitutes CSEPT targets.

**C-specific Decision Metric** See Section 16.5 for an explanation.

**Decisions** *Multi-block Decision, Condition, MC/DC and MCC.* Switches between single-block and multi-block coverage tracking for Simulink logical operator blocks. For more details about this feature see Section 6.3.1.

**Multiple Condition Coverage (MCC)** *Maximum number of conditions per decision.* If the actual number of conditions for a decision exceeds the number given here then MCC coverage will not be tracked for that decision. If this happens a warning will be shown when enabling Simulator or generating test cases. A warning will also appear in coverage reports.

**Boundaries** *Boundary coverage for relational operators.* As described in Section 6.3.2.2, Reactis tracks boundary value coverage for Relational Operator blocks. Since this coverage metric has the potential to introduce a large number of very hard to cover targets, some users might prefer to not track these targets. This check box enables and disables boundary value coverage tracking for relational operators.
Include relational operators comparing floating-point values. Toggles whether relational operators with floating point inputs are included in the set of relational operator boundary value targets.

Relative tolerance. When relational operators comparing floating-point values are included, this entry box enables you to specify a relative tolerance as described in Section 6.3.2.2. To generate the smallest increment representable in floating-point, type the term “next”.

Track coverage for ‘equals’ case. When relational operators comparing floating-point values are included, this box toggles whether a coverage target should be created for the ‘equals’ case, i.e. a=b as described in Section 6.3.2.2.

Boundary coverage for integer saturation. When enabled, targets are created for blocks which are set to saturate on integer overflow. See Section 6.3.2.3 for details.

Include targets for no saturation. Toggles whether or not a target is created which is covered when integer saturation does not occur for blocks which are set to saturate on integer overflow. See Section 6.3.2.3 for details.

Validator Objectives Track coverage for contents of Validator Objectives. Toggles whether or not the targets within diagram-based Validator Objectives are tracked. The objective itself (assertion or UDT) is tracked even when the targets within the objective are not.

5.11 Excluded Coverage Targets

The Excluded Coverage Targets pane lists all coverage targets which have been excluded from coverage. Every excluded target occupies one row within the pane, which contains 5 columns, labeled 1 through 5 in Figure 5.15.
5.12. VALIDATOR OBJECTIVES

1. **Metric.** The coverage metric which the excluded target measures.

2. **Item.** The name of the immediate item (block, state, transition, etc.) where the target is located.

3. **Target.** A description of the target (e.g., Condition 1 true).

4. **Status.** The status of the target, which has three possible values:

   - **excluded** The target is excluded without monitoring.
   - **[excluded]** The target is excluded and monitored by assertion, and the target has not been covered.
   - **[covered]** The target is excluded and monitored by assertion, and the target is covered.

5. **System.** The path to the subsystem where the target is located.

Most targets are added to the **Excluded Coverage Targets** pane by right-clicking on them in the main panel and selecting **Track Coverage**. Boundary, condition, decision, lookup table element, MC/DC, and MCC targets are added to the pane by first right-clicking and selecting **View coverage details**, and then right-clicking on the target within the **Coverage Details** dialog and selecting **Track Coverage**.

The status of a target listed in the **Excluded Coverage Targets** pane can be edited when Simulator is enabled by left-clicking on the row containing the target, then right-clicking in the **Status** column and selecting **Edit...** Note that enabling coverage tracking for the target will cause it to disappear from the **Excluded Coverage Targets** pane, and currently the only way to undo such a change is to find the target in the main panel and exclude it from coverage a second time. See Section 6.5 for details.

Double left-clicking on any row of the **Excluded Coverage Targets** pane will cause the corresponding target to be highlighted in the main panel with a yellow flashing background for a few seconds.

Left-clicking on the header for any column will sort the targets by the the entries in that column. Left-clicking a second time on the same column header will reverse the order.

---

5.12 Validator Objectives

Chapter 9 describes the Validator component of Reactis. Validator gives the user a set of tools to check whether a model captures the intended behavior. **User-defined targets** and **virtual sources** are used to specify scenarios that should be tested. **Assertions** can be used to check that a model responds to tests as expected. Collectively, these three types of virtual instrumentation are called **Validator objectives**.

The **Validator Objectives** pane of the Info File Editor enables you to view and manipulate the Validator objectives that have been inserted into a model. Each objective has a row in the pane with entries for the following columns:

**Type**  Assertion, user-defined target, or virtual source.

**Status**  This field reports different information depending on the type of the objective and whether Simulator is currently on or off. When Simulator is off, this column simply reports whether the objective is enabled, disabled, or orphaned. An objective is orphaned
when the subsystem in which it is located is removed from the model or the path to
the subsystem changes (e.g. A/B/C changes to A/D/C). Orphaned objectives remain
visible in the Validator Objectives pane to facilitate their reinsertion into a model. To
reinsert an objective simply select the row of the objective, then select Edit → Copy, and
then in the main panel of Reactis right-click at the appropriate location and select paste.
When Simulator is on, for non-orphaned objectives, the column reports whether asser-
tions have been violated and user-defined targets have been covered.

Table 5.1: Possible values of the Status column.

<table>
<thead>
<tr>
<th>Objective Type</th>
<th>Simulator off</th>
<th>Simulator on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion</td>
<td>enabled/disabled/orphaned</td>
<td>okay/violated/orphaned</td>
</tr>
<tr>
<td>User-Defined Target</td>
<td>enabled/disabled/orphaned</td>
<td>covered/not covered/orphaned</td>
</tr>
<tr>
<td>Virtual Source</td>
<td>enabled/disabled/orphaned</td>
<td>enabled/disabled/orphaned</td>
</tr>
</tbody>
</table>

Name Name of the objective.

System Location of the objective in the model.

Double-clicking on a row highlights the corresponding objective in the main panel. Obje-
tives may be edited or removed by selecting the row and using the Edit menu or right-clicking
on the row.

You can specify whether the coverage targets within diagram-based Validator Objectives
should be tracked using the Track coverage for contents of Validator Objectives setting in the Cov-
erage Metrics pane. The objective itself (assertion or UDT) is tracked even when the targets
within the objective are not.

5.13 C Code

When using the Reactis for C Plugin, this pane displays a list of places in a model that reference
C code (S-Functions and Stateflow custom code) and gives the user the ability to adjust the
settings for white-box analysis of each S-Function. Please see Chapter 16 for details.

5.14 External EML Functions

When using the Reactis for EML Plugin, this pane lets you specify a list of external EML
functions (stored outside the model in .m files) that are used by the model. Coverage targets
within the external functions will be tracked and targeted by Reactis Tester. You can also step
into the external functions when running the model in Reactis Simulator.

5.15 Callbacks

The Callbacks pane gives users a way to specify fragments of MATLAB code that Reactis will
run before or after loading a model in Simulink. Note that these callbacks are distinct from
those maintained by Simulink.
5.16 Search Path

This pane gives users a way to specify a model-specific search path. This path is prepended to the global search path to specify the list of folders (for the current model) in which Reactis will search for files such as Simulink model libraries (.mdl), MATLAB scripts (.m), and S-Functions (.dll, .mexw32, .m).

The model-specific path is manipulated in exactly the same manner as the global path is manipulated from the Path pane of the Reactis Global Settings dialog. For a description of how the operations work, please see Section 4.8.5. Note that the one difference between the two path panes is that the Import button of the model-specific pane first loads the model in MATLAB before importing the MATLAB path. This causes any pathdef.m or setup.m files in the model folder to execute. Any model callbacks (pre-load, post-load functions) will also be executed, and perhaps modify the MATLAB path, prior to importing it into Reactis. The net result is that the imported MATLAB path is the one that results after the model is loaded in MATLAB.

5.17 File Dependencies

In general, models may contain references to items defined in auxiliary .m files To enhance performance, Reactis computes and stores information about models when they are loaded in order to speed up subsequent loads. If a .m file that a model depends on changes, the stored information needs to be re-computed. The dependency information given in this pane tells Reactis which .m files, when changed, should trigger this re-computation. Any changes to these files will cause any preprocessing information that Reactis has cached about the model to be discarded and recomputed. Note that .slx files need not be listed here; the .slx files on which a model depends are inferred automatically.

Note that users should include all .m files that their model depends on, including those referenced from:

- Any Simulink model callback or block callback that is executed when loading or running the model (PreLoadFcn, PostLoadFcn, InitFcn, StartFcn).
- A “startup.m” file located in the folder where the model file is located.
- The “Callbacks” tab of the Reactis Info File Editor.

---

4This caching feature may be disabled from the General pane of the Info File Editor by unchecking the check box Create and use cache files.
Chapter 6

Reactis Coverage Metrics

Reactis uses a number of different coverage metrics on Simulink / Stateflow models to measure how thoroughly a test or set of tests exercises a model. In general, coverage metrics record how many of a given class of syntactic constructs, or coverage targets, that appear in a model have been executed at least once. Some of the metrics supported by Reactis involve only Simulink, some are specific to Stateflow, and the remaining are generic in the sense that they include targets within both the Simulink and the Stateflow portions of a model. When using the Reactis for C Plugin, coverage targets are also tracked in the C code portions of models (S-Functions and Stateflow custom C code). Additionally, when using the Reactis for EML Plugin, coverage targets are tracked within any Embedded MATLAB code used by a model. The metrics discussed in this chapter may be visualized using Simulator and are central to test generation and model validation using Tester and Validator.

6.1 Simulink-Specific Metrics

The Simulink-specific metrics include the following: conditional subsystem coverage, branch coverage, and lookup table coverage.

6.1.1 Conditional Subsystem Coverage

A (conditional) subsystem is deemed covered if has been executed at least once in some simulation step. In general, every subsystem within a Simulink diagram executes during every simulation step. However, conditional subsystems may be disabled during a simulation step and hence not execute.

6.1.2 Branch Coverage

A number of different Simulink blocks contain targets included in the branch coverage measure. Each block in this group has the characteristic that the set of all possible outcomes of evaluating the block may be partitioned into well-defined sets of mutually exclusive outcomes. For example, the possible outcomes of evaluating a Logical Operator block are true or false. Therefore, for each Logical Operator block, we consider the true outcome and the false outcome to be the two branches associated with the block. Each branch becomes a coverage target in the branch-coverage metric.

The following discusses branch coverage in more detail. Reactis Tester aims to exercise all such branches in the test suites it generates. Reactis Simulator includes several ways to
track the branches that have been exercised. In particular, in model diagrams, every block that includes branches is drawn in a way to show each branch and uncovered branches are drawn in red. We now list the blocks included in the branch-coverage metric; for each block, we describe its associated branches and how coverage information for the branches is drawn in Simulator.

**Logical Operator.** The branches are:

1. the block evaluates to true
2. the block evaluates to false

**Relational Operator.** The branches are:

1. the block evaluates to true
2. the block evaluates to false

![Figure 6.1: Branches in Relational and Logical Operator blocks.](image)

**Switch.** Let $C$ be the condition for passing the first input of a Switch block; i.e. $C$ is $u_2 \geq \text{threshold}$, $u_2 > \text{threshold}$, or $u_2 \neq 0$. Then the branches are:

1. $C$ evaluates to true.  
2. $C$ evaluates to false.

**Multiport Switch.** If a Multiport switch has $n$ data inputs, then it is considered to have $n$ branches, one for each data input. Branch $i$ is covered when the control input has value $i$ during evaluation of the block.

**Dead Zone.** Let $X$ be the input to, $L$ the lower limit of, and $U$ the upper limit of a Dead Zone block. Then the branches are:

1. $X \leq L$
2. $L < X < U$
3. $X \geq U$

**Saturation.** Let $X$ be the input to, $L$ the lower limit of, and $U$ the upper limit of a Saturation block. Then the branches are:
6.1. SIMULINK-SPECIFIC METRICS

Figure 6.2: Branches in Switch and Multiport Switch blocks. Each branch target in a Switch block or a Multiport Switch block is drawn as a line from an input of the block to the output of the block. The line from an input represents the branch where that input is taken as output for the block.

1. \( X < L \)
2. \( L \leq X \leq U \)
3. \( X > U \)

Figure 6.3: Branches in Dead Zone and Saturation blocks. Each line segment represents a branch.

**Discrete-Time Integrator.** Let \( V \) be the integrated value, \( L \) the lower limit of, and \( U \) the upper limit of a Discrete-Time Integrator block. Then the branches are:

1. \( V < L \)
2. \( L \leq V \leq U \)
3. \( V > U \)

**If.** The Block Parameters dialog for the If block includes a “Show else condition” check box that determines whether or not the block includes an output port for the else case of the block. The definition of the branches for this block differ slightly depending on the value of this setting.
• If an If block has $n$ outputs and the “Show else condition” check box is selected, then the block is considered to have $n$ branches, one for each output. Branch $i$ is covered when the system connected to output $i$ executes.

• If an If block has $n$ outputs and the “Show else condition” check box is not selected, then the block is considered to have $n + 1$ branches, one for each output and one for the implicit else case. For branches 1 to $n$, branch $i$ is covered when the system connected to output $i$ executes. Branch $n + 1$ is covered when the If block executes, but no output fires (that is when the implicit else case is taken).

Switch Case. The Block Parameters dialog for the Switch Case block includes a “Show default case” check box that determines whether or not the block includes an output port for the default case. Reactis tracks a branch for the default case whether or not this check box is selected. Additionally the parameters of this block define a sequence of integer sets $S_1, S_2, ..., S_n$. Each set is associated with an output of the block. Reactis tracks a branch target for each element of each set $S_i$. A branch value $n$ in set $S_i$ is covered if the control input to the block is $n$ and the outport associated with $S_i$ fires.

Some sets may contain only a single value.
6.1. SIMULINK-SPECIFIC METRICS

Enable. Let $X$ be the input to an Enable port of the subsystem $S$ in which the Enable block resides. Then the branches are:

1. $X > 0$ - subsystem $S$ executes
2. $X \leq 0$ - subsystem $S$ does not execute

MinMax. If a MinMax block has $n$ inputs, then it is considered to have $n$ branches, one for each input. Branch $i$ is covered when input $i$ is selected as the minimum or maximum value of inputs to the block.

Relay. The branches are

1. The relay is on.
2. The relay is off.

Abs and Sign blocks. Let $X$ be the input to an Abs block. Then the branches are:

1. $X \geq 0$
2. $X < 0$

Let $X$ be the input to a Sign block. Then the branches are:

1. $X > 0$
2. $X = 0$
3. $X < 0$
6.1.3 Lookup Table Coverage

Reactis supports coverage tracking for the Lookup Table, Lookup Table (2-D) blocks, Pre-Lookup, Interpolation-Nd and some Lookup Table (n-D) blocks. Intuitively a coverage target will be allocated for each interval specified by the input settings of a table. To view the coverage information for a table in Simulator, right-click on the table and select View Coverage Details.

### 6.1.3.1 1-D Tables

If a Lookup Table block (one-dimensional) has inport \( u \) and “Vector of input values” \( [u_1, u_2, ..., u_n] \) then the block will have the following targets:

<table>
<thead>
<tr>
<th>Condition</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u &lt; u_1 )</td>
<td>( u_1 \leq u &lt; u_2 )</td>
</tr>
<tr>
<td>( u_2 \leq u &lt; u_3 )</td>
<td>( u_3 \leq u &lt; u_4 )</td>
</tr>
<tr>
<td>( u_n-1 \leq u &lt; u_n )</td>
<td>( u \geq u_n )</td>
</tr>
</tbody>
</table>
6.1. SIMULINK-SPECIFIC METRICS

Figure 6.10: Coverage targets for a 1-D Lookup Table with input $u$ and breakpoints $[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]$. This table may be displayed in Simulator by right clicking on the lookup table block and selecting View Coverage Details.
6.1.3.2 2-D Tables

If a Lookup Table (2-D) block has inputs \( u1 \) and \( u2 \), “Row index input values” \([r_1, r_2, ..., r_m]\), and “Column index input values” \([c_1, c_2, ..., c_n]\) then the block will have the following targets:

\[
\begin{array}{cccc}
u1 < r_1 \text{ and } u2 < c_1 & u1 < r_1 \text{ and } c_1 < u2 < c_2 & u1 < r_1 \text{ and } c_2 < u2 < c_3 & \ldots & u1 < r_{n-1} \text{ and } u2 < c_n & u1 < r_1 \text{ and } u2 > c_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
r_1 \leq u1 < r_2 \text{ and } u2 < c_1 & r_1 \leq u1 < r_2 \text{ and } c_1 \leq u2 < c_2 & r_1 \leq u1 < r_2 \text{ and } c_2 \leq u2 < c_3 & \ldots & r_{n-1} \leq u2 < r_n \text{ and } u2 > c_n & r_1 \leq u1 < r_2 \text{ and } u2 > c_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
r_2 \leq u1 < r_3 \text{ and } u2 < c_1 & r_2 \leq u1 < r_3 \text{ and } c_1 \leq u2 < c_2 & r_2 \leq u1 < r_3 \text{ and } c_2 \leq u2 < c_3 & \ldots & r_2 \leq u1 < r_3 \text{ and } u2 > c_n & r_2 \leq u1 < r_3 \text{ and } u2 > c_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
... & ... & ... & ... & ... & ... \\
\end{array}
\]

\[
\begin{array}{cccc}
r_m-1 \leq u1 < r_m \text{ and } u2 < c_1 & r_m-1 \leq u1 < r_m \text{ and } c_1 \leq u2 < c_2 & r_m-1 \leq u1 < r_m \text{ and } c_2 \leq u2 < c_3 & \ldots & r_m-1 \leq u1 < r_m \text{ and } u2 > c_n & r_m-1 \leq u1 < r_m \text{ and } u2 > c_1 \\
\end{array}
\]

\[
\begin{array}{cccc}
u1 \geq r_m \text{ and } u2 < c_1 & u1 \geq r_m \text{ and } c_1 \leq u2 < c_2 & u1 \geq r_m \text{ and } c_2 \leq u2 < c_3 & \ldots & u1 \geq r_m \text{ and } u2 > c_n & u1 \geq r_m \text{ and } u2 > c_1 \\
\end{array}
\]

Figure 6.11: Coverage targets for a 2-D Lookup Table with inputs \( u1 \) and \( u2 \). The table shown may be displayed in Simulator by right-clicking on the lookup table block and selecting View Coverage Details.

6.2 Stateflow-Specific Metrics

The Stateflow metrics are defined with respect to the graphical syntax of Stateflow. Among other things, Stateflow diagrams contain states, transition segments, and junctions. Each transition segment in turn may have a label that includes all or some of the following: an event, a condition, a condition action, and a transition action. A segment’s condition action is executed whenever the segment’s condition evaluates to true. A transition consists of a sequence of segments leading from one state to another. The transition will fire when the condition on each segment in the sequence evaluates to true. A segment’s transition action is executed only when it is included in such a firing transition. Highlighting of coverage in Stateflow diagrams is shown in Figure 6.12. Note that in addition to the Stateflow-specific metrics, Decision, Condition, MC/DC and MCC coverage targets may be associated with transition segments.
6.2. STATEFLOW-SPECIFIC METRICS

6.2.1 State Coverage

The targets in this metric are states; a state is covered if it has been entered at least once.

6.2.2 Condition Action Coverage

The targets in this metric are transition segments; a segment is deemed to be covered if its condition action has been executed at least once, or, if the segment has no condition action, if its condition has evaluated to “true” at least once. Note that a segment with an empty condition is assumed to have a condition that always evaluates to “true” when the segment is considered for inclusion in a firing transition during model execution.

6.2.3 Transition Action Coverage

The targets in this metric are transition segments that have transition actions; such a segment is deemed to be covered if its transition action has been executed at least once. Note that if a segment has no transition action, it is ignored by this metric.
6.2.4 Child State Exit via Parent Transition

Consider the Stateflow diagram in Figure 6.12. The state \textit{On} has three child states: \textit{Inactive}, \textit{Active}, and \textit{Init}. \textit{Child State Exit via Parent Transition (CSEPT)} coverage tracks whether the transition from \textit{On} to \textit{Off} has fired while \textit{On} is in each of its child states. So in this case there are three CSEPT targets. Namely the transition from \textit{On} to \textit{Off} fires when:

1. \textit{On} is in state \textit{Inactive}
2. \textit{On} is in state \textit{Active}
3. \textit{On} is in state \textit{Init}

More precisely, a Stateflow state \( S \) has the following CSEPT targets:

- If \( S \) is a top-level state (it has no parent), then \( S \) has no CSEPT targets.
- If \( \text{Parent}(S) \) is the parent of \( S \), then for every transition that causes \( \text{Parent}(S) \) to exit, \( S \) has a corresponding CSEPT target.

If your model has states nested deeper than one level (e.g. \( \text{Parent}(S) \) has a parent for some state \( S \)) or a transition causing \( \text{Parent}(S) \) to exit has multiple transition segments, then you can adjust the definition of CSEPT targets with two settings that you specify in the Coverage tab of the Info File Editor (see Section 5.10). These settings specify which states and transitions will be paired with a state to form CSEPT targets. In the following:

- \( \text{PairStates}(S) \) is the set of states paired with \( S \) to define the CSEPT targets of \( S \)
- \( \text{ExitTrans}(S') \) is the set of either transition segment sequences or single transition segments that cause a state \( S' \) to exit.

The two settings you control indicate the following definitions for \( \text{PairStates} \) and \( \text{ExitTrans} \):

**CSEPT States** can be set to All ancestors or Parent only:

- \textbf{All ancestors} \( \text{PairStates}(S) = \text{Ancestors}(S) \) where \( S'' \in \text{Ancestors}(S) \) if \( S'' = \text{Parent}(S) \) or there exists \( S'' \in \text{Ancestors}(S) \) such that \( S'' = \text{Parent}(S') \).

- \textbf{Parent only} \( \text{PairStates}(S) = \text{Parent}(S) \).

**CSEPT Transitions** can be set to Full transition paths or First segment only:

- \textbf{Full transition paths} \( \text{ExitTrans}(S') = \) all transition segment sequences that form a complete transition that can fire and cause \( S' \) to exit.

- \textbf{First segment only} \( \text{ExitTrans}(S') = \) all transition segments that are the first segment of a transition sequence that can fire and cause \( S' \) to exit.

These options yield a parameterized definition of CSEPT targets. A Stateflow state \( S \) has the following CSEPT targets:

- If \( S \) is a top-level state (it has no parent), then \( S \) has no CSEPT targets.
- For every state \( S' \) in \( \text{PairStates}(S) \), for every transition \( T \) in \( \text{ExitTrans}(S') \), \( S \) has a corresponding CSEPT target.
6.3 Generic Metrics

Generic coverage metrics define targets which may appear in the Simulink, the Stateflow, the Embedded MATLAB \(^2\), or the C code \(^3\) portions of a model. There are five generic coverage metrics supported by Reactis: (1) Decision coverage, (2) Condition coverage, (3) Modified Condition/Decision Coverage (MC/DC), (4) Multiple Condition Coverage (MCC) and (5) Boundary coverage. These metrics are all are based on well-known coverage metrics of the same names developed for measuring coverage of source code.

6.3.1 Decision, Condition, MC/DC and MCC Metrics

Reactis includes facilities for generating tests to meet the Decision, Condition, Modified Condition/Decision Coverage (MC/DC) and Multiple Condition Coverage (MCC) requirements. Each of these metrics involve boolean-valued structures (conditions, decisions) within a model; understanding these metrics requires that the notions of conditions and decisions be defined precisely. As the notions were first developed in the context of traditional programming languages (e.g. C, Ada), the next paragraphs first review their traditional definitions. The adaptations of these notions to Simulink and Stateflow are then discussed.

In software testing, a **decision** is a boolean-valued expression used to determine which execution path to follow, and a **condition** is a boolean-valued subexpression of a decision which cannot be broken into smaller boolean subexpressions because it does not contain any boolean operators. Decisions are typically constructed by using boolean operators to combine several conditions. For example, consider the following statement from a C program:

```c
if ((x > 0) && (y > 0))
    z = 1;
else
    z = 2;
```

Here \(((x > 0) \&\& (y > 0))\) is a decision, since the value of the expression determines whether \(z\) is assigned the value 1 or 2. Both \((x > 0)\) and \((y > 0)\) are conditions, since they are boolean-valued expressions that cannot be broken into smaller boolean expressions.

Traditional Decision coverage may now be defined as follows. Each decision in a program gives rise to two targets: the evaluation of the decision to true, and to false; a program is fully covered by a test suite when each target has been covered, i.e. each decision has evaluated to both true and false. Condition coverage is defined very similarly; the difference is that each condition evaluates to both true and false, rather than each decision.

MC/DC is somewhat more complex to define. It was introduced by John J. Chilenski of Boeing in the early 90s; the definitive research paper was published by Chilenski and Steve Miller, of Rockwell-Collins, in 1994. MC/DC is the level of testing mandated by the Federal Aviation Administration (FAA) in its DO-178/B guidelines for the “most safety-critical” components of aviation software. The metric was subsequently adopted in the ISO 26262 safety standard for automotive software. The MC/DC targets in a program are the conditions; a condition \(C\) in decision \(D\) is covered by a test suite if there are two test steps \(X\) and \(Y\) (not necessarily consecutive) in the suite such that:

\(^2\)Embedded MATLAB coverage tracking becomes enabled when using the Reactis for EML Plugin; otherwise coverage is not tracked in the Embedded MATLAB portions of a model.

\(^3\)C code coverage tracking becomes enabled when using the Reactis for C Plugin; otherwise coverage is not tracked in the C code portions of a model.
• C evaluates to a different truth value (true or false) in step X than in step Y; and

• each condition other than C in D evaluates to the same truth value in both step X and step Y; and

• D evaluates to a different truth value in step X than in step Y.

In other words, each condition must be shown to independently affect the outcome of its enclosing decision.

Multiple Condition Coverage (MCC, also known as Condition Combination Coverage) tracks one target for each combination of values to which the conditions in a decision evaluate. For a decision with N conditions this will create $2^N$ MCC targets. A decision $D = C_1 \&\& C_2 \&\& C_3$ will need 8 MCC targets to track all possible combinations of C1, C2, and C3 (each row in the table below represents one MCC target):

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Short-circuiting of boolean expressions can affect the difficulty of achieving full MC/DC or MCC coverage of a program. A short-circuited operator avoids evaluating all conditions of a decision if the outcome is determined after evaluating only a subset of the conditions. For example, a short-circuited “and” operator would not evaluate its second argument if the first evaluates to false. Virtually all programming languages use short-circuiting for reasons of efficiency. Without going into the technical details, it can be said that achieving full MC/DC coverage is easier if short-circuited boolean operators are used. For MCC coverage, short-circuited expressions generate a reduced number of coverage targets compared to non-short-circuited expressions. The $D = C_1 \&\& C_2 \&\& C_3$ decision creates only 4 (as opposed to 8) MCC targets (the x characters in the table below represent “don’t care” terms - conditions that were not evaluated due to short-circuiting and whose value therefore can not be recorded):

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>x</td>
<td>x</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>x</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

Section 4.8.1 explains how Reactis may be instructed to treat Simulink / Stateflow boolean operators as short-circuited.

In order to adapt the notions of Decision, Condition, MC/DC and MCC coverage to Simulink / Stateflow, it suffices to define what the conditions and decisions in a model are. These are as follows.

• For Logical Operator blocks, Reactis supports two alternative modes for tracking Decision, Condition, MC/DC and MCC coverage. The mode is set from the Coverage pane
of the Info File Editor (see Section 5.10). The two modes are single-block and multi-block. In the former, a decision is a single logical operator block; while in the latter, several logical operator blocks may be grouped into a single decision. To understand the difference between the two modes, consider the following model fragment:

In single-block mode, this fragment includes three separate decisions; each with its own conditions, MC/DC and MCC targets:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A or B</td>
<td>A, B</td>
</tr>
<tr>
<td>C or D</td>
<td>C, D</td>
</tr>
<tr>
<td>op1 and op2</td>
<td>op1, op2</td>
</tr>
</tbody>
</table>

When Multi-Block Decision Coverage is enabled, these three operators are combined into a single decision:

<table>
<thead>
<tr>
<th>Decisions</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A or B and C or D</td>
<td>A, B, C, D</td>
</tr>
</tbody>
</table>

When Multi-Block Decision Coverage is enabled, hovering over a Logical Operator block will show arrows indicating the group to which the block belongs. If no arrows appear, then the block is not part of any group.

The definition of decisions and conditions for the two modes are as follows.

**Single-Block.** The output of each Logical Operator block is a decision and its inputs constitute the conditions within the decision.

**Multi-Block.** Multiple Logical Operator blocks that are directly connected are grouped into a single decision. The inputs to blocks in the group that are not connected to the outputs of other blocks in the group constitute the conditions of the decision. When Multi-Block Decision Coverage is enabled, the following rules are used to combine the Logical Operator blocks of a model into groups, each of which constitutes a single decision:

- Groups may only contain Logical Operator blocks.
- Groups may not cross subsystem boundaries.
- Each group has exactly one exiting signal line. The output port of the block at which the exiting signal line originates will be marked as the decision. A red dot on the output port indicates that the decision has not evaluated to both true and false. The coverage details table for the decision may be invoked by right-clicking on the output port and selecting View Coverage Details. Hovering over the block causes blue lines to be drawn from each condition in the decision to the output port.
– Each group may have multiple incoming signal lines. The input ports of the blocks where the entering signal lines end will be marked as the conditions of the decision. A red dot on one of the inports implies that the port has an uncovered condition or MC/DC target.

– The set of Logical Operator blocks $L$ within a subsystem is partitioned into a set of groups as follows. If the output of a Logical Operator block $l \in L$ feeds into any block other than another Logical Operator block or the output of $l$ is monitored by a Reactis test point, then $l$ is the root of a group. The set of blocks contained in the group rooted at a block $r$, denoted $G(r)$ is defined as:

- $r \in G(r)$
- If $l \in L$ and all branches of the output of $l$ feed into a block $l' \in G(r)$ and the output of $l$ is NOT monitored by a Reactis test point, then $l \in G(r)$

• The “if expression” and “elseif expressions” within an If block in Simulink are decisions. In this case, the atomic boolean-valued sub-expressions are the conditions.

• Recall that Stateflow transitions may have a label of the form:

  \[ \text{event[guard]{condition action}}/\text{transition action}. \]

Note that for this discussion we use the term “guard” rather than “condition” for the boolean expression in braces to avoid confusion with the use of that word in context of MC/DC, Decision, and Condition coverage. The \text{event[guard]} combinations on Stateflow transitions are considered Decisions. They evaluate to true if the event is present and the guard is true, and to false otherwise. The conditions within the decision associated with a transition segment are:

– If the label includes an event, then the event is a condition. The condition is “true” if it is present and “false” otherwise.

– The atomic Boolean-valued expressions in the guard are also conditions.

• When using the Reactis for C Plugin, additional decisions are tracked as described in Chapter 16.

• When using the Reactis for EML Plugin, additional decisions are tracked as describe in Chapter 17.

Coverage information for Decision, Condition, and MC/DC coverage is rendered as shown in Figure 6.13. Right-clicking on an outport of a Logical Operator block and selecting View Coverage Details gives additional information.

### 6.3.2 Boundary Value Coverage

Boundary Value Coverage tracks whether a data item assumes values of interest for a particular block. In the case of top-level model imports, test points, and configuration variables, these are the boundary values of the item’s domain of possible values. In the case of a Relational Operator block, the values of interest occur when the two inputs to the block are equal or very close.
### 6.3.2.1 Boundary Values for Inports, Test Points, and Configuration Variables

The boundary values tracked for an inport, test point, or configuration variable are determined by its associated type constraint as shown in Table 6.1. If an inport or configuration variable has a type not shown in the table, then it has no boundary value targets.

---

#### 6.3.2.2 Boundary Values for Relational Operators

Reactis also tracks boundary value coverage for the Relational Operator block. For a relational operator block with integer inputs $a$ and $b$, the boundary value targets are:

- $a = b$
- $a = b + 1$
- $a = b - 1$

For a relational operator block with floating point inputs $a$ and $b$, the boundary value targets are:

- $a = b$

---

4Recall that types are associated with top-level imports using the Port Types tab of the Reactis Info File Editor (described in Chapter 5).
<table>
<thead>
<tr>
<th>RSI Type</th>
<th>Boundary Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t[i,j]$, where $t$ is double, single, sfix*, or ufix*</td>
<td>if $i &lt; 0$ and $j &gt; 0$ then $i, 0.0, j$; otherwise $i, j$</td>
</tr>
<tr>
<td>$t[i,j]$, where $t$ is an integer type</td>
<td>if $i &lt; 0$ and $j &gt; 0$ then $i, i + 1, 0, j - 1, j$; otherwise $i, i + 1, j - 1, j$</td>
</tr>
<tr>
<td>$t{e_1,\ldots e_n}$</td>
<td>$e_1, \ldots e_n$</td>
</tr>
<tr>
<td>$t[i:j:k]$</td>
<td>if there exists a positive integer $l$ such that $i + l \times j = 0.0$ then $i, i + j, 0.0, k - j, k$; otherwise $i, i + j, k - j, k$</td>
</tr>
<tr>
<td>$t\ delta\ [i,j]$</td>
<td>boundary values of $t$</td>
</tr>
<tr>
<td>boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>int8</td>
<td>-128, -127, 0, 126, 127</td>
</tr>
<tr>
<td>int16</td>
<td>-32768, -32767, 0, 32766, 32767</td>
</tr>
<tr>
<td>int32</td>
<td>-2147483648, -2147483647, 0, 2147483646, 2147483647</td>
</tr>
<tr>
<td>uint8</td>
<td>0, 1, 254, 255</td>
</tr>
<tr>
<td>uint16</td>
<td>0, 1, 65534, 65535</td>
</tr>
<tr>
<td>uint32</td>
<td>0, 1, 4294967294, 4294967295</td>
</tr>
<tr>
<td>sfix*</td>
<td>let S, B, W be the total slope, bias and width (in bits) of the type, $i = - (2^{W-1}) \times S + B$ and $j = (2^{W-1} - 1) \times S + B$: if $i &lt; 0$ and $j &gt; 0$ then $i, 0.0, j$; otherwise $i, j$</td>
</tr>
<tr>
<td>ufix*</td>
<td>let S, B, W be the total slope, bias and width (in bits) of the type, $i = B$ and $j = (2^{W - 1}) \times S + B$: if $i &lt; 0$ and $j &gt; 0$ then $i, 0.0, j$; otherwise $i, j$</td>
</tr>
<tr>
<td>double</td>
<td>0.0</td>
</tr>
<tr>
<td>single</td>
<td>0.0</td>
</tr>
<tr>
<td>Simulink.IntEnumType ${e_1,\ldots e_n}$</td>
<td>$e_1, \ldots e_n$</td>
</tr>
</tbody>
</table>
6.3. GENERIC METRICS

\[ a < b \text{ and } (b - a) \leq \text{reltol} \times |a| \]
\[ b < a \text{ and } (a - b) \leq \text{reltol} \times |a| \]

where \text{reltol} is the relative tolerance specified in the Coverage tab of the Info File Editor. Note that boundary value coverage for relational operators has the potential to introduce a large number of very hard to cover targets. Since some users might prefer to not track such targets, this functionality is disabled by default and can be enabled from the Coverage tab of the Info File Editor.

6.3.2.3 Boundary Values for Integer Saturation

Reactis can be configured to track whether or not saturation has occurred for blocks which are set to saturate on integer overflow. Reactis will create up to three boundary value targets for each block with saturate on integer overflow enabled:

- **Saturated Max** – the block has received an input which caused saturation at the maximum output value.
- **Saturated Min** – the block has received an input which caused saturation at the minimum output value.
- **No Saturation** – the block has received an input which did not cause saturation to occur.

The tracking of integer saturation is controlled from the Coverage tab of the Info File Editor. When enabled, integer saturation is tracked for the following blocks:

- Abs
- Add
- Bias
- Divide
- Dot Product
- Math Function
- MinMax
- Product
- Product of Elements
- Subtract
- Sum
- Sum of Elements
- Unary Minus
- Switch
- Multi-Port Switch
- Index Vector
- Data Type Conversion
6.4 Validator-Related Targets

See Chapter 9 for a description of the two Validator-related targets: assertions and user-defined targets.
6.5 Excluding Coverage Targets

Reactis supports three different ways to disable coverage tracking for a subset of targets. When a target is excluded using any of the three mechanisms, Reactis Tester will not try to generate a test to exercise the target and Reactis Simulator will not report the target as uncovered.

6.5.1 Disabling a Coverage Metric

In some cases, tracking a certain metric may not be desirable. For example, if a model contains no lookup tables, you may wish to not use lookup table coverage when working with the model in Reactis. Or, if accomplishing 100% MCC coverage requires an excessive number of tests, you may wish to not track that metric. To help you focus on the set of metrics of highest interest to you, Reactis lets you disable a whole coverage metric.

To disable a coverage metric, do the following:

- make sure Reactis Simulator is disabled
- open the Coverage Metrics tab in the Reactis Info File Editor (select Edit → Coverage Metrics...)
- switch the desired metric to "Disabled"

Disabling a coverage metric will cause it to not be shown in any of the various places where Reactis lists coverage metrics, including the Tester launch and run dialogs, the Coverage Summary dialog, the Coverage Report Browser, and exported coverage reports. Also, Reactis Tester will not target disabled metrics when generating tests.

Figure 6.15: Disabling all branch coverage targets via the Info File Editor (Edit → Coverage Metrics)
6.5.2 Excluding a Subsystem

Reactis lets you disable coverage tracking for all targets in a given subsystem of a model. For example, it may not be desirable to track coverage for a referenced model or library system if that portion of your model is being tested by other unit tests.

When Simulator is disabled, you can enable or disable coverage tracking for a subsystem by right-clicking on the subsystem in the Reactis hierarchy panel and selecting the Coverage Tracking entry which has the following sub-menus:

Enable Enable coverage tracking for targets in the subsystem.

Disable Disable coverage tracking for targets in the subsystem.

Inherit The coverage tracking for the subsystem is the same as its parent’s setting.

Reset to inherited... Set the coverage tracking setting for the subsystem and all its descendants to inherited.

Cumulative Enable cumulative coverage tracking for all library subsystem blocks and model references within the selected subsystem. This option is explained in Section 6.6.

By default, coverage tracking is enabled for the top-level of the model and set to inherit for all other subsystems. As shown in Figure 6.16, visual cues in the hierarchy panel indicate the current coverage tracking setting for each subsystem. An X mark on the icon to the left of a subsystem name indicates that coverage tracking is disabled, whereas a check mark indicates that coverage tracking is enabled. If the icon has neither a check nor an X, then the subsystem is set to inherit its coverage tracking setting. If a subsystem currently has coverage tracking disabled or it inherits a disabled setting, then the subsystem name is grayed out (drawn in a lighter color).

Turning off coverage tracking for a subsystem has the following effects:

- Tester will not attempt to cover targets within the subsystem
- Simulator will not track coverage for targets in the subsystem, so the targets will not be:
  - included in the Coverage Summary dialog,
  - included in reports displayed or exported by the Coverage Report Browser,
  - highlighted in the main panel

6.5.3 Excluding Individual Targets

Excluding individual targets from coverage tracking allows fine-grained control if a model includes a few specific targets that cannot be exercised. Excluding such targets from coverage tracking (after confirming their unreachable status) can help you achieve the goal of 100% coverage of reachable targets.

Reactis offers two alternative ways to exclude a coverage target. If you simply exclude a target, Reactis will not attempt to exercise the target when generating tests and it will not report it as covered or uncovered. The second way to exclude a target is to exclude and monitor a target. This method lets you assert that a target is unreachable and therefore should not be included in the coverage reports; but, Reactis will also monitor the target and alert you if the
target is ever exercised – that is, you will be notified if your claim that the target is unreachable is incorrect. Reactis accomplishes this by automatically creating a Validator assertion for each excluded and monitored target to flag if the target is ever exercised. If such an assertion is violated (either while generating tests in Tester or running the model in Simulator), Reactis will report the violation and Simulator can be employed to investigate how the presumably unreachable target got covered.

All target types tracked by Reactis can be excluded. The method to exclude a target differs slightly by target type.

For Decision, Condition, MC/DC, MCC, CSEPT, Lookup table and input boundary value targets (see Figure 6.17):

1. Right-click on the respective Simulink block, Stateflow transition or C or EML code fragment.
2. Select View Coverage Details.
3. In the resulting dialog, right-click on the test/step information of the specific target (e.g. "-/-" or "3/17").
4. Select the "Track Coverage" menu item.

For all other targets right-click on the block, Stateflow transition, or C or EML code fragment, select Track Coverage and select the appropriate target from the sub-menu (see Figure 6.18). A check-mark in the sub-menu listing coverage targets indicates that coverage is currently being tracked for that target.

Selecting Track Coverage brings up the dialog shown in Figure 6.19. This dialog offers three choices for the coverage exclusion status:

**Track coverage for this target** If this option is selected, coverage is tracked as usual for the target.

**Exclude target from coverage tracking** If selected, coverage is not tracked for the target:

- If the target has a specific visual representation (e.g. a branch in a Switch block is represented by a line from an inport to an outport), the visual representation will be drawn in blue (instead of black or red).
CHAPTER 6. REACTIS COVERAGE METRICS

Figure 6.17: Changing the exclusion status of a condition target within a decision of a Stateflow transition.

Figure 6.18: Changing the exclusion status of a branch target within a Switch block

- The target’s parent (Simulink block or Stateflow transition) will also be highlighted in blue if all non-excluded targets associated with the block have been covered. As long as any uncovered sub-targets remain, the parent will be highlighted in red.

- The target will not be counted in the coverage summary.

- The target will be added to the list of targets in the Excluded Coverage Targets pane of the Info File Editor (invoked by Edit → Excluded Coverage Targets).

- Whenever the coverage status is displayed for the target (e.g. in coverage reports and when hovering) the status will be “excluded”.

**Exclude target from coverage tracking and monitor via assertion** The same rules apply as for the case above, with the following additions:
• An assertion will be automatically created for the target. The assertion is included in the "Assertion" count in the coverage summary.

• To indicate the "monitored" status of the target, its status will be shown as "[excluded]" in the coverage report, list of excluded targets and when hovering.

If an excluded and monitored target gets covered, this is recorded as follows:

• The associated assertion changes to "violated" status.
• The target and parent block/transition are highlighted in yellow in the main panel.
• In the coverage report, the list of excluded targets, and when hovering the status will be listed as "[t/s]" where t/s is the test/step number in which the target was covered.
• The associated assertion will be listed as violated in the test execution report.

As shown in Figure 6.19, when either Exclude target from coverage tracking or Exclude target from coverage tracking and monitor via assertion is selected, the comment box within the dialog is enabled. The comments are saved in the .rsi file for future reference.

6.6 Cumulative Subsystem Coverage

When a model contains multiple instances of a reference system (either a library subsystem block or model reference), Reactis lets you track coverage cumulatively. Cumulative coverage tracking shares targets between all instances of the same subsystem, which effectively treats the multiple instances as if they were a single instance.

When Simulator is disabled, you can enable cumulative coverage tracking for a subsystem by right-clicking on it in the Reactis hierarchy panel and selecting Coverage Tracking → Cumulative. This will change all library subsystem blocks and model references within the selected subsystem to use cumulative coverage tracking. Cumulatively tracked subsystems are distinguished by a C superimposed on the upper right corner of their icon. A special hierarchy tree node named Cumulatively tracked systems serves as a collection point where all
subsystems whose coverage is cumulatively tracked can be inspected. Figure 6.20 shows a hierarchy tree with three subsystems set to use cumulative coverage tracking.

When you right-click on a subsystem and select Coverage Tracking, the Cumulative entry in the sub-menu will be checked if cumulative coverage tracking is turned on. When cumulative coverage tracking is turned on for a subsystem, selecting Coverage Tracking → Cumulative turns it off. Turning off cumulative coverage tracking for a subsystem will cause every instance of the subsystem to be tracked separately. Cumulative coverage for all descendants of a subsystem can be turned off by right-clicking on it and selecting Coverage Tracking → Reset to inherited. Cumulative coverage can also be turned off throughout the entire model by right-clicking on the top node in the hierarchy tree and selecting Coverage Tracking → Enable all.

If a subsystem containing Validator objectives is switched to cumulative coverage then those objectives are no longer tracked because coverage for that instance is no longer tracked individually. Such objectives will be shown grayed-out.

You can add Validator objectives within the cumulative section of the hierarchy tree. Such objectives will then be tracked for all instances of the corresponding system. For assertions, this means that a violation will be registered if an assertion is violated in any instance. User-defined targets will show as covered if they are covered in any instance.

When switching a subsystem to cumulative coverage and at least one instance of that system contains Validator objectives then Reactis will present a warning, allowing you to automatically create copies of those objectives in the cumulative section.
Chapter 7

Reactis Simulator

Simulator provides an array of features — including single and multi-step forward and backward execution, breakpoints, and simulations driven by Tester-generated tests or user inputs — for simulating Simulink / Stateflow models (and C code if using the Reactis for C Plugin). The tool also allows visual tracking of coverage data and the values data items in the model assume during simulation.

Figure 7.1: The Reactis Simulator toolbar controls.

Figure 7.1 contains an annotated screen shot of a portion of the top-level Reactis window when Simulator is enabled. Some of the buttons and pull-down menus on the leftmost part of the window have been elided; Chapter 4 contains descriptions of these items. The next section describes the labeled items in Figure 7.1, while the section following discusses the Simulator-related menu entries. The subsequent sections discuss the different modes for generating inputs during simulation, ways to track data values, how to monitor model coverage, importing and exporting test suites, and the different model highlighting styles used by Simulator.

7.1 Labeled Window Items

1. Disable Reactis Simulator.

2. Enable Reactis Simulator.
3. Reset the simulation; the model is returned to the start state, and coverage information is appropriately reset.

4. Take \( n \) steps back, where \( n \) is specified by window item 10. Coverage information is updated appropriately upon completion of the last backward step.

5. Take one step back. Coverage information is updated appropriately.

6. Advance by one mini-step. In Simulink, a mini-step evaluates the next block in the evaluation order. In Stateflow, a mini-step evaluates the next transition segment condition or transition action. In C, a mini-step executes a single C statement, stepping into a function when at a function call.

7. Advance forward by one full step; that is, values are read on the top-level inports, the model’s response is computed and values are written to the top-level outports. If a step has been partially computed using the step-into button (window item 6), then execution picks up with the current partially computed step and continues until the end of the step, at which point values are written to the top-level outports.

8. Execute \( n \) slow simulation steps, where \( n \) is specified by window item 10. The diagram in the main panel (window item 17) is updated during simulation to reflect the currently active Simulink block, Stateflow state / transition, or C statement. When Coverage \( \rightarrow \) Show Details is selected, coverage targets will change from red to black as they are covered during the simulation run. If a single test is selected and the end of the test is reached before \( n \) steps execute, then simulation stops. When executing multiple tests (including an entire suite) the tests are executed one after another and simulation stops when \( n \) steps have executed or the end of the test suite is reached.

When a slow simulation is running, clicking this button pauses the simulation.

9. Execute \( n \) fast simulation steps, where \( n \) is specified by window item 10. The diagram in the main panel (window item 17) is not updated while the simulation is in progress but is updated when simulation halts. If the end of the current test or test suite is reached then simulation halts.

When a fast simulation is running, clicking this button pauses the simulation.

10. This window item determines how many steps are taken when buttons corresponding to window items 4, 8, or 9 are clicked. When the Source-of-Inputs dialog (window item 11) is set to a test or test suite, the number of steps may be set to 0 to indicate that the entire test or test suite should be executed.

11. The Source-of-Inputs dialog determines what values are fed into inports to drive a simulation: random values, values from the user, or values from a test. See Section 7.4 for details.

12. Create a new, empty test suite. The name of the .rst file containing the suite is initially “unnamed.rst” and is displayed in the title bar of the Reactis window.

13. Open a dialog for selecting a test suite (.rst file) to be loaded into Simulator. After it is loaded, the test suite’s name is displayed in the title bar, and the tests are listed in the Source-of-Inputs dialog (window item 11).
14. Save the current test suite.

15. View Reactis Simulator help.

16. The model hierarchy panel (not shown explicitly) supports the structure-based navigation of a model, as described in Section 4.1. Right-clicking on an item in the navigation panel brings up a menu that also allows you to view data items and set breakpoints. Data viewing is covered in more detail in Section 7.5. Breakpoints may be set by right-clicking on a subsystem or Stateflow state in the hierarchy panel and selecting Toggle Breakpoint. The name in the hierarchy panel is then decorated with a “stop sign” icon (🔴). When a subsystem breakpoint is set, simulation pauses whenever an item in the subsystem (Simulink block or Stateflow transition) executes. When a Stateflow state breakpoint is set, simulation pauses whenever the state is entered or exited.

17. The main panel displays the currently selected Simulink or Stateflow diagram, C code if you are using the Reactis for C Plugin or Embedded MATLAB code if you are using the Reactis for EML Plugin. You may interact with the diagram in a number of different ways using the mouse including hovering over model items, double-clicking on items, or right-clicking in various parts of the panel. Section 4.1 describes how you interact with the main panel when Simulator is disabled. The following mouse operations are available when Simulator is enabled:

**Hovering...**
- over a data item (Simulink block or signal line, Stateflow variable, or C variable) will display its current value and type.
- over a Goto block will cause it and its associated From block(s) to be highlighted in yellow.
- over a From block will cause it and its associated Goto block to be highlighted in yellow.
- over any Tester coverage target will display the test and step within the test during which the target was first executed. This information is presented in a message of the form “Covered: test/step”. A “.” in the test location indicates the current simulation run. “-/-” indicates the target has not yet been covered. For more details on querying coverage information see Section 7.6, Chapter 10, and Chapter 6.
- over a Validator objective will cause its wiring information to be drawn in blue.
- over an output of a Logical Operator block that roots a Multi-Block Decision Coverage group will cause blue lines to be drawn to the conditions of the group (when Multi-Block Decision Coverage is enabled). See Section 6.3.1 for a description of Multi-Block Decision Coverage.

**Double-Clicking...**
- on a Scope block will open a scope window for that block (see Section 7.5.3).
- on a Display block will add that block as a watched variable (see Section 7.5.2).
- on a From or Goto block will open a dialog listing all other From or Goto blocks in the model associated with the block.
- on a Data Store Memory, Data Store Read, or Data Store Write block will open a dialog listing all matching Data Store blocks.
• on a Simulink subsystem will cause the subsystem diagram to be displayed in
  the main panel.
• on a Stateflow state will cause the state’s diagram to be displayed in the main
  panel.
• on a top-level input port while running in user-guided simulation mode will
  bring up a panel to modify that inport’s current input value.
• on a configuration variable in the Configuration Variable Panel (see Section 4.4)
  while Simulator is in the initial state (no simulation steps have been taken) will
  bring up a panel to modify the variable’s current value.
• on a line number within a C source file will toggle a breakpoint on that line.
• on any other Simulink block will display the block’s parameters.

**Right Clicking...**

Causes different pop-up menus to be displayed. The contents of the menus vary
based on where the click occurs and whether or not Simulator is enabled. A sum-
mary of the menu items available when Simulator is enabled follows. For descrip-
tions of the menu entries available when Simulator is disabled, see Section 4.1.

<table>
<thead>
<tr>
<th>Right-Click Location</th>
<th>Menu Entries (when Simulator is enabled)</th>
</tr>
</thead>
</table>
| Simulink signals, Simulink blocks, Stateflow variables | **Inspect value** Inspect the value of a bus (see Section 7.5.1).  
**Add To Watched** Add item to watched variables list (see Section 7.5.2).  
**Open Scope** Display item in scope (see Section 7.5.3).  
**Open Distribution Scope** Display item in distribution scope (see Section 7.5.4).  
**Add To Scope** Add item to previously opened scope. This item only appears when other scopes are open. |
| User defined target or assertion | **Run to Violation** If an assertion is violated, this menu entry becomes enabled. Selecting it directs Reactis to switch to the test that caused the violation and run to the step where the violation occurs.  
**View Properties** View assertion, user defined target, or virtual source properties in read-only mode. |
| Logical Operator block, Lookup Table, top-level Inport, non-else outport of If block, or Stateflow transition segment, Stateflow state | **View Coverage Details** Display dialog containing detailed coverage information for the item. (see Section 6.1.3, 6.3.1, 6.3.2, 6.2, and 7.6.2) |
| Simulink blocks | **View Block Parameters** Display Simulink block parameters. |
7.1. LABELED WINDOW ITEMS

<table>
<thead>
<tr>
<th>Right-Click Location</th>
<th>Menu Entries (when Simulator is enabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Operator block, Branching block, Lookup table, top-level Import, Stateflow transition segment, Stateflow state, or a test/step within a Coverage Details dialog</td>
<td>Track Coverage  Open a dialog from which the target can be excluded from coverage (with or without assertion), or included in coverage (see Section 6.5.3).</td>
</tr>
<tr>
<td>Top-level import or configuration variable in Configuration Variable Panel (see Section 4.4)</td>
<td>Change Value  Modify current value of top-level imports or configuration variables. Note that top-level input values may only be updated when in user-guided input mode (see Section 7.4.1) and configuration variables may only be updated in between tests (not during a test).</td>
</tr>
<tr>
<td>Top-level outport.</td>
<td>Open Difference Scope  This menu item is enabled when a test suite is loaded. The feature is used when differences exist between the value stored in the test for the outport and the value computed by the model for the outport. The resulting scope plots the expected value (from the test) against the actual value (from the model) as shown in Figure 7.13.</td>
</tr>
<tr>
<td>Top-level outport.</td>
<td>Edit Tolerance  Open the Info File Editor to view or change the output tolerance of the selected outport. Note that the tolerance cannot be changed while Simulator is enabled.</td>
</tr>
<tr>
<td>Non-virtual Simulink block or Stateflow transition</td>
<td>Toggle Breakpoint  Enable or disable breakpoint for an item.</td>
</tr>
<tr>
<td>Simulink subsystem</td>
<td>Extract Subsystem  Extract a subsystem and save it in a separate model file (see section 4.5).</td>
</tr>
</tbody>
</table>

Left-Clicking on Signals...

Causes the signal wire to be highlighted in yellow. The highlighting travels in both directions: back to its block source and forward to one or several block destinations. To make it easy to identify the relevant signal path, the subsystems the signal penetrates are highlighted as well. The signal highlighting will travel through virtual blocks such as Subsystems, Froms, Gotos, inports, outports, data-store reads and data-store writes. To continue tracing a signal through a block, click on the wire on the non-highlighted side of the block. To remove signal highlighting, left-click the mouse button in empty space.

The following five labeled window items are only available in the toolbar if you are using the Reactis for C Plugin to step through C code.

18. Step backward to the point just before the current function was called.

19. Step backward one statement. Any function calls performed by the statement are stepped over.
20. Step backward one statement. If the statement performed any function calls, stop at the end of the last function which was called.

21. Step forward one statement. Any function calls performed by the statement are stepped over.

22. Advance until the currently executing function returns.

7.2 Menus

Except for the documented exceptions related to editing .rsi files, the menus described in Section 4.2 work in the same manner when Simulator is enabled. The following additional menu items are also active when Simulator is enabled.

**View menu.** The following entries become enabled when Simulator is “on”.

*Show Watched Variables.* Toggle whether or not watched-variable list is displayed. The default is not to show them; adding to the list automatically causes the list to be displayed.

*Add Watched Variables...* Add data items (Simulink blocks or signal lines, Stateflow variables, or C variables) to the watched-variable list. Selecting this entry brings up a list of data items. You can toggle whether or not an entry in the list is selected by control-left-clicking on it; clicking OK causes the selected items to be added to the watch list.

*Clear Watched Variables.* Remove all items from the watch list.

*Open Scopes...* Open scopes for data items (Simulink blocks or signal lines, Stateflow variables, or C variables). Selecting this entry brings up a list of items. You can toggle whether or not an entry in the list is selected by control-left-clicking on the variable; clicking OK causes scopes to be opened for each selected item.

*Open Distribution Scopes...* Open distribution scopes for data items (Simulink blocks or signal lines, Stateflow variables, or C variables). Selecting this entry brings up a list of items. You can toggle whether or not an entry in the list is selected by control-left-clicking on the variable; clicking OK causes distribution scopes to be opened for each selected item.

*Close All Scopes.* Close all open scopes.

*Open Scope (Signal Group).* Open a scope for the specified signal group. A signal group is created by clicking the save button (保存) in a scope to save the current configuration of the scope as a signal group (set of signals along with the scope settings for displaying them).

*Delete Signal Group.* Delete the selected signal group.

*Save Profile as...* Save the current view profile under a new name. The view profile contains the currently opened scopes and watched variables. Profiles are saved in a file with the .rsp suffix.

*Load Profile...* Load a different view profile (.rsp file). This will automatically open all scopes and watched variables stored in the profile.

---

1 Any operation which modifies the .rsi file is disabled when Simulator is enabled.
Simulate menu. The following entries are available when Simulator is enabled. Note that some entries (Step Over, Step Out Of, Reverse Step Into, Reverse Step Over, Reverse Step Out Of) are only available when using the Reactis for C Plugin with a model containing C code.

Simulator on/off. Enable or disable Simulator. When disabled Simulator behaves as a model viewer; that is, the model can be viewed but simulation capabilities are disabled.

Fast Run with Report. Execute a fast simulation and produce a report. See Section 7.3 for details.

Fast Run. Same as window item 9.
Run. Same as window item 8.
Step. Same as window item 7.
Step Into. Same as window item 6.
Step Over. Same as window item 21.
Step Out Of. Same as window item 22.
Stop. Stop a fast or slow simulation run.
Reverse Step Into. Same as window item 20.
Reverse Step Over. Same as window item 19.
Reverse Step Out Of. Same as window item 18.
Back. Same as window item 5.
Fast Back. Same as window item 4.
Reset. Same as window item 3.

Toggle Breakpoint. Sets a breakpoint for the currently selected item in the model-hierarchy panel if none exists, or clears the breakpoint if one has already been set. Simulation will halt when the item becomes active, which may be in the middle of a simulation step. The simulator controls may then be used to continue execution of the model.

Clear Breakpoints. Removes all breakpoints.

Set Animation Delay... When running a slow simulation, this value specifies the duration of the pause between the evaluation and highlighting of different model elements.

Switch Configuration Variable Set When hovered or clicked on, a list of available configuration variable sets will be displayed, from which one set may be selected. See Section 5.4.1 for more details.

Update Configuration Variable... Initiates a dialog for changing values of configuration variables, which are workspace variables whose values can only change between tests/simulation runs (but not during a test/simulation run). The simulation must be reset to the start state (by clicking the reset button ◇, window item 3) before the value of a configuration variable may be updated. Note also that whenever inputs are read from a test, the configuration variable values from the test will be used. In other words, manual updates to a configuration variable using this menu item will only have effect when in random or user input mode.

Test Suite menu.
New. Same as window item 12.
Open. Same as window item 13.
Save. Same as window item 14.

Save and Defragment. Removing tests from a test suite can cause the test suite to become fragmented, meaning that space within the file becomes unused. Reactis will reuse those gaps when you add tests. Selecting this menu item will save the current test suite and reorganize it, removing all gaps.

Save As... Save current test suite in an .rst file. A file-selection dialog is opened to determine into which file the test suite should be saved.

Import... Import tests and add them to the current test suite. Importing is described in more detail in Section 7.7.2.

Export... Export the current test suite in different formats. Exporting is described in more detail in Section 7.7.1.

Create... Launch Reactis Tester. See Chapter 8 for details.

Update... Create a new test suite by simulating the current model using inputs from the current test suite, but recording values for outputs and test points generated by the model. This feature is described in Section 7.8.

Browse... Open a file selection dialog, and then launch the Test-Suite Browser on the selected file. See Chapter 11 for details.

Browse Current. Launch the Test-Suite Browser on the currently loaded test suite. See Chapter 11 for details.

Add/Extend Test. At any point during a simulation, the current execution sequence (from the start state to the current state) may be added as a test to the current test suite by selecting this menu item. After the test is added, it will appear in the Source-of-Inputs dialog (window item 11). Note that the new test will not be written to an .rst file until the current test suite has been saved using window item 14 or the Test Suite → Save menu item.

Remove Test. Remove the current test from the current test suite. Note that the test will not be removed from the .rst file until the current test suite has been saved using window item 14 or the Test Suite → Save menu item.

Compare Outputs. Specify whether or not Simulator should compare the simulation outputs against the outputs contained in the test suite being executed. When enabled, a warning is reported for each significant difference between the computed value and the value stored in the test suite. A difference scope may then be opened by right-clicking on a top-level outport and selecting Open Difference Scope (see Figure 7.13). The tolerance used to determine which differences are significant may be specified as described in Section 5.6.

Validate menu. See Chapter 9 for a description of this menu.

Coverage menu. The Coverage menu contains the following entries. Details about the different coverage objectives may be found in Chapter 6. The coverage information available from the various menu items is for the current simulation run. If a test suite is being executed, the coverage data is cumulative. That means all targets covered by the portion of the current test executed so far, plus those targets exercised in previous tests are listed as covered.
7.3. CREATING TEST EXECUTION REPORTS

Show Summary. Open the coverage summary dialog shown in Figure 7.15.

Show Details. Report coverage information by coloring diagram elements as defined in the Line Style dialog shown in Figure 7.24. Generally, uncovered targets are drawn in red.


Show Quick HTML Report Shows a HTML coverage report. This is the same as selecting Coverage → Show Report..., selecting Report → Export... within the report and then clicking the “Preview” button. The window for this report stays open but does not get updated for changes in coverage when working with Simulator. To get an updated report after taking steps in Simulator, re-execute the Show Quick HTML Report item.

Highlight Subsystems, Branches, Lookup Targets, States, CSEPT, Condition Actions, Transition Actions, Decisions, Conditions, MC/DC, MCC, Boundaries, User-Defined Targets, Assertion Violations, C Statements. Each of these menu entries corresponds to one of the model coverage metrics tracked by Reactis and described in Chapter 6. When a menu entry is selected and Show Details is selected, any uncovered target in the corresponding coverage metric will be colored.

Select All. When Show Details is selected, show coverage information for all metrics.

Deselect All. When Show Details is selected, show no coverage information.

Highlight Unreachable Targets. When Show Details is selected, color unreachable targets. A target is unreachable if it can be determined (without executing the model) that the target will never be covered regardless of the input values used during testing. The analysis used is conservative: marked items are always unreachable, but some unmarked items may also be unreachable.

7.3 Creating Test Execution Reports

Fast Simulation Run with Report executes all tests within the current test suite and produces a report which lists all runtime errors (divide-by-zero, overflow, memory errors, missing cases, assertion violations, etc.) and significant differences between outport values stored in the test suite and those computed by the model.

When Simulate → Fast Run with Report... is selected, the dialog shown in Figure 7.2 will appear. This dialog is used to select the items which will appear in the report. The following items are labeled in Figure 7.2:

1. The Report Options panel is used to select optional report items. These include the date, pathnames of input files, coverage information, and plots of test data. If file paths are included in the report, the model version will also be included within parentheses following the model name. By default, the version number is the return value after calling the MATLAB function:

   get_param('modelName','ModelVersion');

   where modelName is the name of your model. If you wish to include a different version number, you can redefine the workspace variable reactis_model_version in the Reactis
CHAPTER 7. REACTIS SIMULATOR

Post-Load Function. For example, when your model is loaded, select Edit → Callbacks..., then in the Post-Load entry box enter:

```c
reactis_model_version = my_function_to_construct_version();
```

If Include step-to-covered-targets map is selected, then the generated report will include a table similar to that shown below for each test. The table lists each step in the test that covers some target along with the targets covered by the step.
2. When Include coverage report is selected in the Report Options panel, the Coverage Metrics panel is used to select which coverage metrics are included in the test execution report. There are three choices for each metric:

- **Summary & Details.** Targets of the metric will appear in both the coverage summary and coverage details sections of the report.
- **Summary Only.** Targets of the metric will appear in the coverage summary only.
- **None.** Targets of the metric will be omitted from the report entirely.

Note that due to dependencies between metrics, some combinations are not allowed. For example, **Summary & Details** cannot be selected for Condition targets unless **Summary & Details** is also selected for Decision targets.

3. The **Output Format** panel is used to select the format of the exported report (HTML or RRX/XML). If **Preview before saving** is checked, the exported report will be displayed and you will have the option of saving or discarding it.

4. The **Output** panel is used to specify where the report will be stored. A test execution report can be stored in a single file, or it can be stored in multiple files (one file per test).

5. When you are satisfied with the selected report options, clicking on this button will close the dialog and start the simulation run.

6. Clicking on this button will close the dialog without initiating a simulation run.

Once the simulation run begins, it does not stop until all tests have been executed. During each test, if a runtime error is encountered, the remaining steps of the test are skipped and Simulator continues execution with the following test. After the last test is executed, a window containing the test execution report will appear, as shown in Figure 7.3. An HTML version of the report can be saved by clicking the **Save** button in the report window.

An HTML test execution report will contain some or all of the following sections, depending on which options are selected:

1. A **report header** listing the data, input files, Reactis version, etc.

2. A **test summary** listing the tests/steps which were executed and the number of errors and differences detected for each test. Non-zero error and difference totals can be clicked-on to jump to a detailed description of the error or difference.

3. The **tolerance** used to test the value of each output and test point.
4. A list of test details. For each test, includes the details for each error and difference that occurred, and plots of test data. The plots for a test are hidden by default, but they can be viewed by either clicking on the ± to the left of the signal name, or by clicking on Open all. See section 7.3.1 for details.

5. The model hierarchy. The name of each member of the hierarchy can be clicked on to jump to the coverage details for that member.

6. Coverage details for each component of the model. The coverage details for a model
component begin with a summary of the local and cumulative coverage, followed by
details for each metric. The details for a metric show, for each target, whether or not the
target was covered, and if the target was covered, the test step when coverage occurred.
The contents of this section are identical to a coverage report (Section 10.4).

7.3.1 Test Data Plots

Figure 7.4: An output plot from a test execution report.

Figure 7.4 shows a typical plot from a test execution report. Test data (inputs, outputs,
or test points) are plotted with the simulation time on the x-axis and the data value(s) on
the y-axis. For outputs and test points, two values are shown: the test value (green), and
the computed value (blue). The test value is the value stored in the test being executed for
the output. The computed value is the value computed by the model for the output while
executing the test. The acceptable tolerance between the two values is shaded yellow. Regions
where the difference between the two signals is larger than the tolerance are shaded red.

Plots can be inspected when viewed from within a web browser or the preview dialog.
The current focus of inspection is indicated by the intersection of two gray dashed lines. The
focus can be moved either by the mouse, or the left and right arrow keys on the keyboard.
Pressing S will move the focus to the start of the plot, and pressing E will move the focus to
the end.

There are six values \(^2\) displayed at the top of the plot for the current focus. These are (1) the
step number, (2) the simulation time, (3) the test value (y value of green line), (4) the computed
value (y value of blue line), (5) the difference between the test value and the computed value,
and (6) the maximum difference between the test and computed values which is tolerated.
These six values are updated whenever the focus is moved.

7.4 Specifying the Simulation Input Mode

Reactis Simulator performs simulations in a step-by-step manner: at each simulation step
inputs are generated for each top-level inport, and resultant outputs reported on each top-
level outport. You can control how Simulator computes input values using the Source-of-
Inputs dialog (window item 11 in Figure 7.1) shown in Figure 7.5. This dialog always includes
the Random Simulation and User Guided Simulation entries; if a test suite has been loaded,
then the dialog also includes an entry for each test and the All button becomes enabled. The
dialog is used to specify how input values are generated as follows.

Random Simulation. For each inport, Reactis randomly selects a value from the set of al-
lowed values for the inport, using type and probability information contained in the

\(^2\)only the first three are shown for inputs
CHAPTER 7. REACTIS SIMULATOR

Figure 7.5: The Source-of-Inputs dialog enables you to specify how Simulator computes input values.

associated .rsi file. See Chapter 5 for a description of how to enter this information using the Reactis Info File Editor.

User Guided Simulation. You determine the value for each inport using the Next Input Values dialog, which appears when the User Guided Simulation entry is selected. See Section 7.4.1 below for more information on this mode.

Individual Tests. When a test suite is loaded, each test in the suite has a row in the dialog that contains a test number, a sequence number, a name and the number of steps in the test. Selecting a test and clicking OK will cause inputs to be read from the test.

Subset of Tests. You may specify that a subset of tests should be run by holding down the control key and clicking on each test to be run with the left mouse button. You can also hold the shift key while clicking to select a block of tests to be executed. The tests will be run in the order they are selected. As tests are selected, the sequence number column is updated to indicate the execution order of the tests. When a new test is started, the model is reset to its starting configuration, although coverage information is not reset, thereby allowing you to view cumulative coverage information for the subset of tests.

All Tests. Clicking the All button in the lower left corner specifies that all tests in the suite should be executed one after another. The tests are executed sequentially. When a new test is started, the model is reset to its starting configuration, although coverage infor-
7.4. SPECIFYING THE SIMULATION INPUT MODE

You can change the sorting order of the tests in the table by clicking on the column headers. For example, to sort the tests by the number of steps, simply click on the header of the “Steps” column. Clicking again on that header will sort by number of steps in descending order.

You may also use the Inputs Source dialog to change the name of a test. To do so, select the test by clicking on it, then click on the name and, when the cursor appears, type in a new name.

### 7.4.1 User Input Mode

When the User Guided Simulation mode is selected from the Source-of-Inputs dialog, you provide values for inports at each execution step. This section describes how this is done.

To enter the user-guided mode of operation, select UserGuidedSimulation from the Source-of-Inputs dialog (window item 11 in Figure 7.1). Upon selecting user-guided mode, a Next Input Values dialog appears, as shown in Figure 7.6, that allows you to specify the input values for the next simulation step. Initially, each top-level inport of the model has a row in the dialog. You can remove inputs from the dialog or add outputs, test points, and configuration variables by clicking the gear button (⚙️) in the toolbar of the Next Input Values dialog. Each row in the dialog contains 6 items (labeled 1-6 in Figure 7.6). The toolbar for the dialog contains items 7-13. The elements of the dialog work as follows:

1. The name of an item (inport, outport, test point, or configuration variable).
2. This checkbox toggles whether the item is included in a scope displaying a subset of the signals from the Next Input Values dialog.
3. This pull-down menu has two entries that determine how the next value for the port is specified:

- **Random** Randomly select the next value for the inport from the type given for the inport in the `.rsi` file.
- **Entry** Specify the next value with the text-entry box in column four of the panel.
- **Min** Use the minimum value allowed by the inport's type constraint.
- **Max** Use the maximum value allowed by the inport's type constraint.
- **Test** Read data from an existing test suite (see Section 7.4.1.4).

4. If the pull-down menu in column three is set to “Entry”, then the next input value is taken from this text-entry box. The entry can be a concrete value (e.g. integer or floating point constant) or a simple expression that is evaluated to compute the next value. These expressions can reference the previous values of inputs or the simulation time. For example, a ramp for input drag can be specified by `pre(drag) + 0.0001`. A sine wave can be generated by `sin(t) * 0.001`. For the full description of the expression notation see Section 7.4.1.1 below.

5. If the pull-down menu in column three is set to “Entry”, then clicking the history button (labeled H) displays recent values the inport has assumed. Selecting a value from the list causes it to be placed in the text-entry box of column four.

6. The arrow buttons in this column enable scrolling through the possible values for the port. The arrows are available for inports or configuration variables:
   - having a base type of integer, boolean or fixed point; or
   - having a base type of `double` or `single` and either a resolution or subset of values constraint.

7. When you enter a search string in this box, Reactis displays only the rows for items whose names contain the given search string.

8. When you check this box, all signals in the Next Input Values dialog are plotted in a scope. When you uncheck this box, all signals are removed from the scope and no scope is displayed.

9. This pull-down menu sets the input mode for all ports at once to either “Random”, “Entry”, “Min”, “Max” or “Test.”

10. These buttons control the simulation stepping exactly as they do in the top-level main Simulator window.

11. The entry in this box is a positive integer which specifies how many steps to take when clicking one of the stepping buttons that triggers multiple steps (e.g. fast simulation button).

12. Open a dialog to select the set of signals (inputs, outputs, test points, configuration variables) to be included in the Next Input Values dialog.

13. Save the current configuration of the Next Input Values dialog for future use or load a previously saved saved configuration.
When “run fast simulation” (window item 9 in Figure 7.1) is selected, the input value specifications in the Next Inputs Values dialog are used for each step in the simulation run.

7.4.1.1 Syntax of Next Input Value Expressions

The value an input should assume in the next simulation step can be specified from its row by selecting Entry in column 3 and then entering an expression in the box in column 4. We now describe the language used to define an expression.

Assume $foo$ is an input. Then the following examples demonstrate some possible expressions to specify the next value of $foo$.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value $foo$ will have in next step</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.0</td>
<td>4.0</td>
</tr>
<tr>
<td>$pre(foo)$</td>
<td>The value $foo$ had in the previous step</td>
</tr>
<tr>
<td>$pre(foo,2)$</td>
<td>The value $foo$ had two steps back</td>
</tr>
<tr>
<td>$pre(foo) + 1$</td>
<td>Add 1 to the value of $foo$ in the previous step</td>
</tr>
<tr>
<td>$pre(u)$</td>
<td>Shorthand denoting the value of $foo$ in the previous step</td>
</tr>
<tr>
<td>$t$</td>
<td>The current simulation time</td>
</tr>
<tr>
<td>$\sin(t)$</td>
<td>The sine of current simulation time (i.e. generate a sine wave)</td>
</tr>
</tbody>
</table>

The complete syntax of a next input value expression $NIV$ is specified by the following grammar.
7.4.1.2 Reading Data from Existing Test Suites in Expressions

Within the expression for specifying the next input value for a port the testval_step and testval_time functions can be used to read values from an existing test suite.
7.4. SPECIFYING THE SIMULATION INPUT MODE

\textbf{testval} \_ \textit{step} (suite \textit{filename}, \textit{portname}, testnum, stepnum)  \textit{\textup{Reads a value from a test suite, specified by the step number within the test suite.}}

\textbf{testsuitefilename}  \textit{\textup{The file name of the test suite from which data should be read. If a relative path is used then it is relative to the directory in which the current model resides. Must be enclosed in double-quotes.}}

\textbf{portname}  \textit{\textup{The name of the port from which data should be read. This can be an input port, test point or output port, as they appear in the Reactis Test Suite browser. You can use the function "port()" here to use the same name as the port whose input you are specifying. The name must be enclosed in quotes.}}

\textbf{testnum}  \textit{\textup{The number of the test in the test suite from which data should be read. This number is 1-based, i.e. first step is 1.}}

\textbf{stepnum}  \textit{\textup{The step number within the test. Use the "step()" function to refer to the current simulation step. You can use arithmetic expressions to adjust the step number, for example to add an offset.}}

Example: \textit{testval} \_ \textit{step} ("cruise.rst", "onOff", 3, step() + 5) \textit{\textup{will read data from the 3rd test of test suite "cruise.rst", port "onOff", offset by 5 steps, i.e. the first step read from the test suite will be step 6.}}

\textbf{testval} \_ \textit{time} (suite \textit{filename}, \textit{portname}, testnum, time, interpolate)  \textit{\textup{Reads a value from a test suite, specified by the simulation time in the test suite.}}

\textbf{testsuitefilename}  \textit{\textup{The file name of the test suite from which data should be read. If a relative path is used then it is relative to the directory in which the current model resides. Must be enclosed in double-quotes.}}

\textbf{portname}  \textit{\textup{The name of the port from which data should be read. This can be an input port, test point or output port, as they appear in the Reactis Test Suite browser. You can use the function "port()" here to use the same name as the port whose input you are specifying. The name must be enclosed in quotes.}}

\textbf{testnum}  \textit{\textup{The number of the test in the test suite from which data should be read. This number is 1-based, i.e. first step is 1.}}

\textbf{time}  \textit{\textup{The simulation time within the test. Use "t" to refer to the current simulation model time. You can use arithmetic expressions to adjust the time value, for example to add an offset or scaling.}}

\textbf{interpolate}  \textit{\textup{If the time value specified falls between the time for two steps in the test suite then this parameter defines how the value is computed. If the interpolate parameter is 0 then the value from the test suite corresponding to the step before the time value is used. If interpolate is 1 then the value is interpolated (linear) between the steps before and after the simulation time.}}

Example: \textit{testval} \_ \textit{time} ("cruise.rst", port(), 4, \textit{t}/10, 1) \textit{\textup{will read data from the 4th test of test suite "cruise.rst" matching the current port name. Time will be slowed down by factor 10 and values will be interpolated between steps.}}
7.4.1.3 Select Signals Dialog

The Select Signals dialog lets you choose which signals to include in the Next Input Values dialog during user-guided simulation. Initially all top-level inputs are included, but you can remove inputs or add outputs, test-points, and configuration variables by clicking the button in the Next Input Values dialog and then using the Select Signals dialog to specify the desired subset of signals. Note that in the case of outputs and test points the signal values are only observed, not controlled.

![Select Signals dialog](image)

The labeled items in Figure 7.7 work as follows. Note that since outputs and test points are only observed and not controlled, those tabs include only a subset of the columns described below.

1. This tab lets you select top-level inputs.
2. This tab lets you select top-level outputs.
3. This tab lets you select test points.
4. This tab lets you select configuration variables.
5. Port number.
7. Toggles whether or not the item is included in the Next Input Values dialog.
8. Generate a random next input value.
9. Generate the smallest allowed next input value.
10. Generate the largest allowed next input value.
7.4. SPECIFYING THE SIMULATION INPUT MODE

11. Reads a value from an existing test suite (see Section 7.4.1.4).

12. Use an expression to specify the next input value (item 12).

13. An expression used to generate the next input value.

Note that if item 7 is checked for an input, then the settings specified by items 8-10 will determine the initial configuration of the Next Input Values when it is first opened. The settings can subsequently be modified while stepping. If window item 7 is not checked for an input, then the settings specified by items 8-10 will be the only ones used during stepping.

7.4.1.4 Reading Data from Existing Test Suites

In user-guided simulation mode you can set up any number of inputs to read data from one or more existing test suites:

- To set all inputs to receive data from the same test suite, either select the “Test” entry in the “all ports” drop-down box in the Next Input Values window (item 9 in Figure 7.6) or click on the header of the “Test” column in the Select Signals dialog (item 11 in Figure 7.7).

- To set a single input to receive data from a test suite, either select the “Test” entry in the drop-down box for the desired input in the Next Input Values window (item 3 in Figure 7.6) or click the radio button in the “Test” column for the desired input in the Select Signals dialog (item 11 in Figure 7.7).

- To set a range of inputs to receive data from the same test suite, open the Select Signals dialog (Figure 7.7) and click in the “Test” column of the first input to be set to a test. Make your selections in the test selection dialog (see below) and click OK. Then hold down the shift key and click the last input to be set to a test, which will again bring up the test selection dialog, pre-configured with your previous selection. Click the OK button again.

Doing any of the above will bring up the dialog shown in Figure 7.8, the labeled items work as follows.

1. Test data will be read from the test suite currently loaded in Simulator. Note that the test suite must be saved before it can be selected here.

2. Test data will be read from the test suite selected here.

3. Selects the test from which data will be read.

4. Data item (input port, output port, configuration variable or test point) in the test suite that the data will be read from. If set to “[Current]” then data will be read from the port in the test suite whose that matches the current port name. This choice is disabled when selecting test data for multiple inputs.

5. If this mode is selected then data will be read from the test suite on a step-by-step basis, disregarding simulation time. This mode is more efficient and avoids possible rounding errors if the sample rate of the model and the data in the test suite are identical. If test data is read for steps past the end of the test data, the last step value will be repeated.
6. This allows to set an offset to the step number in the test suite. For example, if set to 5 then the first step read from the test suite will be step number 6. If this is a negative value then the data read before step 1 will be the same as the data for step 1.

7. If this mode is selected then data will be read from the test suite based on the current simulation time. This is useful if either the sample rate of the test data does not match the model’s sample rate or to perform time scaling (see below).

8. If this is checked and the time from which data is to be read from the test falls between two time steps in the test suite then linear interpolation will be used to calculate the test data value. If this is not checked then the data from the previous step in the test suite will be used. This also influences the behavior if test data is read for time less than zero or past the end of the test. If checked then the two first or last steps in the test data are used to extrapolate the test data value for the given time. Otherwise the test data for the first or last step is used.

9. Provide a time offset when reading data from the test suite. This value will be added to the current simulation time to determine the time for which data is read from the test suite.

10. Provide a time scaling when reading data from the test suite. The current simulation time will be multiplied by this value (before adding the offset) to determine the time for which data is read from the test suite. This allows to speed up or slow down time. For example, if set to 0.5 then data from the test suite will be read at half the speed at which the model simulation time progresses.
Note that this dialog is meant to provide a more user-friendly way to specify the parameters to the testval_step() and testval_time() functions described above in Section 7.4.1.2. Using the functions directly will provide more flexibility in how to read test data. If you set up a port to read data from a test using this dialog and subsequently switch the test data selector (item 3 in Figure 7.6) to “Entry” then you will see the function expression that was constructed to reflect your dialog choices.

7.4.2 Test Input Mode

Simulation inputs may also be drawn from tests in a Reactis test suite. Such tests may be generated automatically by Reactis Tester, constructed manually in Reactis Simulator, or imported using a comma separated value file format. By convention files storing Reactis test suites have names suffixed by .rst.

A Reactis test suite may be loaded into Simulator by clicking the in the tool bar to the right of Source-of-Inputs dialog (window item 13 in Figure 7.1) or by selecting the Test Suite → Open... menu item.

When a test suite is loaded, the name of the test suite appears in the Reactis title bar and the tests of the suite are listed in the Source-of-Inputs dialog.

When executing in test input mode while Test Suite → Compare Outputs is selected, Simulator compares the values computed by the model for test points and top-level output ports against the values stored in the test suite for those items. These comparisons are performed after each simulation step. A difference is flagged if it exceeds the tolerance specified for that port. See Section 5.6 for more information on specifying tolerances for output ports.

If a value in the test differs from that computed by the model for a top-level outport, the difference may be observed (as shown in Figure 7.13) by right-clicking on the outport and selecting Open Difference Scope.

7.5 Tracking Data-Item Values

Reactis Simulator includes several facilities for interactively displaying the values that data items (Simulink blocks or signal lines, Stateflow variables, or C variables) assume during simulation. The watched-variable list, or “watch list” for short, displays the current values of data items designated by the user as “watched variables.” You may also attach scopes to data items in order to display values as they vary over time. Scopes behave like Simulink Scope blocks except that they are not hard-wired into models and are instead opened and closed during simulation. Distribution scopes enable you to view the set of values a data item has assumed during simulation (but not the time at which they occur). Difference scopes may be opened for top-level outports when reading inputs from a test in order to plot the values computed by the model against the values stored in the test for the outport.

You may add data items to the watch list, or attach scopes to them, as follows.

Using the View menu. The View menu contains operations for adding data items to the watch list, opening scopes, and opening distribution scopes. These are described in more detail in Section 7.2.

Using pop-up menus in the model hierarchy panel. Right-clicking on a subsystem in the hierarchy panel brings up a pop-up menu that includes the entries:
150

CHAPTER 7. REACTIS SIMULATOR

- Add Watched Variables...
- Open Scopes...
- Open Distribution Scopes...

Selecting one of these entries will cause a dialog to appear listing data items in the subsystem which may be added to the watched variable list or to which scopes may be attached.

**Using pop-up menus in the main panel.** Right-clicking on a data item in the main panel of Simulator invokes a menu that enables you to add the data item to the list of watched variables or open a scope or distribution scope to monitor the values of the data item during simulation. This menu also includes an entry *Add To Scope* that enables you to plot the data item on a previously opened scope.

You may save the current configuration of the data tracking facilities (the variables in the watch list and currently open scopes along with their locations) for use in a future Simulator session. Do so by selecting *View → Save Profile As...* and using the resulting file selection dialog to specify a file in which to save a *Reactis profile* (.rsp file). The profile may be loaded at a future time by selecting *View → Load Profile*.

### 7.5.1 Selecting Bus Components

![Open Scope](image)

Figure 7.9: Opening a scope on a bus component.

When any of the standard value-inspection operations (*Inspect value..., Add Watched Variables..., Open Scopes..., Open Distribution Scopes...*) are applied to a bus, an intermediate menu appears which allows you to select a bus component of interest. An example of this is shown in Figure 7.9, where the second element of bus member S2 is about to be selected.
7.5. TRACKING DATA-ITEM VALUES

The watch list is displayed in a panel at the bottom of the Simulator screen as shown in Figure 7.10. By default this panel is hidden, although adding a variable to the watch list causes the panel to become visible. Visibility of the panel may also be toggled using the View menu as described in Section 7.2. The panel displays a list of data items and their values. The values are updated when Simulator pauses.

The contents of the watch list may be edited using a pop-up menu that is activated from inside the watch-list panel. Individual data items in the panel may be selected by left-clicking on them. Once an item is selected, right-clicking invokes a pop-up menu that enables the selected item(s) to be deleted, have a scope opened, or have a distribution scope opened. If no item is selected, then these choices are grayed out. The right-click pop-up menu also includes an entry Add Variables which displays a list of all data items in the model which may be added to the watch list.

The View menu contains operations for displaying / hiding the watch list, adding data items to the watch list, clearing the watch list.

Right-clicking on a variable in the list brings up a menu containing the following options:

- **Add Variables**... Same as View → Add Watched Variables...
- **Remove Variable** Remove the currently selected variable from the list.
- **Set Significant Digits** Specify the number of significant digits to display for the currently selected variable in the list. Specifying -1 will reset to the default number of digits.
- **Inspect Value**...
- **Add Component to Watched**... If the currently selected variable in the list is a vector, matrix or bus, selecting this menu item will bring up a selection dialog (see Section 7.5.1) that
allows you to add a sub-element of the current selection as a watched variable.

**Show Location** Highlights the location of the currently selected watched variable in the main panel.

**Open Scope** Opens a scope for the currently selected variable.

**Open Distribution Scope** Opens a distribution scope for the currently selected variable.

**Highlight updated entries** By default, Reactis highlights variables in the watched variable list if their values have changed. This sub-menu allows you to modify the highlighting behavior:

- **Do not highlight** Never highlight any variables in the list
- **Highlight when updated** Always highlight a variable when it is assigned a new value, even if the new value is the same as the previous value.
- **Highlight when updated with different value** Highlight a variable if it is assigned a value different from its previous value. This is the default.

### 7.5.3 Scopes

Scopes appear in separate windows, an example of which may be found in Figure 7.11. The tool bar of each scope window contains nine or more items.

![Figure 7.11: A scope window plotting desired speed (green) and actual speed (yellow).](image)

### 7.5.3.1 Labeled Window Items

1. Reset the zoom level of the scope to fit the whole plot (see more on zooming below).

2. Plot signal as solid lines.
3. Plot signal as points.

4. If a scope displays multiple signals, this button toggles whether or not all signals share the same y-axis or each is plotted on its own y-axis.

5. Save the current scope configuration as a signal group. A signal group is a set of signals along with the settings for displaying the signals in a scope. After saving a signal group, you can reopen a scope for the group in future Reactis sessions. You can add additional signals to a signal group by right-clicking on a signal in the main Reactis panel (when Simulator is enabled), selecting Add to Scope, and selecting the signal group to be extended. To reorder the signals in a group or remove a signal, open a scope for the signal group then click the Settings button (item 8).

6. Export scope data as either text (csv) or graphics (png, gif, bmp, tif or jpg).

7. Copy a screen shot of the scope to the clipboard.

8. Configure the scope settings, including reordering the signals or deleting a signal.


10. Toggle display of signal 1.

11. Toggle display of signal 2.

To zoom in to a region of interest of the signal shown in the scope, left-click the top-left corner of the region, hold the button and drag to the lower right corner of the region. The scope will zoom in to the selected region. To zoom out, either click the zoom-to-fit button in the toolbar or right-click in the scope window. Right-clicking will return to the zoom level that was active before the last zoom.

When zoomed in, it is possible to move the displayed region within the scope window. To move the region, hold down the CTRL key and click-and-drag with the left mouse button.
If more than one data item is plotted on a scope, then a toggle button will appear in the tool bar for each data item (window items 10 and 11 in Figure 7.11). Turning one of these buttons off will hide the corresponding data item in the scope. Hovering over the button will display the data item to which the button corresponds.

### 7.5.4 Distribution Scopes

Distribution scopes also appear in separate windows, an example of which may be found in Figure 7.12. The values a data item assumes are displayed as data points distributed across the X-axis. Zooming in distribution scopes works the same as in regular scopes.

![Distribution scope example](image)

Figure 7.12: Distribution scopes plot the values a data item has assumed during simulation.

### 7.5.5 Difference Scopes

When executing tests from a test suite, a difference scope may be opened by right-clicking on a top-level outport and selecting *Open Difference Scope*. The resulting scope plots the expected value (from the test) against the actual value (from the model) as shown in Figure 7.13. If the difference between the two values exceeds the tolerance specified for the output port (see Section 5.6), then a red background in the difference scope and a red bar on the X-axis highlight the difference.

After zooming into an area of difference, white and yellow and green background regions around the plotted values highlight the tolerance, as shown in Figure 7.14. The green region represents the overlap between the tolerance of the test and model values. A difference is flagged whenever the test or model value lie outside of the green region.

### 7.6 Tracking Model Coverage

Chapter 6 describes the coverage metric that Reactis employs for measuring how many of a given class of syntactic constructs or *coverage targets* that appear in a model have been executed at least once. Simulator includes extensive support for viewing this coverage information about the parts of the model that have been exercised by the current simulation run. If a test suite is being executed, the coverage data is cumulative. That is, all targets covered by the portion of the current test executed so far, plus those targets exercised in previous tests are listed as covered.
Figure 7.13: A difference scope may be opened by right-clicking on a top-level outport and selecting Open Difference Scope. The scope plots the values stored in a test for an output and the values computed by the model for the output. Differences are flagged in red.

Figure 7.14: The white and yellow colored backgrounds around the value lines highlight the tolerance intervals of the test and model values. The overlap between the yellow and white regions is colored green. If either the test or model value lie outside the green region, a difference is flagged.
7.6.1 The Coverage Summary Dialog

The Coverage Summary Dialog shown in Figure 7.15 may be invoked at any time Simulator is enabled by selecting Coverage → Show Summary. The dialog reports summary statistics for each coverage metric tracked by Reactis. Each row in the dialog corresponds to one of the metric and includes five columns described below from left to right.

1. The name of the coverage metric reported in the row.
2. The number of targets in the metric that have been exercised at least once.
3. The number of targets in the metric that are unreachable. A conservative analysis is performed to check for unreachable targets. Any target listed as unreachable is provably unreachable; however, some unreachable targets might not be flagged as unreachable.
4. The number of reachable targets in the metric that have not been exercised.
5. The percentage of reachable targets in the metric that have been exercised at least once.

![Coverage Summary Dialog](image)

**Figure 7.15: The Coverage Summary Dialog**

7.6.2 Coverage Information in the Main Panel

Selecting Coverage → Show Details causes uncovered targets to be drawn in red in the main panel. Targets that have been covered are drawn in black. Hovering over an exercised target will cause a pop-up to be displayed that gives the test and step in which the target was first executed. This type of test and step coverage information is displayed with a message of the form test/step. A “.” appearing in the test position ./step denotes the current simulation
7.6. TRACKING MODEL COVERAGE

run which has not yet been added to a test suite. If $-/-$ is displayed, the target has not yet been covered.

![Figure 7.16: Viewing MC/DC-related coverage information for a Stateflow transition.](image)

For items included in the MC/DC coverage measure (Simulink Logic and If blocks and Stateflow transition segments whose label includes an event and/or condition) or CSEPT coverage (states, transition segments), detailed coverage information may be obtained by right-clicking on the item and selecting View Coverage Details. A dialog similar to that in Figure 7.16 will appear and give coverage information for decision coverage, condition coverage, and MC/DC.

The table in this figure describes coverage for the decision:

$$set == 1 \land deactivate == 0$$

This decision contains the following two conditions:

- $set == 1$
- $deactivate == 0$

Conditions are the atomic boolean expressions that are used in decisions. The first two columns of the table list the test/step information for when the decision first evaluated to true and when it first evaluated to false. A value $-/-$ indicates that a target has not yet been exercised. The third column lists the conditions that make up the decision, while the forth and fifth columns give test/step information for when each condition was evaluated to true and the false.

MC/DC Coverage requires that each condition independently affect the outcome of the decision in which it resides. When a condition has satisfied the MC/DC metric in a set of tests, the sixth and seventh columns of the table explain how. Each element of these two
columns has the form $bb:\text{test}/\text{step}$, where each $b$ reports the outcome of evaluating one of the conditions in the decision during the test and step specified. Each $b$ is either $T$ to indicate the condition evaluated to true, $F$ to indicate the condition evaluated to false, or $x$ to mean the condition was not evaluated due to short-circuiting.

In addition to MC/DC, Reactis can also measure Modified Condition Coverage, or MCC. MCC targets every possible combination of conditions within a decision, so that a decision containing $N$ conditions can result in as many as $2^N$ MCC targets, although the actual number may be less if short-circuited evaluation is used.

Figure 7.17 shows the MCC coverage details for a Stateflow transition. The MCC coverage details are in the form of a table where all except the last 2 columns correspond to conditions, and each row corresponds to a single MCC target. For each target, all conditions have one of three possible values:

- **True.** The condition is true.
- **False.** The condition is false.
- **x.** The condition is not evaluated due to short-circuiting.

The last two columns of the MCC details contain the result of the decision and the test/step when the target was covered.

As shown in Figure 7.18, MCC coverage details can be filtered by clicking on the column headers. A filtered column header is indicated by a prefix of $T:\text{ filters}$, $F:\text{ filters}$, or $x:\text{ filters}$, which correspond to the column values True, False, and $x$, respectively. Clicking on a column header advances the
7.6. TRACKING MODEL COVERAGE

Figure 7.18: Filtering MCC coverage information for a Stateflow transition.

filter setting for that column to the next feasible value, eventually cycling back to unfiltered. The Covered column includes one additional prefix E: to display the targets that have been excluded (see Section 6.5.3). All columns can also be reset to the unfiltered state at any time by clicking on the Clear Filter button. Note that the individual filters for each column are combined exclusively (i.e., using the Boolean and operator), so that only targets which satisfy all active filters are shown. Figure 7.18 (c) shows the results of setting the filters for set==1.0 and deactivate==0.0 to true. In this case, the only target shown is the one where both of these conditions are true.

Figure 7.19 shows the Coverage Details dialog for Child State Exit via Parent Transition (CSEPT) coverage. For the full definition of this metric see Section 6.2.4. Conceptually CSEPT tracks whether each child of a Stateflow state S has been exited by each transition that causes S to exit. In the figure, CSEPT tracks that each of the states Inactive, Active, and Init have been exited as a result of the transition from On to Off firing.

Right-clicking on a (non-top-level) state S and selecting View Coverage Details causes the display of a dialog that lists the transitions that cause the parent of S to exit. Clicking the Highlight buttons in the dialog lets you identify the state and transitions from the list in the main panel.

Right-clicking on a transition segment that is part of a transition T that causes a parent state to exit and selecting View Coverage Details causes the display of a dialog that lists the child states that can be exited as a result of T firing. Clicking the Highlight buttons causes each transition to be highlighted in the main panel.

7.6.3 The Coverage Report Browser

The Coverage-Report Browser enables you to view detailed coverage information and export the reports in HTML format. It is invoked by selecting Coverage → Show Report and is described in detail in Chapter 10.
7.7 Exporting and Importing Test Suites

7.7.1 Exporting Test Suites

The export feature of Reactis allows you to save .rst files in different formats so that they may be processed easily by other tools. The feature is launched by selecting Test Suite → Export... when a test suite is loaded in Simulator. You specify the format and name of the exported file in the General tab of the Export Dialog (Figure 7.20). For some export formats, other tabs appear in the dialog to enable you to fine-tune exactly what is included in the exported file. In the case of .csv files, you may specify a subset of tests from the test suite to be exported as well as which data items (inputs, outputs, test points, configuration variables) should be included in each test step. The following formats are currently supported:

.m files: Suites may be saved as MATLAB scripts so that they may be run using The MathWorks’ Simulink / Stateflow environment. Section 12.2 describes how to execute exported .m files in Simulink. That section describes how the rsRunTests utility distributed with Reactis enables you to load an exported .m file, execute the tests therein, and report any differences between the values computed by Simulink for outputs and the values stored in the tests. When exporting to this format you have a choice of exporting fixed-point values on top-level input or output ports as either “double” values (easier to read) or Simulink fixedpoint objects (more precise). Enumerated values can be exported as either their underlying integer values or instances of their enumeration objects.
.mat files: Suites may be saved as .mat files so that they may be run using The MathWorks’ Simulink / Stateflow environment. This binary format enables values in tests to be represented with more precision than is possible in the ASCII-based .m file format. When running a Reactis-generated test suite on a Simulink model, the higher precision of test data helps avoid some rounding errors. Section 12.2 describes how to execute exported .mat files in Simulink. The rsRunTests utility works for .mat files exactly as described above for .m files. When exporting to this format, fixpoint values will always be exported as “double” values and enumerated values will be exported as their underlying integers.

.mat files (for FromWorkspace blocks): Suites may be saved in an alternative .mat file format so that they may be run using The MathWorks’ Simulink / Stateflow environment on a modified version of the model that uses ‘FromWorkspace’ blocks in place of top-level inports. Section 12.2 describes the contents of these exported files and how to execute them in Simulink.

.csv files: Suites may be saved as comma separated value (CSV) files. The different tabs of the export dialog enable you to specify which data from a test suite should be exported. Namely, you can indicate which tests should be exported and for each test step which inputs, outputs, test points, and configuration variables should have values recorded.

The first line of an exported file will contain a comma separated list of the names of the model’s input and output ports, test points, and configuration variables that were
selected for export. A column recording the simulation time has the label \_\_t\_\_\_. Any names containing non-alphanumeric characters will be surrounded by double quotes ("), and newlines in names will be translated to \n. Subsequent lines contain either:

• A comma-separated list of values that includes one value for each item appearing in the first row. The order of the values in a row corresponds to the order the items appeared in the first line. Each such line contains the values for one simulation step.
• An empty line signaling the end of a test.

The options for configuring the CSV output work as follows:

**Compress output.** If selected, then test steps will be omitted if no item that would be recorded in the step other than the simulation time is different from the corresponding value in the previously recorded step. This is especially useful when exporting only inport data for a test in which inputs are held constant for a number of steps.

**Export vector, matrix and bus elements in separate columns.** If checked then each of these items will be placed in its own column. If not checked, when a port carries a vector or bus signal, then the values of the signal appear within double quotes (") as a comma-separated list.

**Prepend ‘|’ to configuration variable names (to prevent ambiguities).** This option helps avoid problems if a port and configuration variable have the same name.

**Export each test to a separate file.** Instead of exporting to a single file and separating tests by an empty line, each test is exported to a separate file. The file name specified in the Output File box will be used as the base file name to which \_csv will be appended to form the file name (where \_N\_ is the test number).

**Export test point column headers as.** Selects the column header content for exported test points: “name” will show the test point’s name as specified in the test point parameter dialog, “location” will show the test point’s complete path within the model hierarchy. Select “legacy” for to use the same header format as Reactis V2016.2 and earlier for backwards compatibility.

**Export boolean values as.** Select whether boolean values will be exported as integers (0, 1) or doubles (0.0, 1.0).

**Export enumerated values as.** Lets you specify each enumerated value should be exported as its underlying integer or as its name.

**Significant digits for floating-point values.** Lets you specify the number of significant digits to be used when exported floating point values.

### 7.7.2 Importing Test Suites

Reactis can also import tests and add them to the current test suite. Test suites may be imported if they are stored in the Reactis’s native \_rst file format or in the comma separated value (CSV) format (described above) that Reactis exports. The import feature is launched by selecting **Test Suite → Import...** when Simulator is enabled.

To execute a test suite in Simulator, the test suite must **match** the executing model. A test suite matches a model if it contains data for the same set of inports, outports, test points, and configuration variables as the model. If an externally visible data item (inport, outport,
test point, or configuration variable) is added to or removed from a model, then previously constructed test suites will no longer match the new version of the model. The import facility gives you a way to transform the old test suite so that it matches the new version of the model. Such remapping is also available when importing .csv files.

![Figure 7.21: The Import Test Suite dialog allows you to import external test data (comma separated value format) and if necessary transform the data to produce a test suite that matches a model. The import facility is also used to transform an .rst file to make it match a model.](image)

The Import Dialog, shown in Figure 7.21, is used to specify how test data should be remapped during import. The dialog contains a tab for each type of data item stored in a test suite (inputs, outputs, test points, configuration variables). In the case of .csv files, the import dialog also contains a tab Not Imported that lists items present in the CSV file that are not scheduled to be imported into the new test suite. When an .rst file includes an item not scheduled to be imported, it is placed at the bottom of the appropriate tab. For example, if a test suite contains an inport X and is being imported with respect to a model that has no inport X, then X will appear at the bottom of the Input Ports tab and be highlighted in yellow.

Each data item tab (e.g. Input Ports) includes a column (e.g. Model Port Name) listing the model items in that category. The Suite column lists items from the file being imported that map to the corresponding model item. In most cases a data item X in the test suite being imported will map to an item with the same name in the model. If the model contains an item not found in the test suite being imported, then the corresponding Suite column entry will be listed as Random Value and be highlighted in yellow (as shown in Figure 7.21 for inport brake). If this setting is not changed, then upon import a random value will be generated for the import at each test step. The value to be assigned for any model item may be changed by double clicking on the corresponding entry in the Suite column (alternatively selecting the item and clicking the button Select Suite Item) and then using the resulting dialog to either select an item from the test suite being imported or set it to Random Value.

If the model has bus input ports, Import Test Suite, these will be marked with a [+] or [-] to the left of the port name. If a name is preceded by [+] this indicates that there are hidden sub-components which are not currently being displayed. Conversely, if a name is preceded by [-], this indicates that there are sub-components which appear immediately below the name, and are indented to indicate their status as a sub-component. Left-clicking on the name of a bus input toggles the display of its sub-components. Figure 7.22 shows a test suite being imported
into a model containing a bus input.

Finally, note that when loading a test suite stored in the pre-V2006 format, the test suite will be automatically converted to the new format.

7.8 Updating Test Suites

When a test suite is loaded in Simulator, this feature can be invoked by selecting menu item Test Suite → Update… to open the dialog shown in Figure 7.23. The dialog offers two methods for updating an existing test suite (.rst file):

- The first method Update using Reactis executes a test suite in Reactis, updates items (inputs, configuration variables, test points, and outputs) as configured by the checkboxes, and writes the modified suite to a new .rst file.

- The second method Update outputs using Simulink exports the test suite to .mat format, executes the exported suite in Simulink, updates the test suite to contain the output values computed by Simulink, saves the updated suite to a .mat file or .m file (in the format supported by the Reactis rsRunTests utility).

7.8.1 Update Using Reactis

This method creates a new test suite by simulating the current model using inputs from the current test suite, but recording values for outputs and test points generated by the model. Updating the outputs can be useful for updating test suites when a model is modified, but its input ports remain unchanged. The result of invoking this routine with these options selected is a new test suite with the same input values at each step, but with outputs and test points updated (as specified in the dialog) with values generated by the currently loaded version of the model.

Additionally, inputs and configuration variables can also be updated to conform with constraints specified in the .rsi file. When these options are selected, if an input or configuration
variable has a test suite value not conforming to its constraint, then the test suite is updated to contain the closest value adhering to the constraint.

The five check-boxes specify what is written to the new test suite as follows:

<table>
<thead>
<tr>
<th>Contents of new test suite when:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Checked</td>
<td>Not Checked</td>
</tr>
<tr>
<td>Adapt top-level input port values to match constraints</td>
<td>If an input value in the test suite does not conform to the constraint specified for that input in the .rsi file, then convert the input value to the closest value adhering to the constraint.</td>
</tr>
<tr>
<td>Update inputs controlled by virtual sources</td>
<td>Each input currently controlled by a virtual source shall be updated with the values computed by the controlling virtual source at each step as the tests execute.</td>
</tr>
<tr>
<td>Adapt configuration variable values to match constraints</td>
<td>If a configuration variable value in the test suite does not conform to the constraint specified for that variable in the .rsi file, then convert the value to the closest value adhering to the constraint.</td>
</tr>
<tr>
<td>Update test points</td>
<td>Each test point shall be updated with the value computed by the model for the test point as the tests execute.</td>
</tr>
<tr>
<td>Update outputs</td>
<td>Each output shall be updated with the value computed by the model for the output as the tests execute.</td>
</tr>
</tbody>
</table>

### 7.8.2 Update Outputs Using Simulink

This method creates a new test suite by loading the current model in Simulink, executing it (in Simulink) using the input values from the current test suite, and capturing the outputs pro-
duced by the model. The inputs and outputs are stored using the .m or .mat format supported by the Reactis rsRunTests utility.

7.9 Model Highlighting

Simulator renders model diagrams using a number of different colors and line styles to convey information during simulation. In this section, we describe these different drawing styles and their semantics.

Some of the default drawing colors are as follows. During slow, single-step, or mini-step simulation, a model element is drawn in green while it is being evaluated. Selecting Coverage → Show Details configures Reactis to highlight uncovered model elements in red and unreachable model elements in purple. Please refer to Chapter 6 for a description of the different coverage metrics tracked by Reactis.

![Select Line Styles dialog](image)

Figure 7.24: The Select Line Style dialog.

The dialog shown in Figure 7.24 (invoked by selecting View → Select Line Styles...) enables you to configure how Simulator should draw various diagram elements. Each row in the dialog specifies the rendering of one group of model elements. The different groups of configurable diagram elements are:

**Uncovered Block.** A Simulink block B is in this group if it has not been fully exercised, i.e. one of the following holds:

- B is a block included in the branch coverage metric and at least one of B’s branches remains uncovered.
• B is a conditional subsystem that has never been exercised.
• B is a logical operator block that has not satisfied the requirements for all MC/DC-related coverage metrics (decision, condition, and MC/DC).
• B is a relational operator block and some boundary value has not been exercised.
• B is a top-level import and not all boundary values have been exercised.
• B is a block with *saturate on integer overflow* enabled and some boundary value has not been exercised.
• B is a Lookup table and has uncovered lookup targets.
• B is an uncovered user-defined target.
• B is a violated assertion.

**Uncovered State.** A Stateflow state is in this group if it has never been entered.

**Uncovered Condition Action.** A Stateflow transition segment is in this group if it has not met the requirements for condition action coverage. In other words, its condition action has never been evaluated. If a segment has no condition action, then it is considered uncovered according to condition action coverage if its condition has never evaluated to true. Note that a segment with an empty condition is assumed to evaluate to true whenever the segment is evaluated during simulation.

**Uncovered Transition Target.** A Stateflow transition segment is in this group if it has met the requirements for condition action coverage, but has not met the requirements for one of the other coverage metrics associated with transition segments. These metrics include transition action coverage, decision coverage, condition coverage, MC/DC, and CSEPT.

**Unreachable Block.** A Simulink block B is in this group if Reactis has determined that some aspect of the block’s behavior can never happen and that all behaviors that are possible have occurred. If some reachable behavior has not occurred, then the block is considered a member of the “Uncovered Block” group and rendered accordingly.

**Unreachable State.** A Stateflow state is in this group if it can never be entered.

**Unreachable Condition Action.** A Stateflow transition segment is in this group if it can never meet the condition action coverage requirement.

**Unreachable Transition Target.** A Stateflow transition segment is in this group if it has met the requirements for condition action coverage, but cannot meet the requirements for one of the other coverage metrics associated with transition segments.

**Active Block.** A Simulink block is in this group if it is currently being evaluated.

**Active State.** A Stateflow state is in this group if it is currently active.

**Active Condition Action.** A Stateflow transition segment is in this group after its condition evaluates to true and until the next model element is highlighted as active.

**Active Transition Action.** A Stateflow transition segment is in this group as it is firing as a part of a transition.

---

3Note that “condition coverage” and “condition action coverage” are two distinct metrics.
Chapter 8

Reactis Tester

Reactis Tester generates tests automatically from Simulink / Stateflow models. Each test consists of a sequence of stimulus / response pairs, where each stimulus assigns an input value to each top-level inport of the model and each response records an output value for each top-level outport. The tests may be run on Simulink / Stateflow models using either Reactis Simulator or the Simulink simulation environment of The MathWorks, Inc. The tests may also be used to drive testing of source code implementations of models as described in Chapter 12.

A collection of tests generated by Tester is termed a test suite. Test suites are intended to maximize coverage for the different metrics defined in Chapter 6 while minimizing the total number of execution steps in the overall suite. The remainder of this chapter describes the workings of Tester.

8.1 The Tester Launch Dialog

Figure 8.1 contains an annotated screen shot of the Tester launch dialog, which may be invoked from the Reactis top-level window using the Test Suite → Create... menu item. This section describes the labeled items in the figure.

1. Specify how long Tester should run. There are three options to choose from:
   - A fixed amount of time
   - A fixed number of steps
   - A specified number of random and targeted tests/steps.

2. The Preload Files section lets you optionally specify a list of test suites to be executed before generating new test data. The coverage achieved by the preloaded tests will be tracked and subsequently generated tests will aim to exercised targets not covered by the preloaded tests. If no preload suites are given, new test data will be generated from scratch.

   When the check-box column to the right of a file name titled ‘Prune’ is checked, unnecessary steps (those that do not increase the level of coverage) will be pruned from the ends of tests in the preloaded suite. When the check-box is not checked, no pruning of the suite will occur. When the check-box column titled ‘Use Virtual Sources’ is checked, the Virtual Source signals that are mapped to specific inports of the current model will be used, rather than the inport signals stored in the named .rst file. Signals which have no virtual source mapping will be derived from the .rst file, as usual.
3. Clicking this button invokes a file-selection dialog that enables you to specify an .rst file to be added to the preload list.

4. Clicking this button removes the currently selected .rst file from the list of test suites to be preloaded.

5. When running Tester for a fixed length of time, the number of hours and minutes is entered here. If 100% coverage of all targets is reached prior to the time limit, Tester will terminate early.

6. When running Tester for a fixed number of steps, the number of steps is entered here. Tester will decide how many of these steps will be random or targeted. Because of pruning, the number of steps in the final test suite will typically be less than the number entered here.

7. When running Tester for a specified number of random and targeted steps, the number of tests in the random phase is entered here. Because of pruning that occurs at the end of
the random phase, some tests may be eliminated entirely, leading to a smaller number of tests at the end of the random phase than what is specified here.

8. In order to minimize test suite size, Tester usually prunes random tests after the random phase, removing any steps at the end of a test that do not cover any new targets. Unchecking this option will skip the pruning operation such that all random tests will be added to the test suite in their full length.

9. When running Tester for a specified number of random and targeted steps, the number of steps to take while constructing each test of the random phase is entered here. Upon completion of the random phase, unimportant steps are pruned from the ends of tests, so the lengths of the final tests will usually be shorter than the length specified here.

NOTE: Specifying too many steps in the random phase can cause Reactis to run out of memory. The upper bound on the number of steps possible depends on model size, but in general much more time should be spent in the targeted phase which is more optimized for memory usage.

10. When running Tester for a specified number of random and targeted steps, the number of execution steps to take during the targeted phase is entered here. The targeted phase uses sophisticated strategies to guide the simulation to exercise parts of the model not visited during the preload or random phases. The value entered specifies an upper bound on the number of simulation steps executed during the targeted phase.

11. The Coverage Objectives section enables you to specify the coverage metrics used to guide test generation. Each coverage metric defines a set of target elements in models to exercise. The metrics supported by Tester are described in Chapter 6.

12. This entry box enables you to pass one or more of the following parameters to Tester:

   - `-a1` turns inputs abstraction on, `-a0` turns inputs abstraction off. Inputs abstraction usually improves the performance of Tester and should be left on (default). In rare cases, turning it off may improve coverage. If coverage problems are encountered with inputs abstraction on, it may be beneficial to take a test suite produced with abstraction on, preload it into Tester, turn abstraction off, and then run Tester again.
   - `–c n` sets the maximum number of input variables that may change during an execution step to `n`, which must be a positive integer. The default is that every input variable can change at every step. Restricting the number of input variables that can change can lead to easier-to-understand test suites.
   - `–s randomSeed` seed for the random number generator. This is useful for replaying a previous run of Tester. The random seed used to create a .rst file can be found in the test-suite log (which may be viewed in the Test Suite Browser described in Chapter 11), after the “–s” in the “Created by Tester:” line.

13. The name of the .rst file to be generated.

14. Clicking this button opens a file-selection dialog for specifying the name of the .rst file to be generated.

15. Clicking this button displays Tester help.
16. Clicking this button opens a file selection dialog to specify an .rtp file from which to load Tester launch parameters. Reactis may be configured from the General pane of the Info File Editor to generate an .rtp file for each Tester or Validator run.

17. Reset the Tester launch parameters to the default values Reactis ships with.

18. Scroll backward in the parameter history.

19. Scroll forward in the parameter history.

20. This button is visible only if you went back to this screen by clicking the Back button in the Tester progress/results dialog. Clicking it will bring back the progress/results dialog with its previous results.

21. Clicking this button starts test-suite generation.

22. Clicking this button closes the Tester launch dialog without starting test-generation.

---

## 8.2 The Progress Dialog

The Tester progress dialog, shown in Figure 8.2, is displayed while Tester is running. What follows describes the labeled items in the figure. When Tester completes, buttons 7–11 become enabled.


2. Time elapsed (real time) since start of test generation, and estimated time until test generation completes.

3. Total number of simulation steps to be taken during test generation, and the percentage thereof taken so far. Note that since many steps are pruned from final test suite, this number will typically be much larger than the cumulative number of execution steps in the generated test suite.

4. Progress bars that report percentages of reachable targets covered for Simulink-specific coverage metrics. These metrics are described in Section 6.1.

5. Progress bars that report percentages of reachable targets covered for Stateflow-specific coverage metrics. These metrics are described in Section 6.2.

6. Progress bars that report percentages of generic (applying Simulink, Stateflow, and C code) coverage metrics. These metrics are described in Section 6.3.

7. When warnings occur, this button may be clicked to view descriptions of the warnings.

8. Open the Reactis Test Suite Browser and load the newly generated test suite.

9. Enable Reactis Simulator and load the newly generated test suite.

10. View the model with coverage information for the newly generated test suite represented by highlighting the diagram elements of the model.
11. View the model with coverage information for the newly generated test suite in the Coverage-Report Browser.

12. View help for the progress dialog.

13. Enable Reactis Simulator, load the newly generated test suite, and close the Tester results dialog.

14. Go back to the Tester launch dialog.

15. Interrupt test generation and write the tests generated so far to an .rst file.
Chapter 9

Reactis Validator

Reactis Validator searches for defects and inconsistencies in models. The tool enables you to formulate a property that model behavior should have as an assertion, attach the assertion to a model, and perform an automated search for a violation of the assertion. If Validator finds an assertion violation, it returns a test that leads to the problem. This test may then be executed in Reactis Simulator to gain an understanding of the sequence of events that leads to the problem.

Validator also allows you to specify specific test scenarios that you want exercised using one of two alternative mechanisms. **User-defined targets** enable you to specify an abstract test scenario, whereas **virtual sources** give you a way to easily specify a concrete scenario.

User-defined targets (Section 9.1.2) may be seen as extensions to the built-in coverage metrics supported by Reactis: in generating test data, Validator (and Tester) will attempt to generate tests that satisfy the indicated scenarios. Like assertions, user-defined targets are virtually attached to models within Reactis, so your model is not modified.

Virtual sources (Section 9.1.3) are placed at the top level of a model to control one or more top-level inports as a model executes in Reactis. That is you can specify a sequence of values to be consumed by an inport during simulation or test-generation. Virtual sources can be easily enabled and disabled. When enabled, the virtual source controls a set of inports and while disabled those inports are treated by Reactis just as normal top-level inports.

Validator is particularly useful in requirements validation. Given a list of requirements on model behavior, you can formulate assertions (to check whether a requirement is being satisfied) and user-defined targets or virtual sources (to define test scenarios intended to “stress” the requirement).

Engineers use Validator as follows. First, a model is instrumented with assertions to be checked, user-defined targets, and virtual sources. We refer to assertions, user-defined targets, and virtual sources as **Validator objectives**. The tool is then invoked on the instrumented model to search for assertion violations and paths leading to the specified user-defined targets. The output of a Validator run is a test suite that includes tests leading to objectives found during the analysis.

Assertions and user-defined targets may be added to any Simulink system or Stateflow diagram in a model; whereas, virtual sources may only be added at the top-level of the model. When adding Validator objectives to a model, three mechanisms (only the first two are available for virtual sources) for formulating objectives are supported:

**Expression objectives** are C-like Boolean expressions.

**Diagram objectives** are references to subsystems in standard Simulink/Stateflow libraries.
**Timer objectives** are directives that tag a model data item as a timer or counter.

An expression objective is a C-like Boolean expression whose variables are wired to data items from the context of the model in which the objective is inserted. Expression objectives are easily attached and modified from the Reactis GUI. For more information on valid expressions see Sections 9.3.1 and 9.3.2.

Diagram objectives give users the full power of Simulink / Stateflow to formulate assertions, user-defined targets, and virtual sources. The objectives may use any Simulink blocks supported by Reactis, including Stateflow diagrams. Diagram objectives are attached to a model using the Reactis GUI to select a Simulink system from a library and “wire” it into the model. The diagrams are created using Simulink and Stateflow in the same way standard models are built. After adding a diagram objective to a model, the diagram is included in the model’s hierarchy tree, as are other library links in a model.

A timer objective indicates that a model element acts as a timer or counter. A timer target is covered when the value of the selected data item is changed from the start value to the end value (typically by repeatedly adding the step value to the data item, although this is not strictly necessary — the timer target will still be covered if some or all of the intermediate values are skipped). For some models, adding timer targets will help Reactis Tester attain higher levels of coverage. Conceptually, this is because the timer target guides Reactis to generate tests which continue stepping until a timer expires or a counter reaches its final value. In the case of a timer assertion, the assertion is considered violated if the conditions which would cause a timer target with the same parameters to be covered occur.

---

**9.1 The Meaning of Validator Objectives**

We now explain in more detail how Reactis interprets Validator objectives. Coverage of these objectives may be tracked in the same manner as built-in coverage targets (e.g. MC/DC); namely through highlighting and hovering in the main Reactis panel, through the Coverage-Report Browser, and through reports generated by **Simulate → Fast Run with Report**.

**9.1.1 Assertions**

Assertions specify properties that should always be true for a model. For a given assertion, Validator searches a model for a simulation run that leads to a state in which the assertion does not hold. An expression assertion fails to hold if the given expression evaluates to false\(^1\). A timer assertion fails to hold if the data item it tracks ever changes from its start value to the end value by a specified increment. A diagram assertion fails to hold if any output port assumes the boolean value “false” or a numeric value zero. Note that if a diagram has more than one output port, the individual outports are treated as distinct assertions. That is each individual outport is listed in the objective list of the Info File Editor. Also, the individual ports are highlighted independently in the main panel according to whether they are violated or not.

An assertion is considered “covered” when a violation is found. So in contrast to targets, where being covered is considered good, covering an assertion is bad. We therefore highlight *covered* assertions in red and yellow, in contrast to targets where *uncovered* targets are highlighted in red.

\(^1\) The expression language for Validator objectives employs the C language convention of representing false as the numeric value zero and true by any non-zero value
9.1.2 User-Defined Coverage Targets

User-defined coverage targets extend the Reactis built-in targets described in Chapter 6 (branch coverage, state coverage, condition-action coverage, MC/DC, etc). If a user-defined target is specified as an expression, Reactis will treat it as covered when the expression evaluates to a non-zero numeric value. If a user-defined target is specified as a timer, the target is considered covered after the data item tracked by the timer changes from its start value to its end value by its specified increment. If a user-defined target is specified as a diagram, Reactis will treat it as covered when all output ports of the associated subsystem have assumed either a boolean “true” value or any non-zero numeric value. Note that if a diagram has more than one output port, the individual port names are listed in the target list and highlighted independently according to whether they are covered or not.

9.1.3 Virtual Sources

Virtual sources control top-level inports of a model. If a virtual source is an expression, then at each simulation step, the expression is evaluated to compute a value that will be fed into the top-level inport controlled by the virtual source. If a virtual source is specified as a diagram, then each outport of the diagram may be fed into a top-level inport of the model. Some outports of the virtual source diagram may be left unconnected and monitored by an assertion.

9.2 Use Cases of Validator Objectives

9.2.1 Checking a Requirement with an Expression Assertion and an Expression User-Defined Target

Consider a cruise control application that has the following requirement:

The cruise control shall remain inactive if the speed of the vehicle is less than 30 mph.

To properly test this requirement we need to execute a test in which an attempt is made to activate the cruise control while the speed of the vehicle is less than 30 mph. We can capture this as a simple expression-based user-defined target:

\[
\text{on} \land \text{activate} \land (\text{speed} < 30)
\]

This expression would be "wired-in" such that on monitors whether the cruise control is turned on, activate monitors whether an attempt has been made to activate the cruise control, and speed is the speed of the vehicle. When this expression becomes true, we have found a test to check our requirement.

We can test whether the applications response to this test is correct with the following expression assertion:

\[
!((\text{active} \land (\text{speed} < 30))
\]

This expression is wired in so that active monitors whether the cruise control is active. If this expression ever becomes false, then the requirement is violated. Alternatively, this requirement could be specified with an equivalent expression that uses the logical implication operator:

\[
\text{active} \Rightarrow (\text{speed} \geq 30)
\]
9.2.2 Checking a Requirement with a Diagram Assertion

Consider another requirement for a cruise control:

When active, cruise control shall not permit actual, desired speeds to differ by more than 1 mph for more than 3 seconds.

This requirement can be checked by a diagram assertion that monitors the vehicle speed, whether the cruise control is active, and the speed at which the driver has set the cruise control. The top-level interface of the Simulink subsystem (in this case a Stateflow diagram) is shown in Figure 9.1.

![Diagram](image_url)

Figure 9.1: The diagram to check the speed maintenance requirement monitors three data items in the model and raises a flag on its output if the requirement is violated.

The simple Stateflow diagram that implements the assertion is shown in Figure 9.2. It works as follows:

1. Initially the diagram is in the *Inactive* state.
2. When the cruise control becomes active it enters state *Active* and child state *OkDiff*.
3. When in *OkDiff*, the diagram computes the difference between the vehicle speed and the speed at which the driver has set the cruise control.
   (a) If the difference exceeds the tolerance, then go to state *BigDiff*.
   (b) If the difference is less than the tolerance, then stay in *OkDiff*.
4. When entering *BigDiff*, start a counter
   (a) If the speed difference is corrected within 3 steps, then return to *OkDiff*.
   (b) If the speed difference is not corrected within 3 steps, then flag an error by outputting the value 0 on its outport.
9.2. USE CASES OF VALIDATOR OBJECTIVES

9.2.3 Creating a Functional Test Using a Virtual Source

Section 3.4.6 showed how you can create a functional test for the cruise control example using the user-guided simulation feature of Reactis Simulator. Virtual sources offer an alternate mechanism for capturing functional tests. Consider the need to test that the cruise control activates and deactivates as expected when executing the following scenario:

<table>
<thead>
<tr>
<th>Input</th>
<th>Expected Output of Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Turn cruise control on</td>
<td>Off</td>
</tr>
<tr>
<td>2) Press set</td>
<td>On</td>
</tr>
<tr>
<td>3) Press cancel</td>
<td>Off</td>
</tr>
<tr>
<td>4) Press set</td>
<td>On</td>
</tr>
<tr>
<td>5) Press brake</td>
<td>Off</td>
</tr>
<tr>
<td>6) Press resume</td>
<td>On</td>
</tr>
<tr>
<td>7) Press gas</td>
<td>Off</td>
</tr>
</tbody>
</table>

This scenario can be captured using the virtual source shown in Figure 9.3. The Signal Builder block shown implements the virtual source by capturing the scenario described above. The first six signals are fed into inputs of the model while the final output is monitored by a simple assertion that compares the expected value for active against the actual value produced by the model.
CHAPTER 9. REACTIS VALIDATOR

Figure 9.3: Virtual sources offer an easy way to implement functional tests. Any Simulink / Stateflow construct supported by Reactis may be used to implement the virtual source. Signal Builder blocks are especially useful. A Validator assertion can monitor an output of the virtual source, relieving the user of the tedious task of checking expected values against actual responses of the model.

9.3 Adding, Editing, and Removing Objectives

Adding, editing, and removing of objectives is only possible when Simulator is disabled. This is necessary because objectives are linked into a model when Simulator is invoked.
A Validator objective may be added to a model by first selecting in the hierarchy panel the Simulink subsystem or Stateflow state in which the objective should be inserted and then:

- right-clicking in an empty space in the subsystem in the main panel of the Reactis window and selecting Add Assertion or Add User-Defined Target or Add Virtual Source; or
- by selecting the Validate → Add Assertion or Validate → Add User-Defined Target or Validate → Add Virtual Source menu items from the main menu bar.

Usage of the resulting dialogs is explained in Section 9.3.1 (expression objectives within Simulink), Section 9.3.2 (expression objectives within Stateflow), Section 9.3.3 (timer objectives), and Section 9.3.4 (diagram objectives).

In Stateflow charts, only expression objectives are supported. Using a group of radio buttons in the wiring dialog, you can choose for the objective to be evaluated at state entry or exit, or every time the chart is triggered while the state is active. These correspond to the usual entry, exit, and during actions of Stateflow. Evaluation of the objective occurs at the end of the corresponding state actions (entry, during, exit).

To edit properties of an objective, either right-click on the objective and select Edit → Properties..., or click on the objective to select it and then choose the Validate → Edit Objective... menu item. Note that all the controls in the properties dialog are disabled if Simulator is currently active. To remove an objective, either:

- right-click on the objective and select Remove, or
- click on the objective to select it and then choose the Validate → Remove Objective menu item, or
- click on the objective to select it and then press the delete key.

Standard cut, copy, and paste operations are available for objectives both from the top-level Edit menu and by right-clicking on an objective. Edit → Undo and Edit → Redo enable you to undo and redo these operations. Objectives may be moved by dragging them with the mouse.

### 9.3.1 The Simulink Expression Objective Dialog

The numbers below refer to the labels in Figure 9.4.

1. **Objective name.** The name must be unique with respect to the subsystem where the objective resides.

2. **Enable or disable the objective.** When disabled:
   - assertions are not checked
   - user-defined targets are not tracked
   - virtual sources do not control inports

3. **Expression.** Reactis evaluates this expression at each simulation step and interprets the scalar result of numeric type as follows:
   - **Assertions.** A zero result means a fail has occurred.
   - **User-Defined Targets.** A non-zero result means the target is covered.
4. # steps to hold. This is a non-negative integer value. For assertions, this entry specifies the number of simulation steps that the expression must remain false before flagging an error. For user-targets, the entry specifies the number of steps that the expression must remain true before the target is considered covered.

5. Inputs. This is where variables used to construct the expression are declared. Each variable can be viewed as an input to the objective. Clicking the Add Variable button adds a row to the section. The rightmost entry box of a row is where you enter the name of a variable. The column(s) to the left of the variable name specify which data item from the model is wired to the variable. Clicking the X button to the right of a row deletes the variable declaration.

Note that you can leave any of these selections empty and then later connect them via...
9.3. ADDING, EDITING, AND REMOVING OBJECTIVES

<table>
<thead>
<tr>
<th>Symbol/Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, /</td>
<td>Add, Subtract, Multiply, Divide</td>
</tr>
<tr>
<td>&lt;, &gt;</td>
<td>Less than, Greater than</td>
</tr>
<tr>
<td>&lt;=, =&gt;</td>
<td>Less than or equal, Greater than or equal</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Logical Implication</td>
</tr>
<tr>
<td>!</td>
<td>Logical Not</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical And</td>
</tr>
<tr>
<td>[]</td>
<td>Vector element access</td>
</tr>
<tr>
<td>a.b</td>
<td>Access element b of bus a</td>
</tr>
<tr>
<td>abs(), fabs()</td>
<td>Absolute value</td>
</tr>
<tr>
<td>x^y, pow(x,y), power(x,y)</td>
<td>x to the power y</td>
</tr>
<tr>
<td>exp()</td>
<td>Exponent</td>
</tr>
<tr>
<td>ln(), log(), log10()</td>
<td>Logarithm</td>
</tr>
<tr>
<td>sqrt()</td>
<td>Square root</td>
</tr>
<tr>
<td>rem()</td>
<td>Division remainder</td>
</tr>
<tr>
<td>inf</td>
<td>Infinity value</td>
</tr>
<tr>
<td>floor(), ceil()</td>
<td>Rounding</td>
</tr>
<tr>
<td>cos(), sin(), tan()</td>
<td>Trigonometric functions</td>
</tr>
<tr>
<td>cosh(), sinh(), tanh()</td>
<td>Hyperbolic trigonometric functions</td>
</tr>
<tr>
<td>acos(), asin(), atan(), atan2()</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>pre()</td>
<td>Access values from previous simulation steps (see Section 9.3.5).</td>
</tr>
</tbody>
</table>

Table 9.1: Simulink expression objective operators and functions.

drag-and-drop in the main panel (see Section 9.3.6). Any variables referenced in the expression that are not declared in this section will be automatically added to the list (with empty wiring).

When the dialog is dismissed, the wiring specified here may be viewed from the main panel of the Reactis window by hovering over the objective. The wiring will be shown as blue lines which can be thought of as virtual wiring. For each input of the objective, a blue line will be drawn from a data item in the model to the input to indicate how the input is wired.

6. Auto-Wire. Clicking on this causes Reactis to attempt to automatically select the data items from the model that should feed into the inputs of the objective. For an inport $X$ of an objective $O$, the pairing algorithm selects a data item $D$ if:

- $D$ is a Simulink block named $X$, or
- $D$ is an outport named $X$ of a subsystem with the same parent as $O$

If there are no matches or more than one possible match, the auto-wiring attempt will fail for $X$ and the inport will need to be manually bound to a data item.

For virtual sources an additional pull-down menu at the bottom of the dialog enables you to specify the model inport that the virtual source controls.

9.3.2 The Stateflow Expression Objective Dialog

The numbers below refer to the labels in Figure 9.5.
1. Objective name. The name must be unique with respect to the objectives in the state where the objective resides.

2. Enable or disable the objective. When disabled:
   - assertions are not checked
   - user-defined targets are not tracked

3. Expression. Reactis evaluates this expression at the point determined by window item 6 and interprets the scalar result of numeric type as follows:
   - **Assertions.** A zero result means a fail has occurred.
   - **User-Defined Targets.** A non-zero result means the target is covered.

   Valid operators and functions are listed in Table 9.2.

4. # steps to hold. This is a non-negative integer value. For assertions, this entry specifies the number of simulation steps that the expression must remain false before flagging an
### 9.3. ADDING, EDITING, AND REMOVING OBJECTIVES

<table>
<thead>
<tr>
<th>Symbol/Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -, *, /</td>
<td>Add, Subtract, Multiply, Divide</td>
</tr>
<tr>
<td>&lt;, &gt;</td>
<td>Less than, Greater than</td>
</tr>
<tr>
<td>&lt;=, =&gt;</td>
<td>Less than or equal, Greater than or equal</td>
</tr>
<tr>
<td>==</td>
<td>Equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
<tr>
<td>-&gt;</td>
<td>Logical Implication</td>
</tr>
<tr>
<td>!</td>
<td>Logical Not</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical And</td>
</tr>
<tr>
<td>[]</td>
<td>Vector element access</td>
</tr>
<tr>
<td>a.b</td>
<td>Access element b of bus a</td>
</tr>
<tr>
<td>abs(), fabs()</td>
<td>Absolute value</td>
</tr>
<tr>
<td>x^y, pow(x,y)</td>
<td>x to the power y</td>
</tr>
<tr>
<td>exp()</td>
<td>Exponent</td>
</tr>
<tr>
<td>log(), log10()</td>
<td>Logarithm</td>
</tr>
<tr>
<td>sqrt()</td>
<td>Square root</td>
</tr>
<tr>
<td>floor(), ceil()</td>
<td>Rounding</td>
</tr>
<tr>
<td>cos(), sin(), tan()</td>
<td>Trigonometric functions</td>
</tr>
<tr>
<td>cosh(), sinh(), tanh()</td>
<td>Hyperbolic trigonometric functions</td>
</tr>
<tr>
<td>acos(), asin(), atan(), atan2()</td>
<td>Inverse trigonometric functions</td>
</tr>
<tr>
<td>double(), single(), boolean(), int8(), ...</td>
<td>Type cast functions</td>
</tr>
<tr>
<td>int16(), int32(), uint8(), uint16(), uint32()</td>
<td>Access values from previous simulation steps (see Section 9.3.5).</td>
</tr>
<tr>
<td>pre()</td>
<td>If expr1 is true, evaluate expr2. Otherwise evaluate expr3.</td>
</tr>
</tbody>
</table>

| Table 9.2: Stateflow expression objective operators and functions. |

error. For user-targets, the entry specifies the number of steps that the expression must remain true before the target is considered covered.

5. Visible variables and constants. This lists the variables and constants visible in the scope where the objective is located. Double-clicking on an item in this list will insert it into the expression (window item 5) text box.

6. Specifies the point during execution when this objective is checked:

**When executing state entry actions** The objective is checked after the “entry” actions of the state in which it is located execute.

**When executing state during actions** The objective is checked after the “during” actions of the state in which it is located execute.

**When executing state exit actions** The objective is checked after the “exit actions” of the state in which it is located execute.

**On chart entry** The objective is checked every time the chart becomes active, before any events are processed within the chart.

**On chart exit** The objective is checked every time the chart becomes inactive, after all events have been processed within the chart.

### 9.3.3 The Timer Objective Dialog

The timer objective dialog is used to add a timer target to a model or edit the parameters of an existing timer target. A timer target can be added by right clicking within the main panel and
selecting Add User-Defined Target → Timer. Timer targets can be added within blank areas of Simulink subsystems, or anywhere within a Stateflow chart (including inside a state). A timer objective dialog is shown in Figure 9.6. The numbers below refer to the labels in that Figure.

1. Name. The name must be unique within the subsystem where the objective resides:
   - If a timer objective is located in Simulink, then its name must not be the same as any block or other Validator objective residing in the same subsystem.
   - If a timer objective is located in Stateflow, then its name must not be the same as any other Validator objective residing in the same state.

2. Timer. The data item from the model to be tracked as a timer. The pull-down menu lists all possible data items in the subsystem/state where the objective will reside.

3. Enable or disable the objective. When disabled the timer target is not tracked.

4. Start value. The initial value of the timer.

5. Step size. The amount the timer increases or decreases during a step.

6. End value. The end value of the timer.

A typical use-case for a timer target is shown in Figure 9.7. It shows a subsystem which signals a fault when a sensor input is negative for 100 consecutive simulation steps. The timer target Sustained Fault is covered when the output value of the Product block reaches 110 (at which point the output fault will have been true for 10 consecutive steps). Adding this target to the model helps Tester rapidly reach 100% coverage.

One important note: in Figure 9.7, the timer target does not use the output of the Count block as the counter. Instead it uses the input to the Count block. When a unit delay block is used as a timer/counter, Tester generally produces better coverage for timer targets when using the input to the unit delay block rather than the output of the block. The recommended best practice for creating timer targets which rely on unit delay blocks is to use the input to the block rather than the output as the basis for the target.
9.3.4 The Diagram Objective Dialog

The Diagram Objective Dialog is the vehicle for adding diagram-based assertions and user-defined targets. It is also used to modify existing diagram objectives. You invoke it by right-clicking in the main panel in white space within a Simulink subsystem and selecting either:

- Add User-Defined Target → Diagram, or
- Add Assertion → Diagram

The numbers below refer to the labels in Figure 9.8.

1. Objective name. The name must not be the same as any block or Validator objective within the subsystem where the objective resides.

2. Enable or disable the objective. When disabled:
   - assertions are not checked
   - user-defined targets are not tracked
   - virtual sources do not control inports

3. Name of the currently chosen objective library. This is a standard .slx file or .slx file storing a Simulink library containing the Simulink / Stateflow subsystems that will be wired into a controller model as Validator diagram objectives. This file is constructed in the usual manner using The MathWorks’ Simulink and Stateflow editors. Note that the actual wiring is established and maintained by Reactis, so the controller model being validated need not be modified at all. The wiring information is stored by Reactis in the .rsi file associated with a model.

4. Clicking here opens a file selection dialog for choosing a Validator objective library.

5. System. This tree represents the structure of the library specified in window item 3. Selecting an item in the tree indicates the Simulink subsystem from the library that encodes the objective.

6. Parameters. If the system chosen above is a masked subsystem, the values for the mask parameters may be entered here.

Figure 9.7: The timer target *Sustained Fault* is covered when the output of the *Product* block steps from 0 to 110.
7. Inputs. This section enables the user to specify the wiring between input ports of the diagram objective and the blocks in the subsystem of the model where the objective resides. Labels on the right represent the input ports of the diagram objective. The menus to the left list data items from the model that may be wired to the inputs of the objective. Note that an entry in the second column of menus only appears if the item selected in the first column is a block with more than one output. The second column menu selects one of the multiple outputs of the block in column one.

Note that you can leave any of these selections empty and then later connect them via drag-and-drop in the main panel (see Section 9.3.6).

When the dialog is dismissed, the wiring specified here may be viewed from the main panel of the Reactis window by hovering over the diagram objective. The wiring will be shown as blue lines connecting inputs of the objective to data items that flow into the inputs from the model being instrumented for validation.

If the diagram objective is a virtual source, the dialog will have an additional Outputs section at the bottom. This section contains pull-down menus that enable you to specify the model inports that the virtual source controls. Each output of the virtual source may control one input of the model. Virtual source outputs left unconnected may be monitored by assertions.

When the dialog is dismissed, the wiring specified here may be viewed from the main panel of the Reactis window by hovering over the diagram objective. The wiring will be shown as blue lines connecting inputs of the objective to data items that flow into the inputs from the model being instrumented for validation.
panel of the Reactis window by hovering over the diagram objective. The wiring will be shown as blue lines connecting each outport of the virtual source to the model inport that it controls. Alternatively, you may hover over a controlled model inport to display the blue line connected to the virtual source that controls it.

### 9.3.5 Accessing Values from Previous Simulation Steps

Some Validator objectives may require access to values from previous simulation steps. For example, testing whether a block’s output has changed by more than a certain amount requires subtracting the previous output value from the current output value. In diagram objectives, values from previous steps are accessed via Simulink's Unit Delay block. In expression objectives, prior values are accessed by using the `pre()` function.

The `pre()` function can take either one or two arguments. The first argument is an expression which will be evaluated in the context of a previous simulation step. For example, `pre(x+y)` will return the sum of the values held by `x` and `y` during the previous simulation step. The optional second argument to the `pre()` function specifies the number of simulation steps to go back, so `pre(x+y,2)` will return the sum of `x` and `y` from two steps ago, `pre(x+y,3)` will return the sum of `x` and `y` from three steps ago, and so on.

If the `pre()` function is used to access a value prior to the first simulation step (e.g., when `pre()` is called during the first simulation step), the returned value will be zero. If this produces undesirable results, an if-then-else clause should be used to avoid invoking `pre()` until a sufficient number of simulation steps have been taken.

For example, to create an assertion that checks whether a variable `x` never changes by more than 0.5 per simulation step, use the following expression:

\[
\text{if } t==0 \text{ then } 1 \text{ else } \text{abs}(x-\text{pre}(x))<=0.5
\]

This expression returns true on the first simulation step. On all subsequent steps, it returns true if the difference between the current and previous values of `x` does not exceed 0.5.

### 9.3.6 Wiring Validator Objectives Within the Reactis Main Panel

![Figure 9.9: Connecting the output of “Relational Operator1” to the “active” input of Validator objective “SpdCheck”.

Selecting the proper wiring in the Validator objective property dialogs shown in the previous sections can sometimes be difficult, especially if the blocks in a system do not have descriptive names. As an alternative, you can wire up objectives within Simulink systems via
drag-and-drop in the main panel. When creating a new objective, just leave the entries in the inputs wiring table unconnected when dismissing the dialog. For expression objectives, you do not need to manually add the variables to the wiring table. Reactis will automatically add any missing variables.

After creating an objective and clicking “Ok” in the dialog, you can now use drag-and-drop to connect the inputs of the objective:

1. Left-click on a signal line or a block with only one output port and hold the mouse button down.

2. Reactis will draw a dashed line from the output port of the block that feeds the signal line to the current mouse position.

3. While holding the button, move the mouse cursor to the Validator objective to which you want to connect the signal.

4. While over the objective, release the mouse button. Reactis will select the objective and show a menu listing all of its inputs.

5. Select the desired input in the menu. This sets the wiring. Note that if some other signal was connected to the input, it will be replaced with the new connection.

6. You can inspect the proper wiring by hovering over the objective.

### 9.4 Linking to Requirements Documents

As described above, Validator helps you check if a model satisfies its requirements. Often, requirements for a system are specified using natural language in a requirements document. For requirements documents implemented in Microsoft Word or Microsoft Excel, Reactis offers a facility to establish and manage links between a natural language requirement and Reactis Validator objectives (assertions and user-defined targets) that check for violations of the requirement. This section describes that mechanism for linking to requirements documents.

For each assertion or user-defined target that you insert into a model, you can also establish a link which points to a particular location within a requirements document. Once this link is configured for an objective, you can right-click on the objective in the main Reactis panel and select Go to linked document to see the natural language requirement from which the objective was derived. If you have configured a bidirectional link, you can also go in the other direction from the natural language requirement to Validator objectives which check for compliance of the model to the requirement.

Each dialog for inserting or modifying a Validator objective includes a tab named DocumentLink, as shown in Figure 9.10. This example shows a link between the assertion LowSpeedInactive and a natural language requirement located at a bookmark named LowSpeedInactive in the Microsoft Word document cruise_requirements.docx. This link can be established with the following steps:

1. Right-click in the main Reactis panel and select Add Assertion → Expression and in the Settings tab of the resulting dialog, specify the assertion. Alternatively, a link to a requirement may be added to an existing Validator objective by right-clicking on the objective and selecting Edit Properties....
2. Select the Document Link tab.

3. Select the Microsoft Word radio button to indicate the requirement is in a Word docu-
ment.

4. Click the file selection button to the right of the Document text entry box.

5. In the resulting file selection dialog, select cruise_requirements.docx.

6. In the Location in Document section of the dialog, select radio button Bookmark and
enter the bookmark name LowSpeedInactive. Note, that in this scenario, this bookmark
did not yet exist in the Word document. If a bookmark at the location to which you wish
to link already exists, you can skip the next 3 steps which create a new bookmark in the
Word document.

7. Open cruise_requirements.docx in Microsoft Word.

8. Select the text that specifies the requirement to which you want to link.

9. Return to the Reactis Document Link dialog and click the second Create button to create
a bookmark to the selected text and establish a bidirectional link. This button click does
the following:

- Creates a new bookmark in cruise_requirements.docx named LowSpeedInactive.
- Inserts a red R icon at the bookmark. In Word if you ctrl+click on the R, the
LowSpeedInactive assertion will flash in yellow in the main Reactis panel. This is a
forward link from the natural language requirement to the assertion in the model. Note, that the model must be open in Reactis for this highlighting to occur.
- Establishes the reverse link from the assertion in the model to the bookmark in the
Word document. Subsequently, clicking the Go to link button in the dialog will
cause Word to open cruise_requirements.docx and display the new bookmark.

10. Click the Ok button to save the changes to the document link and dismiss the dialog.

The different elements of the Document Link tab are labeled in Figure 9.11 and work as
follows:

1. The Document section lets you specify the document to which you wish to link. First
use the radio buttons to select the program that manages the requirements document
(Microsoft Word or Excel), then specify the document.

2. When linking to a Word document, the document name goes here.

3. When linking to a Word document, this button can be clicked to open a file-selec-
tion dialog to chose a Word document.

4. When linking to an Excel document, the document name goes here.

5. When linking to an Excel document, this button can be clicked to open a file-selec-
tion dialog to chose an Excel document.

6. The Location in Document section lets you specify a location within a requirements doc-
ument. Two alternative mechanisms for specifying a location within a document are
offered:
Figure 9.10: The dialog for establishing and maintaining a link between a Validator objective and a natural language requirement. This example shows a link to a bookmark in a Microsoft Word document.

- Bookmarks (called named ranges in Excel).
- A search for a match against a specified text string.

The radio selector lets you choose the mechanism.

7. The name of a bookmark (named range) to which to link goes here. If a bookmark already exists in your document, you can simply specify the name here. If you wish to create a new bookmark, you can use buttons 8 and 9.

8. Clicking this button creates a bookmark (named range) in the Word (Excel) document that is specified in the Document section. The location of the bookmark is the currently selected text (for Word) or range of cells (for Excel). If a bookmark name is given in field 7, then that will be the name of the new bookmark. A link from the current Validator objective to the new bookmark will be established such that clicking the Go to link button (item 12) will cause the document to be opened and positioned to the new bookmark. If no name is specified in field 7, then a name will be auto-generated.
9. Clicking this button does everything that is done when button 8 is clicked and also inserts a link from the new bookmark to the Validator objective. This link appears as a red R in the document and when you ctrl+click on it in Word/Excel, the linked Validator objective flashes yellow in the main Reactis panel. Note that the model must be open in Reactis for this highlighting to work.

10. When finding a location within the document using a text search, the search string goes here.

11. If searching in an Excel document, you must also specify the Worksheet in which to search. Specify that here.

12. Clicking this button opens the document specified in the Document section to the location specified with the Location in Document section.

---

9.5 Running Reactis Validator

After adding assertions and user-defined targets to a model, Validator may be invoked by selecting the Validate → Check Assertions... menu item to determine whether or not assertion violations can be found. Figure 9.12 contains an annotated screen shot of the Validator launch dialog; the labeled items are described in the next subsection. It should first be noted, however, that the Validator launch dialog is very similar to that of the Tester launch dialog as described in Section 8.1. This similarity is due to the fact that conceptually, Validator works by generating test data using the Tester test-generation algorithm and then running the tests on the instrumented model to check for assertion violations. For this reason, many of the features of the Validator launch screen are identical to those of Tester. When this is the case, the descriptions below are very brief; more detail may be found in Section 8.1.

9.5.1 Labeled Window Items

1. Specify how long Validator should run. There are three options to choose from:
   - A fixed amount of time
   - A fixed number of steps
   - A specified number of random and targeted tests/steps.

   If 100% coverage of all targets is reached prior to the specified run time, Validator will terminate early. Note that for calculating this early termination, assertions are considered uncovered if they have not been violated. Therefore, if any assertion remains not violated, then early termination will not occur.

2. A list of .rst files containing test suites to be preloaded.

3. When the Prune check-box to the right of a filename is checked, unnecessary steps (those that do not increase the level of coverage) will be pruned from the preloaded test suite. When the check-box is not checked, no pruning of the suite will occur. Note, that assertions are checked for all steps of the preloaded tests. If a violation is found in a step, it will not be pruned.
4. If the Use Virtual Sources check-box is checked for an .rst file, then when executing the tests in the preloaded suite, values produced by enabled virtual sources will be used for controlled inputs instead of the input values from the test suite for those inputs. When this item is not checked, input values from the test suite will be used for all inputs.

5. Clicking this button invokes a file-selection dialog that enables the user to specify an .rst file to be added to the preload list.

6. Clicking this button removes the currently selected .rst file from the list of test suites to be preloaded.

7. When running Validator for a fixed length of time, the number of hours and minutes is entered here.
8. When running Validator for a fixed number of steps, the number of steps is entered here. Validator will decide how many of these steps will be random or targeted. Because of pruning, the number of steps in the final test suite will typically be less than the number entered here.

9. When running Validator for a specified number of random and targeted steps, the number of tests in the random phase is entered here. Because of pruning that occurs at the end of the random phase, some tests may be eliminated entirely, leading to a smaller number of tests at the end of the random phase than what is specified here.

10. When running Validator for a specified number of random and targeted steps, the number of steps to take while constructing each test of the random phase is entered here. Upon completion of the random phase, unimportant steps are pruned from the tests, so the lengths of the final tests will usually be shorter than the length specified here.

NOTE: Specifying too many steps in the random phase can cause Reactis to run out of memory. The upper bound on the number of steps possible depends on model size and available RAM, but in general much more time should be spent in the targeted phase which is more optimized for memory usage.

11. When running Validator for a specified number of random and targeted steps, the number of execution steps to take during the targeted phase is entered here. The targeted phase uses sophisticated strategies to guide the simulation to exercise parts of the model
not visited during the preload or random phases. The value entered specifies an upper bound on the number of simulation steps executed during the targeted phase.

12. This entry box enables the user to pass one or more of the following parameters to Validator:

- `-a1` turns *inputs abstraction* on, `-a0` turns inputs abstraction off. Inputs abstraction usually improves the performance of Validator and should be left on (default). In rare cases, turning it off may improve coverage. If coverage problems are encountered with inputs abstraction on, it may be beneficial to take a test suite produced with abstraction on, preload it into Validator, turn abstraction off, and then run Validator again.

- `-c n` sets the maximum number of input variables that may change during an execution step to `n`, which must be a positive integer. The default is that every input variable can change at every step. Restricting the number of input variables that can change can lead to easier-to-understand test suites.

- `-C n` directs Reactis to use `n` cores during test generation. Currently supported values for `n` are 1 and 2. Leveraging multi-core architectures speeds up test-generation for many models.

- `-s randomSeed` seed for the random number generator. This is useful for replaying a previous run of Validator. The random seed used to create a `.rst` file can be found in the test-suite log (which may be viewed in the Test Suite Browser described in Chapter 11), after the “-s” in the “Created by Tester:” line.

13. The name of the `.rst` file to be generated.

14. Clicking this button opens a file-selection dialog for specifying the name of the `.rst` file to be generated.

15. Clicking this button displays Validator help.

16. Clicking this button opens a file selection dialog to specify an `.rtp` file from which to load Validator launch parameters. Reactis may be configured from the Settings dialog to generate an `.rtp` file for each Tester or Validator run.

17. Reset the Validator parameters to their default values.

18. Scroll backward in the parameter history.

19. Scroll forward in the parameter history.

20. Clicking this button starts Validator run.

21. Clicking this button closes the Validator window.

The Progress Dialog displayed while Validator is running is the same as the Progress Dialog for Tester. For more information see Section 8.2.
9.6 Validator Menus in the Reactis Top-Level Window

Chapters 4 and 7 describe most of the menu entries of the menu bar in the Reactis Top-Level Window. We now describe the entries which are related to Validator.

**Edit menu.** This menu includes entries used to manipulate Validator objectives which are stored in `.rsi` files. Note, that `.rsi` files may be modified only when Simulator is disabled, therefore, when Simulator is active, these items are disabled.

*Undo.* Undo an operation (add, edit, remove, move) on a Validator objective.

*Redo.* Redo last undone operation (add, edit, remove, move) on a Validator objective.

*Cut.* Cut the currently selected Validator objective.

*Copy.* Copy the currently selected Validator objective to the clipboard.

*Paste.* Paste a Validator objective from the clipboard to the current subsystem. To paste an objective to a specific position, right-click on that position in your model and select *Paste* from the context menu.

**Validate menu.** The menu items include:

*Add Assertion.* Add a new Validator assertion at a default location of the currently selected subsystem.

*Add User-Defined Target.* Add a new Validator target at a default location of the currently selected subsystem.

*Add Virtual Source.* Add a new virtual source at a default location of the currently selected subsystem.

*Edit Objective.* Edit the currently selected objective.

*Remove Objective.* Remove the currently selected objective.

*Enable/Disable Objective.* Enable or Disable the objective.

*Check Assertions.* Start a search for tests that violate assertions or cover user-defined targets.

9.7 Tracking Coverage Within Validator Objectives

Reactis tracks any coverage targets located within diagram objectives, but does not track any targets within expression objectives. Since a diagram objective is implemented as a Simulink / Stateflow model it can contain any of the coverage targets that Reactis tracks (e.g. branches, decisions, MC/DC targets, etc.). By default Reactis will treat these targets exactly as it treats targets within your model, i.e. they will be included in all coverage reporting and Reactis Tester will try to exercise them. By examining which coverage of targets within an objective were exercised you can determine how close testing came to violating an assertion or covering a UDT.

However, in some cases you might want to turn off this tracking of targets within objectives. For example, when generating a final report, you may prefer to see the only the statistics for covered targets within your model. You can turn off this tracking from the Coverage Metrics pane of the Info File Editor with the setting named *Track coverage for contents of Validator Objectives.*
Chapter 10

The Reactis Coverage-Report Browser

The Coverage-Report Browser enables you to view detailed coverage information related to testing and to export reports in HTML format for viewing and printing within a web browser. Figure 10.1 contains an annotated screen shot of the Coverage-Report Browser. The title bar includes information describing the test suite, tests, and test steps that were exercised before computing the coverage information. Here, the report describes coverage attained after executing steps 1 through 80 of test 4 in test suite `cruise.rst`. When a report reflects the coverage attained by executing an entire test suite, only the test suite name appears in the title bar.

![Figure 10.1: The Coverage-Report Browser.](image)

199
10.1 Labeled Window Items

1. The model hierarchy panel works much as the hierarchy panel does in the top-level Reactis window. Namely, it displays the subsystems that make up the model and how they are related. Clicking on a name causes coverage information for the subsystem to be displayed in the panels to the right. Double-clicking on a name causes the subsystem diagram to be displayed in the Simulator main panel.

2. This panel displays summary coverage statistics for the subsystem currently selected in the hierarchy panel.

3. This panel shows coverage information for each child subsystem of the subsystem currently selected in the hierarchy panel. The information is presented as a matrix and includes a row for each child. The first column includes the names of the children, while each subsequent column includes statistics for one of the coverage measures tracked by Reactis. In Figure 10.1, the entry 

\[(6)50\%\]

indicates that child subsystem *DesiredSpeed* has 6 branches, 50\% of were covered by steps 1 through 80 of test 4. Double-clicking on a row causes the child to become the current subsystem displayed in the Coverage-Report Browser.

The rows of the table may be sorted based on the values of any column. To cause the values in a particular column to order the rows, simply click on the header of the column. Clicking once sorts in increasing order of the column entries, while clicking a second time sorts in decreasing order.

4. This panel displays which specific targets within the currently selected subsystem are covered, uncovered or unreachable. The way this information is conveyed differs between coverage metrics:

**Decision, Condition, MC/DC** For targets in the MC/DC-related metrics (decision, condition, MC/DC) the list includes one entry of type MC/DC for each decision. The status column shows covered when all decision, condition and MC/DC targets associated with the decision are covered. Right clicking on such an entry will activate a pop-up menu that includes the menu item *Show Details*. When this menu item is selected a dialog conveying coverage information similar to that shown in Figure 7.16 will appear. See Section 7.6.2 for a description of this dialog.

**MCC** The list contains one entry of type “MCC” for each decision in the subsystem. The status column of that entry shows the percentage of MCC targets covered for that decision. Right clicking on such an entry will activate a pop-up menu that includes the menu item *Show Details*. When this menu item is selected a dialog conveying coverage information similar to that shown in Figure 7.17 will appear. See Section 7.6.2 for a description of this dialog.

**Lookup Table** The list contains one entry of type “Lookup” for each lookup table in the subsystem. The status column of that entry shows the percentage of targets covered within that table. Right clicking on such an entry will activate a pop-up menu that includes the menu item *Show Details*. When this menu item is selected a dialog conveying coverage information similar to that shown in Figure 6.10 or Figure 6.11 will appear. See Section 6.1.3 for a description of this dialog.
For all other targets in the subsystem, this panel displays “covered”, “uncovered” or “unreachable” depending on the coverage status of the target.

The rows of the table may be sorted based on the values of any column. To cause the values in a particular column to order the rows, simply click on the header of the column. Clicking once sorts in increasing order of the column entries, while clicking a second time sorts in decreasing order.

### 10.2 Menus

**Report menu.** The Report menu contains the following entries.

*Export...* Invoke the dialog shown in Figure 10.2 that enables the user to export the coverage report in HTML format. See Section 10.3 for details.

*Exit.* Exit the Coverage-Report Browser.

**Help menu.** The Help menu contains the following entries.

*Contents.* Go to the table of contents in the documentation.

*Index.* Go to the index in the documentation.


### 10.3 Exporting Coverage Reports

The Coverage-Report Browser includes a facility that enables you to export coverage reports in HTML format. The resulting files may be viewed and printed using facilities in a web browser. The dialog shown in Figure 10.2 is invoked by selecting the Report → Export... menu item from the Coverage-Report Browser menu bar. Coverage details in the exported reports are organized so that each subsystem of the model has a section in the report; a subsystem’s section includes coverage information for the targets in that subsystem.

The labeled items in the dialog are used as follows.

1. The radio button selected from this group specifies the portion of the model which will be included in the exported report.

2. When this check box is selected, a table of contents containing a link to each section will be included at the top the exported report.

3. When this check box is selected, the date the report was generated is included at the top of the report.

4. When this check box is selected, the full operating system paths to the relevant .slx file and .rst file are included at the top of the report. The .slx file contains the model from which the report was generated. If the report conveys statistics after tests from a test suite have been executed, then the .rst file named is the file in which these tests reside.

5. The **Coverage Export Options** panel can be used to select which coverage metrics are included in the coverage report. There are three choices for each metric:
Summary & Details. Targets of the metric will appear in both the coverage summary and coverage details sections of the report.

Summary Only. Targets of the metric will appear in the coverage summary only.

None. Targets of the metric will be omitted from the report entirely.

Note that due to dependencies between metrics, some combinations are not allowed. For example, Summary & Details cannot be selected for Condition targets unless Summary & Details is also selected for Decision targets.

6. Click this button to display help for the coverage export.

7. Click this button to preview the report to be exported.

8. Click this button to export the report. A file selection dialog will appear to allow you to name the file where the report should be written.

9. Click this button to abort report generation.
10.4 Coverage Report Contents

Based on the selections in the Export to HTML dialog, a report will be generated with different elements. An example report is shown in Figure 10.3 that was generated using the export options shown in Figure 10.2. For each subsystem of the model, the report will include a summary section (coverage statistics for the subsystem) and details section (detailed information about the coverage of each target in the subsystem). The Export to HTML dialog lets you specify which coverage metrics should be included in the summary and detail sections.

When using the Reactis for C Plugin or the Reactis for EML Plugin, Reactis tracks coverage targets within the C or EML code incorporated into a model. In this case, HTML coverage reports will include source code annotated to convey coverage information as shown in Figure 10.4. Lines with uncovered statements are printed in red. A yellow status column to the left of the code and to the right of the line numbers displays additional coverage information.
The left portion of the status column gives the coverage status of each statement in the line. Each statement in the line will have an indicator that is either a red S (meaning uncovered statement) or test/step giving the test and step in which the statement was first exercised. If a line contains more than one statement, then a list of indicators will be separated by commas. A blue indicator S:\texttt{excluded} means the statement has been excluded from coverage tracking.

The right portion of the status column gives the coverage status for the decision-related targets in the line. The aggregate indicator takes the form D:xxx. The D symbol is used to convey that the information relates to a decision. The first position after the colon represents decision coverage, the second condition coverage, and the third MC/DC. Each \( x \) can be:

- One or more target is not covered.
- All targets are covered.
- One or more targets are excluded and all non-excluded targets are covered.

Each aggregate decision indicator is a hyperlink that, when clicked, takes you to the details table for the decision.

The indicators in Figure 10.4 can be interpreted as follows:

**Line 20** \ D:--- means at least one decision, one condition, and one MC/DC target remain uncovered.

**Line 16** \ D:+++ means all decision, condition and MC/DC targets have been covered.
Line 18  $D:++-$ means both decision targets have been covered, but at least one condition and one MC/DC target remain uncovered.
Chapter 11

The Reactis Test-Suite Browser

The Test-Suite Browser allows you to view test suites. Figure 11.1 contains an annotated screenshot of the Test-Suite Browser, is used to examine the contents of a test suite. The body of the window displays the data for the currently selected test, and consists of three tabs:

**Test Data** Displays a matrix whose first column lists the input ports, test points, and output ports of the model and whose subsequent columns each represent a simulation step. Within each of these latter columns there is a value for each inport, test point, and out-port.

**Test History** Displays history information logged by Reactis for the currently selected test.

**Suite History** Displays history information logged by Reactis for the test suite.

11.1 Labeled Window Items

The following items are labeled in Figure 11.1:

1. Display a file-selection dialog to specify an `.rst` file for loading into the browser.

2. Open a *distribution scope* to display the distribution of values the selected port assumes in the test suite, as shown in Figure 11.3. Alternatively, it is possible to view the values assumed during a single test or group of tests, rather than during the entire test suite. This may be done by right-clicking on the selected port or by using the relevant items in the *View* menu; see Section 11.2 for details. The button is disabled when a filter is active.

3. Open a scope for the selected port(s). The scope will be similar to the one shown in Figure 11.2.

4. If either of the Test History or Suite History tabs are selected, copy any of the selected log information to the clipboard.

5. Go back 10 steps in the current test.

6. Go back 5 steps in the current test.

7. Go back 1 step in the current test.

8. Specify a step in the current test to view.
9. Go forward 1 step in the current test.

10. Go forward 5 steps in the current test.

11. Go forward 10 steps in the current test.


13. This pull-down menu enables you to select a test from the currently loaded test suite to view in the browser. The face of the button shows a message of the form #tnum (tsteps): tname where tnum is the test number, tsteps is the number of steps in the test, and tname is the test name. When a filter is active, the text in parentheses changes to matching of tsteps where matching is the number of steps in the test that match the filter.

14. Checking this box activates the filter. When a filter is active only steps for which the filter expression evaluates to true will be displayed.

15. A filter expression may be entered in this text entry box. When a filter is active, the main browser panel shows only test steps for which the expression evaluates to a non-zero value. You activate a filter by:

   - pressing the return key in the entry box, or
   - checking the check box to the left of the entry box, or
   - clicking the enter button to the right of the entry box.
For example, to display the steps in which the brake is active enter \texttt{brake == 1} and press return.

The syntax for filter expressions is ANSI C. The variables in the expression may be the names of inports, outports, or test points from the test suite. The elements of a vector item in the test suite can be accessed using the array indexing notation of the C language (note that indexing is 0-based). The elements of a bus may be accessed using the C structure notation, e.g. \texttt{X.Y} if \texttt{Y} is an element of bus \texttt{X}. Some examples demonstrate the most common operators:

<table>
<thead>
<tr>
<th>Filter Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(brake == 1) &amp;&amp; (active == 1)</td>
<td>All steps in which the brake is pressed and the cruise control remains active (indicating a serious error)</td>
</tr>
<tr>
<td>(speed &lt; 30) &amp;&amp; (active == 1)</td>
<td>All steps in which cruise is active at a speed less than 30</td>
</tr>
<tr>
<td>(cancel == 1)</td>
<td></td>
</tr>
<tr>
<td>(inp[0] &gt; 0) &amp;&amp; (inp[1] &gt; 0)</td>
<td>All steps in which the first two elements of vector input \texttt{inp} are positive</td>
</tr>
</tbody>
</table>

16. Clicking this button activates the filter. When a filter is active only steps for which the filter expression evaluates to true will be displayed.

17. Clicking this button opens the Reactis Test Step Filter Editor dialog shown in Figure 11.4. The editor lets you formulate more complicated filters.

18. This column lists the inports, test points, and outports of the model.

19. Values of inports, test points, and outports in step 1 of the current test.

20. Values of inports, test points, and outports in step 2 of the current test.

21. This column lists the name of each configuration variable.

22. This column lists the value of each configuration variable in the current test.

Figure 11.2: A scope which has been opened from within the test-suite browser. Any test in the suite can be selected for viewing.
11.2 Menus

File menu. The File menu contains the following entries:

- **Open...** Launches a file-selection dialog for choosing a new .rst file to browse.
- **Close.** Close the current .rst file.
- **Exit.** Exit the Test-Suite Browser.

Edit menu. The Edit menu contains the following entries:

- **Copy.** If either of the Test History or Suite History tabs are selected, copy any of the selected log information to the clipboard.
- **Select All.** Select all log information in the currently displayed tab.

View menu. The View menu contains the following entries:

- **10 Steps Back.** Go back 10 steps in the current test.
- **5 Steps Back.** Go back 5 steps in the current test.
- **1 Step Back.** Go back 1 step in the current test.
- **Go to Step...** Specify a step in the current test to view.
- **1 Step Forward.** Go forward 1 step in the current test.
- **5 Steps Forward.** Go forward 5 steps in the current test.
- **10 Steps Forward.** Go forward 10 steps in the current test.
- **Significant Digits (Current Port)...** Specify how many significant digits should be used when displaying the values flowing over the currently selected port. If ’-1’ is specified then the default number of significant digits will be displayed (16 for doubles and 8 for singles).
- **Significant Digits (All Items)...** Specify how many significant digits should be used when displaying the values flowing over all ports in the model. If ’-1’ is specified then the default number of significant digits will be displayed (16 for doubles and 8 for singles).
- **Visible Steps...** Set the number of test steps to be shown in the main panel of the browser window.
- **Open Distribution Scope (Whole Suite).** For the currently selected port, open a distribution scope to display (as shown in Figure 11.3) the values the port assumes when executing the test suite.
- **Open Distribution Scope (Current Test).** For the currently selected port, open a distribution scope to display the values the port assumes when executing the currently selected test.
- **Open Distribution Scope (Selected Tests)...** Open a dialog to select a subset of tests from the current test suite; then, for the currently selected port, open a distribution scope to display the values the port assumes when executing the tests in the selected subset. The dialog for selecting a subset of tests works as follows. Holding down the control key and clicking on different tests enables you to select a group of tests. For example test 1, test 3, and test 5 may be selected by holding down the control key...
and clicking on each of those three tests. Alternatively, holding down the shift key and clicking on two different tests will add the two tests along with all tests between them to the group. This approach may be used to select tests 1-4 by holding down the shift key and clicking on test 1 and then test 4.

Open Scope (Selected Rows) Open a scope to display the currently selected row(s).

Open Scope (Signal Group). Open a scope for the specified signal group. A signal group is created by clicking the save button (.snapshot) in a scope to save the current configuration of the scope as a signal group (set of signals along with the scope settings for displaying them).

Delete Signal Group. Delete the selected signal group.

Filter menu. The filter menu lets you display only a subset of rows and columns in the main panel.

Edit Column Filter... Open the Test Step Filter editor shown in Figure 11.4 and described in the subsequent text. The editor lets you create a filter to identify test steps in the test suite that satisfy a particular condition. For example, all test steps in which the cruise control is active.

Enable Column Filter. This menu entry toggles whether or not the column filter is active. When active, only steps satisfying the filter expression are displayed in the main panel.

Show Hidden Columns. You can toggle this entry to cause Reactis to display steps for which the filter expression is false as well as those for which it is true. Steps for which the filter expression evaluates to false will have their values and step number displayed within parentheses.

Hide Selected Rows. Hide the currently selected rows. Note you may select a range of rows by left-clicking on a row, then holding down the shift key and left-clicking on a second row. All rows from the first row selected to the second row selected (inclusively) will be selected. Alternatively, a group of rows can be selected by holding down the control key and left-clicking on each row to be included in the group.
**Unhide Selected Rows.** Unhide the currently selected rows. Note, to select hidden rows you must first toggle the *Show Hidden Rows* menu entry.

**Show Hidden Rows.** Display hidden rows in the main panel. Note, in this mode, when a row is hidden, the item name and all values will be grayed out.

**Help menu.** The Help menu contains the following entries:

- *Contents.* Go to the table of contents in the documentation.
- *Index.* Go to the index in the on-line documentation.

### 11.3 Test Step Filter Editor

The **Test Step Filter Editor**, shown in Figure 11.4, lets you formulate filters to search for test steps in a suite that satisfy a given condition.

![Test Step Filter Editor](image)

Figure 11.4: The Reactis Test Step Filter Editor lets you craft more complex filters to search test suites for steps satisfying a particular condition.

The labeled items in the figure work as follows.

1. Enable or disable the filter.
2. Enter a filter expression in this entry box. When a filter is active, the Test Suite Browser displays only test steps for which the expression evaluates to a non-zero value. The syntax for the expression is ANSI C. The variables in the expression take their values from inports, outports, or test points from the test suite. The elements of a vector item in the test suite can be accessed using the array indexing notation of the C language (note that indexing is 0-based). Some examples demonstrate the most common operators:
Filter Expression | Result
---|---
(brake == 1) && (active == 1) | All steps in which the brake is pressed and the cruise control remains active (indicating a serious error)
(speed < 30) && (active == 1) | All steps in which cruise is active at a speed less than 30
(cancel == 1) || (decelSet == 1) | All steps in which either cancel or decelSet is pressed
(inp[0] > 0) && (inp[1] > 0) | All steps in which the first two elements of vector input inp are positive

3. The variables used in the expression are defined in this section. They are termed inputs because the filter expression can be viewed as the body of a function and the inputs are the formal input parameters of the function. To apply the filter function to a given test step, values from the test step serve as actual parameters fed into the filter function for evaluation of the filter expression. The elements of a vector item in the test suite can be accessed using the array indexing notation of the C language (note indexing is 0-based). For example, if an input inp is a vector of length 3, then we can check for steps in which elements of inp are positive with the filter expression:

\[(\text{inp}[0] > 0) \&\& (\text{inp}[1] > 0) \&\& (\text{inp}[2] > 0)\]

The elements of a bus may be accessed using the C structure notation, e.g. \(X.Y\) if Y is an element of bus X.

4. Each pulldown in this column includes an entry for each input, output, test point, and sample time in the test suite. The item selected will be fed into the input named in the next column when evaluating the filter expression.

5. Each entry box in this column names an input/variable to be used in the filter expression.

6. Clicking this button deletes the filter input.

7. Clicking this button adds a new filter input.

8. Open help for the Filter Editor.

9. Clicking this button causes Reactis to attempt to automatically compute the entries in the inputs section. For each variable in the filter expression an input row will be added if it does not already exist. Furthermore the input will be wired (pulldown in the first column selected) to the input, output, or test point of the same name if one exists in the test suite.

10. Save the changes made in the Filter Editor and close the dialog.

11. Discard the changes made in the Filter Editor and close the dialog.
Chapter 12

Executing Tester-Generated Test Suites

Once created with Reactis, test suites may be executed in a variety of different ways. You may:

1. Load them into Reactis Simulator and run them.

2. Load them into the Simulink simulation environment of The MathWorks and run them, with or without the help of the graphical rsRunTests tool included in the Reactis distribution.

3. Export them as CSV files for processing by other tools.

This chapter describes these options in more detail.

12.1 Executing Test Suites Using Reactis Simulator

Reactis Simulator includes facilities for running Tester-generated tests on Simulink / Stateflow models. These were detailed in Chapter 7. Here we briefly outline several of the most common use cases for executing tests in Simulator. Note that in all cases, Reactis will check the values stored in test suites for outputs and test points against those computed by the model. Also recall from Chapter 7, that Simulator offers many different ways to track data item values and coverage as tests execute.

![Reactis Simulator toolbar](image)

Figure 12.1: The Reactis Simulator toolbar.

The following uses Figure 12.1 to explain the steps needed to load and run a .rst file named cruise.rst.

1. Select File → Open Model... and use the resulting dialog to select cruise.slx for loading.

215
2. Click button 2 to enable Simulator.

3. Click button 13 and use the resulting dialog to select `cruise.rst` for loading. The name of the file will be shown in the title bar of the Simulator window, and the tests will be listed in Source-of-Inputs dialog (window item 11).

4. Once a test suite is loaded, some possible ways of running tests include:

   - **To run a single test:**
     - (a) Click on the Source-of-Inputs dialog button (window item 11).
     - (b) Select the test to be run. It will appear in the Source-of-Inputs dialog button.
     - (c) Use buttons 3–9 to step through the test as described in Section 7.1. The inputs for each simulation step will be taken from the test and the values computed for test points and top-level outputs will be compared against those stored in the test suite.

   - **To run all tests, one after another:**
     - (a) Click on the Source-of-Inputs dialog button (window item 11).
     - (b) Click the **All** button at the bottom left of the dialog.
     - (c) Use buttons 3–9 to run the tests as described in Section 7.1. The inputs for each simulation step will be drawn from the tests (beginning with Test 1) and the values computed for test points and top-level outputs will be compared against those stored in the test suite. When each test is completed, the simulation is “reset” so that the next test begins in the initial state of the model. Coverage information, however, is not reset, so the final cumulative statistics show the set of targets covered by any test in the suite.

   - **To run all tests and generate an HTML report:**
     - (a) Click on the Source-of-Inputs dialog button (window item 11).
     - (b) Click the **All** button at the bottom left of the dialog.
     - (c) Select **Simulate** → **Fast Run with Report**... The resulting report lists any runtime errors uncovered (divide-by-zero, overflow, memory errors, missing cases, assertion violations, etc.) or differences between outport values stored in the test suite and those computed by the model. If a runtime error is encountered, the remaining steps of the current test are skipped and Simulator continues execution with the following test. Optionally, you can also direct Reactis to include detailed coverage information (if and when each target was covered) in the report.

---

### 12.2 Executing Test Suites in Simulink

Test suites generated by Reactis may also be used to guide simulations in The MathWorks’ Simulink environment. To run tests in Simulink, a test suite must first be exported into a format readable by MATLAB. Test suites may be exported from Reactis Simulator (described in more detail in Chapter 7) by selecting the **Test Suite** → **Export**... menu item. For running tests in Simulink, Reactis supports three export formats which are listed in the export dialog as follows.
12.2. EXECUTING TEST SUITES IN SIMULINK

1. **Matlab script (.m)**
2. **Matlab binary data (.mat)**
3. **Matlab binary data for FromWorkspace blocks (.mat)**

In all cases, when executed, the file initializes matrices that can be used as inputs to drive a Simulink simulation. We describe four ways to load an exported test suite into Simulink and then read input values from the tests during a simulation. The four methods are listed below along with the export file formats with which they work.

<table>
<thead>
<tr>
<th>Method</th>
<th>Compatible File Formats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using the rsRunTests utility</td>
<td>1, 2</td>
</tr>
<tr>
<td>Using the Simulation Parameters</td>
<td>1, 2</td>
</tr>
<tr>
<td>Editing the .m file</td>
<td>1</td>
</tr>
<tr>
<td>Replacing top-level inports with</td>
<td>3</td>
</tr>
<tr>
<td>FromWorkspace blocks</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** When executing Tester-generated test suites in Simulink, *interpolation must be turned off for all top-level inports of the model*. You may turn interpolation off for top-level ports in one of two ways:

1. By using the `interpolateoff` utility distributed with Reactis. The utility is invoked from the MATLAB command line. If given a filename as an argument, `interpolateoff` will open the specified model file and set the inports to not interpolate. When invoked with no argument the current model is modified. If the model does not have any top-level input ports or all top-level input ports are already set to non-interpolate, the script does nothing. To make the changes permanent, the model must be saved (to a different file if you wish to retain the old settings).

2. By right-clicking on each port in a Simulink window, selecting block parameters, and making sure the interpolate check box is not checked.

### 12.2.1 Executing Tests in Simulink with the rsRunTests Utility

The `rsRunTests` tool in the Reactis distribution can be used as follows to run Reactis-generated tests from within Simulink:

1. Ensure that the Reactis MATLAB API folder has been added to the MATLAB path. This can be done by clicking the Set Path button in the MATLAB toolbar or from the MATLAB command line with:

   ```
   >> addpath('C:\Program Files\Reactis V2020\lib\api\MATLAB\reactis')
   ```

2. Start the GUI by entering `rsRunTests` on the MATLAB command line. You will be presented with the window shown in Figure 12.2.

3. Use the Load Model button to specify a Simulink model to load.
4. You may now click the **Show Model** button to view the loaded model.

5. Use the **Load Test File** button to specify a .m file or a .mat file (exported by Reactis) to load.

6. Select a test in the drop-down box.

7. Click the **Run Test(s)** button to execute the selected test.

8. Click the **Save Log** button to save a copy of the contents of the **Test execution log** to a user-selected text file.

9. Any significant differences between the model’s outputs as computed by Reactis (and stored in the test suite) and those computed by Simulink will be listed in the log. A difference between outputs is listed if it exceeds the tolerance specified for the output. The **Tolerance** pull-down may be used to manipulate tolerances. The menu includes a Default entry as well as an entry for each top-level outport. If the setting for an outport is Inherit the Default tolerance is used.

   The initial values of the Tolerance pull-down are imported from the test suite (which was exported by Reactis). You may manipulate both the default and port-level tolerances for a model from the Reactis Info File Editor and Reactis will store these values in the .rsi file associated with the model. Reactis will then export the tolerances from the .rsi file to exported suites (.m or .mat files) for use by rsRunTests.

   The process used to flag differences is described in Section 5.6.
12.2. Executing Test Suites in Simulink

12.2.2 Executing Tests in Simulink with the Simulation Parameters Dialog

An alternative to the `rsRunTests` utility is running the tests manually within MATLAB as follows.

1. Load the `.m` file (or `.mat` file) by entering the file name without the `.m` (.mat) file suffix on the MATLAB command line. For example:

   ```matlab
   >> cruiseTests
   ```

2. Load the model in Simulink and select the Simulation → Model Configuration Parameters... menu item.

3. In the Solver pane of the Configuration Parameters dialog, adjust the simulation time to use the entire test. By convention, Reactis stores the start and stop times for test `n`, where `n` is the number of the test (1, 2, 3, etc.), in variables named `testn_start` and `testn_stop`, respectively.

4. In the Data Import/Export pane of the Configuration Parameters dialog, enable the Input check box and enter the name of the test matrix in the text box next to it. The test matrices in a Reactis-exported `.m` file (.mat file) are by convention named `test1`, `test2`, `test3`, ... For example, enter `test1` in the text box if you want to execute the first test.

5. Start a simulation run in Simulink. The simulation inputs will be taken from the specified test.

12.2.3 Running Tests in Simulink by Editing the `.m` file

This method also allows tests to be run from within MATLAB. The difference with the previous method is that it allows the “batch processing” of multiple tests.

1. Open the `.m` file in the MATLAB editor (by selecting File → Open... from the top-level MATLAB window).

2. In line 2 of the `.m` file make sure `modelName` is assigned the name of the model you would like to simulate.

3. Scroll to the bottom of the file and you will find a number of commented-out lines, one for each test. Uncomment the lines of tests you want to execute and save the file.

4. Load the file by entering its name on the MATLAB command line, and all uncommented tests will be executed.

12.2.4 Running Tests in Simulink by Replacing Top-Level Inports with FromWorkspace Blocks.

Executing tests exported in format 3 requires some modifications to be made to a model before the tests may be run in Simulink. For each test in a test suite, the exported file contains a variable named `testn` (where `n` is the test number). The variable stores an array of structures

---

1Note that if an `.m` file and an `.mdl` file have the same base name, then entering that base name on the MATLAB command line causes the `.slx` file to be loaded.
ready for use in a FromWorkspace block. Each element of the array stores the values that one
inport assumes during the test. A model may be modified to use these structures to execute
test \( n \) as follows:

1. Load the model in Simulink.

2. For each top-level inport,
   (a) Delete the inport and replace it with a FromWorkspace block.
   (b) Double-click on the new FromWorkspace block.
   (c) In the Data field, enter \texttt{testn(i)} (where \( n \) is the test number and \( i \) is the inport
       number. For example \texttt{test3(5)} specifies test 3, inport 5).
   (d) Uncheck the Interpolate field.
   (e) Select Holding final value in the Form output after final value by field.

3. In the Solver tab of the Simulation Parameters dialog, adjust the simulation time to use
   the entire test. By convention, Reactis stores the start and stop times for test \( n \), where \( n \)
is the test number, in variables named \texttt{testn_start} and \texttt{testn_stop}, respectively.

4. Start a simulation run in Simulink. The simulation inputs will be taken from the specified
test.

---

### 12.3 Executing Test Suites on Source Code

Tester-generated test suites may also be run on source-code implementations of Simulink /
Stateflow models; this feature helps automate the important task of determining whether
deployed code will conform to models. Reactis currently supports several methods for running
tests on source.

**Via Reactis Simulator (in Reactis for C Standalone)** The standalone version of Reactis for C
(not the Reactis for C Plugin) lets you test and validate standalone C code that is not a
part of a Simulink model. Reactis for C shares the same test suite format (.rst) with Re-
actis for Simulink which greatly simplifies back-to-back testing of C code (foo.c) against
a model (foo.slx). To do so:

1. Generate a test suite foo.rst from foo.slx with Reactis for Simulink.
2. Load foo.c in Reactis for C
3. Start Simulator (in Reactis for C) and load foo.rst
4. Select Simulate \rightarrow Fast Run with Report... to execute the tests on the C code. Out-
puts computed by the C code will be compared against those stored in the test suite
(which came from the model) and any differences are described in the generated re-
port.

**Via Reactis Simulator (in Reactis for Simulink)** You may use the S-function feature of Math-
Works’ Simulink environment to embed your source code inside a Simulink block. This
wrapper model may then be loaded and run in Simulator with differences between S-
function outputs and outputs stored in tests flagged. This is especially useful when using
the Reactis for C Plugin, since it allows inspection of the C code as the tests execute
and tracking the coverage of the C code.
### 12.3. EXECUTING TEST SUITES ON SOURCE CODE

On executing test suites on source code, one may use the S-function feature of MathWorks' Simulink environment to embed your source code inside a Simulink block. Then, as described in Section 12.2, a number of methods for executing tests are available, including the `rsRunTests` utility provided by Reactis which will execute the tests and automatically flag any differences between the values stored in the tests and those computed by Simulink.

Via user-written test harnesses: Reactis is also capable of outputting a test suite as a comma separated value (CSV) file for easy processing by external tools. Test steps are listed in an easy-to-process fashion. You may write your own test harnesses to read these files, run your program on the test data, and compare the outputs of the program to those stored in the tests.

Both the `.m` files and CSV files may be obtained using the Reactis Simulator export facility; see Section 7.7.1 for more on how to do this.

In order to write test harnesses that process Reactis-exported CSV files, one must know about the format of these files. Figure 12.3 contains a partial listing of a CSV file generated from the cruise-control example used in Chapter 3. See Section 7.7.1 for a description of the CSV file format.

#### CSV File Format

<table>
<thead>
<tr>
<th>Column 1</th>
<th>Column 2</th>
<th>Column 3</th>
<th>Column 4</th>
<th>Column 5</th>
<th>Column 6</th>
<th>Column 7</th>
<th>Column 8</th>
<th>Column 9</th>
<th>Column 10</th>
<th>Column 11</th>
<th>Column 12</th>
<th>Column 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>onOff</td>
<td>accelResume</td>
<td>cancel</td>
<td>decelSet</td>
<td>brake</td>
<td>inacThrDelt</td>
<td>drag</td>
<td><em><strong>t</strong></em></td>
<td>active</td>
<td>thrDelt</td>
<td>speed</td>
<td>InitialSpeed</td>
<td></td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>-0.1</td>
<td>-0.01</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.0989914797712953</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>-0.011152643712953</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>-0.00829318652470756</td>
<td>3.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Figure 12.3: A sample fragment of an exported test suite (CSV format).
Chapter 13

Maximizing Coverage

Reactis Tester and Validator work hard to generate tests that maximize covered targets. Nevertheless, you may encounter models that present challenges to Reactis: generated test suites may not provide adequate coverage. This chapter discusses strategies for overcoming such situations.

When attempting to improve coverage, the following general strategy is often useful:

1. Load the model in Reactis and start Simulator.
2. Load a test suite.
3. Make sure Coverage → Show Details is selected, and run all tests in the suite.
4. Inspect uncovered targets. Are unexercised targets coverable?

**No:** Perhaps the uncovered targets are in fact unreachable. The unreachable targets might be intended (for example, the upper or lower bound of a saturation block that you never expect to be exercised) or unintended due to a modeling error.

**Yes:** If you determine that the uncovered targets could be covered, then try applying one of the techniques described in the following sections to enhance the coverage of the test suite.

### 13.1 Constraining Input Values

In many cases coverage can be improved if tighter bounds are imposed on input values by inport constraints in the .rsi file. First, any range information for inputs should be entered. For example, for vehicle speed, rather than allow any double precision floating point value, we narrow the set of acceptable value to between 0 and 200. This greatly reduces the set of values for this input that Reactis must consider when generating tests. In some cases the allowable set of inputs can be reduced even further to a set of enumerated values. For example, a flag may be restricted to the set including only 0 and 1. In general, the goal is to reduce the number of possible values Reactis must consider when generating tests.

Probability weights may also be used to affect the likelihood with which different conditions become true. For example, in some models, large portions of the model become unreachable if an input flag has a certain value (e.g. this input might indicate the system being modeled should shut down). Assigning low probabilities to those values can increase coverage. Chapter 5 describes how to use the Reactis Info File Editor to set constraints for top-level inports.
When modifying the .rsi file in this fashion, it is good practice to save the modified .rsi file as a new file, so that the original settings are not overwritten. This can be done using the File → Save Info File As... menu item in the top-level Reactis menu, as described in Chapter 5. Note that this operation automatically associates the new .rsi file with the model. The link to the previous .rsi file can be restored using the File → Select Info File... menu item.

13.2 Modifying Test-Generation Parameters

Another approach for improving coverage is to change the parameters used by Tester and Validator. These parameters may be set in the launch screens of the respective tool and may be categorized as follows:

Preload Files. This parameter consists of a list of test suites that Tester / Validator should load initially and then try to extend. When a test suite provides insufficient coverage, a good strategy to try is to re-run Tester / Validator pre-loading the inadequate suite and instructing Reactis to try to enhance it.

Run for. Letting Tester run for a longer period of time can often increase coverage. For large models overnight or even multi-day runs might be helpful. For long runs, specifying a run time in the Run for section is usually easiest (as opposed to giving a number of steps to be taken).

13.3 Using Timer Targets

Certain model constructs that we term interruptible timers present a particular challenge to Reactis. These timers/counters require the application to hold some condition true for a given length of time or number of simulation steps. If the condition becomes false before the threshold is reached the timer/counter resets.

Reactis includes a mechanism called Timer Targets that enables you to identify those elements of your model which are timers or counters. When they are tagged, Reactis can often generate a test automatically to cause a timer/counter to tick up to its threshold. Please see Chapter 9 for details of how to instrument your model with a timer target directive.

The general procedure for using timer targets is as follows:

1. Identify a timer or counter in your model.

2. Instrument your model with a timer target that identifies a model item (Simulink block or Stateflow variable) as a timer or counter and specifies its start, end, and increment values.

3. Run Tester or Validator to generate a test which causes the timer to tick from its start to end value.
13.4 Interactively Tuning Test Suites

If the previously described techniques still do not result in a test suite with a satisfactory level of coverage, then some hand-tuning of the suite may be necessary. Reactis Simulator and Tester / Validator contain several features designed to facilitate this process, including test-suite pre-loading (Tester), and test-suite execution, breakpoints and test-suite editing (Simulator).

The general strategy to be pursued may be summarized as follows:

1. Build an initial test suite.
2. Load the test suite into Simulator and add a test that causes a “hard-to-cover” target to be covered.
3. Pre-load the modified test-suite into Tester / Validator and rerun.

The idea is to help Tester cover things it finds difficult while still letting Tester do as much work as possible.

Coverage problems often arise because of difficult-to-satisfy conditions on Stateflow transition segments or within Simulink. Conditions might be difficult to satisfy for several reasons:

1. A condition might become true when a timer expires, meaning that control must reside in the source state of the transition segment labeled by the condition for a number of simulation steps. If other segments’ conditions become true in the meantime, then the state may be exited before the timer has a chance to expire.

2. A condition might be satisfied by a relatively small subset of the values its variables might be able to assume. An example would be a comparison of a floating-point variable to a constant: only one value for the variable makes the condition true.

The strategy for coping with these problems can be summarized as follows:

1. Identify an uncovered transition-segment whose source state has been covered by a previously constructed test suite.
2. Using the test suite and breakpoints, advance execution until the source state is active.
3. Using user-guided simulation (Section 7.4.1), perform simulation steps that cause the uncovered item to become covered.
4. Add these simulation steps to the test suite, save the suite, and re-run Tester preloading the modified suite.

The specific steps needed to implement this strategy are as follows (It is assumed that Simulator is already running and that the earlier test suite has been loaded):

1. Click on the Coverage menu and make sure that Show Details is enabled.
2. Designate the (covered) source state of the uncovered item as a breakpoint by right-clicking on it in the hierarchy panel of Simulator and toggling the breakpoint indicator.
3. In the Source-of-Inputs dialog governing the source of inputs, select All.
4. Run the tests by clicking the Fast Simulation () button. Simulator will execute until the breakpoint is reached. Because execution breaks immediately prior to entry into the breakpoint state, hit the step button () to cause the state to be entered.

5. Select User Guided Simulation in the Source-of-Inputs dialog.

6. Using your knowledge of the system, select (a sequence of) input values that drive execution to cover the uncovered item.

7. Select Test Suite → Add/Extend Test to cause a new test to be added to the current test suite, which can now be seen in the Source-of-Inputs dialog.

8. Save the current test suite by selecting Test Suite → Save.

The newly saved test suite may now be pre-loaded into Tester using the Preload Files selection box, and Tester rerun. This strategy may be repeated until an acceptable test suite is constructed. Be sure to save the resulting test suite so that it will not accidentally be overwritten by new runs of Tester.

13.5 Model Decomposition

If the steps described above still result in unsatisfactory coverage, consider testing subsystems of your model independently. Reactis makes this possible through a facility to extract subsystems from a model. The idea is to extract a subsystem, along with portions of the model involved in triggering the subsystem, and store the results as a new model in a separate .slx file.

The subsystem extraction utility is invoked by loading a model into Reactis, then right-clicking on the Simulink subsystem or Stateflow chart to be extracted, and selecting the Extract Subsystem entry in the pop-up menu. In the resulting dialog a filename under which to save the extracted subsystem may be specified. Reactis will then extract the subsystem, save it under its new name, open a new Reactis window, and load the extracted model. Note that the extracted subsystem is saved as a standard .slx file. That makes it easy to edit the extracted model using Simulink if changes to the triggering mechanism or inputs or outputs of the model are necessary.

If the extracted subsystem is a triggered subsystem or located within a triggered subsystem or contains triggered subsystems whose triggers are connected to something outside the extracted subsystem, Reactis will keep all blocks and charts that are part of the triggering mechanism. This works recursively, i.e. if the triggering block or chart is triggered itself, Reactis will trace back that trigger, too.

To determine which subsystems might be candidates for independent testing, you can use the following strategy:

1. Generate a test suite for the full model.

2. Load the test suite into Reactis Simulator and select Coverage → Show Details.

3. Run the entire test suite on the model.

4. Inspect the subsystems within the model to look at the coverage obtained. Subsystems having poor coverage are good candidates for being tested independently.
Chapter 14

Preparing Models for Use with Reactis

To use Reactis on a Simulink / Stateflow model, you must ensure that the model satisfies certain constraints. This chapter describes what these constraints are. Please note that, while every effort is made to maintain the accuracy of the following list of supported and unsupported features, due to the complexity and continual evolution of the Simulink / Stateflow notation, this description may not be complete. For example, some blocks listed as supported do not support all parameters for the block. A block or feature not listed as either supported or unsupported should be assumed to be unsupported.

We encourage all potential customers to take advantage of a free evaluation license for Reactis to determine if the subset of Simulink / Stateflow supported by Reactis is adequate for your models. If it’s not, please let us know what features or blocks you would like to see Reactis support. Reactive Systems uses such feedback to prioritize enhancements to Reactis.

14.1 MATLAB

Reactis can process Simulink / Stateflow models that contain MATLAB code in callbacks or mask initializations, provided that certain conventions are observed. On the one hand, files that define workspace data items referred to in a model must be “connected” to the model appropriately, and certain MATLAB functions should be avoided. The remainder of this section elaborates on these points.

14.1.1 Workspace Data Items

Reactis invokes MATLAB to evaluate many MATLAB expressions, but it does not directly interact with an executing MATLAB session in the same way that Simulink does. For this reason, any workspace data items that a model uses must be initialized within one of the following locations:

- Any Simulink model callback or block callback that is executed when loading or running the model (PreLoadFcn, PostLoadFcn, InitFcn, StartFcn).
- A “startup.m” file located in the folder where the model file is located. When using this method, make sure that the “Execute startup.m and pathdef.m scripts in model folder” is checked in the “General” tab of the Reactis Info File Editor.
- The “Callbacks” tab of the Reactis Info File Editor. If in your environment the MATLAB workspace for your model is set up via some external script or graphical user interface
prior to loading the model, you can add the initialization code here so Reactis knows how to set up the workspace. Reactis will automatically execute the “Pre-Load Function” code prior to loading the model and the “Post-Load Function” code after loading the model. This method allows you to use Reactis in such an environment without having to modify your model.

For the cruise-control example, the file \texttt{cruise\_constants.m} defines two workspace variables that are used in \texttt{cruise.mdl}. One attaches \texttt{cruise\_constants.m} to \texttt{cruise.mdl} as follows:

1. Load \texttt{cruise.mdl} into Simulink.
2. From the Simulink window, select the File $\rightarrow$ Model Properties menu item.
3. In the resulting dialog, select the Callbacks tab.
4. In the Model pre-load function entry box enter \texttt{cruise\_constants;}.
5. Save the model.

In general, using the PreLoadFcn callback in this manner is good modeling practice, since once the .m files are attached to a model file, loading the model file into Simulink (and not just Reactis) will automatically load the .m files as well.

### 14.1.2 Unsupported MATLAB Features

While Reactis supports most of MATLAB, the following exceptions may not be used in Simulink / Stateflow models, e.g. in callbacks or mask initializations. Note, that a feature added in V2010.2 adds partial support for these commands. To enable the feature for a model, load the model, select menu item Edit $\rightarrow$ General, and then check the box “Propagate set_param changes by saving the model to a temporary file”. To enable this feature for all models, select File $\rightarrow$ Model Defaults... and then check the box “Propagate set_param changes by saving the model to a temporary file” in the General tab. If this setting is enabled, Reactis will cause Simulink to apply all the changes to the model, then automatically save them to a temporary file. Reactis then imports the temporary model to see the applied changes.

<table>
<thead>
<tr>
<th>Expression Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGNIN</td>
</tr>
<tr>
<td>EVALIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model Construction/Modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_BLOCK</td>
</tr>
<tr>
<td>ADD_LINE</td>
</tr>
<tr>
<td>ADD_PARAM</td>
</tr>
<tr>
<td>DELETE_BLOCK</td>
</tr>
<tr>
<td>DELETE_LINE</td>
</tr>
<tr>
<td>DELETE_PARAM</td>
</tr>
<tr>
<td>NEW_SYSTEM</td>
</tr>
<tr>
<td>REPLACE_BLOCK</td>
</tr>
<tr>
<td>SET_PARAM</td>
</tr>
</tbody>
</table>
14.2 Simulink

Reactis currently supports Simulink releases R2009a through R2020a. Most features of Simulink are supported; but the following are not supported by Reactis:

- Continuous-time blocks.
- The use of complex values (i.e. values with real and imaginary parts).
- Models containing corresponding DataStoreWrite/DataStoreRead blocks whose execution order is not explicitly defined by either the model logic or priorities.

For the subset of Simulink blocks supported by Reactis please refer to Section 14.2.3. For blocks that can be either continuous- or discrete-time, only the discrete-time version is supported. For some blocks identified as supported, not all settings are supported.

14.2.1 S-Functions

Reactis supports both C-Coded and M-File S-functions, with some restrictions. For C-Coded S-functions, the following are not supported by Reactis:

- Port-based sample times.
- Multiple sample times.
- Complex number signals.
- Zero-crossing detection.
- Output of function-calls.
- Level 1 S-functions (For a guide on how to convert Level 1 S-functions to Level 2 S-functions — which are supported by Reactis - please consult the MathWorks documentation)

- Calling any function from MATLAB’s “mex” library (including mexCallMATLAB, mexEvalString and mexGetVariable) from an S-Function.

For M-File S-functions, the following are not supported by Reactis:

- Multiple sample times.
- Complex number signals.
- Level 2 S-functions.

In addition to the above restrictions, care must be taken about any internal data that is stored by S-functions. In order to work properly, Reactis must be able to retrieve and reset the values of all internal states of any S-function occurring within a model. The best way to make internal states visible to Reactis (and Simulink) is to declare the appropriate number of discrete states in the mdlInitializeSizes() function and then use the state vector that Reactis and Simulink will provide. Reactis will also save and restore memory that an S-function allocates as a result of calling the ssGetNumRWork() and ssGetNumIWork() during mdlInitializeSizes.

Reactis has no way of knowing about any other persistent data that an S-function maintains by other means, such as:
• global or static variables in C-code;

• memory allocated by malloc() or mxMalloc() functions in C-code;

• Use of workspace variables in M-File S-functions.

Reactis will also not save and restore memory requested by ssGetNumPWork(), since otherwise pointers stored in this vector by your S-function might get lost or mangled, resulting in memory leaks or crashes.

If an S-function stores internal states in any of the unsupported ways described above, Reactis will seem to work properly, but the test suites generated by Tester may include wrong outputs. One sign of this can be if you run a test suite in Simulator and get an error message saying “Model fails test”. Another problem of such invalid use of internal states may be invalid outputs after using the “back” buttons in Simulator.

In general, Reactis passes S-Function parameters as fixed values at the time the S-Function is first initialized (i.e. when Tester or Simulator is started). Therefore, if a configuration variable is used as a parameter to an S-Function, the S-Function will not see any changes to the configuration variable unless the S-Function is designed to process such updates. To have an S-Function be updated on changes to its parameters, define a mdlProcessParameters function (see Simulink documentation) within the S-Function code. If this function is present, then Reactis will propagate the parameter changes into the S-Function by calling the S-Function’s mdlProcessParameters function at each step with the new parameter values. In the mdlProcessParameters function, the S-Function can then take appropriate actions if any parameters have changed.

### 14.2.2 Lookup Tables

This section describes the settings for each of the standard Simulink Lookup Table blocks that are supported by Reactis. Some settings might not be relevant depending on the version of MATLAB used. For example, many data type settings for the “Lookup Table (n-D)” block were only introduced in MATLAB R2009b and therefore are not relevant when using Reactis with MATLAB R2009a or earlier.

#### 14.2.2.1 1-D Lookup Table

Note that this block was previously named Lookup Table. All block settings and the data type combinations listed below are supported (note that fixpoint includes all fixed-point data types and integer data types int8, uint8, int16, uint16, int32 and uint32).
14.2. SIMULINK

### Supported Data Type Combinations

<table>
<thead>
<tr>
<th>Input of LUT</th>
<th>Output of LUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>single</td>
<td>double</td>
</tr>
<tr>
<td>fixpoint</td>
<td>double</td>
</tr>
<tr>
<td>single</td>
<td>single</td>
</tr>
<tr>
<td>fixpoint</td>
<td>single</td>
</tr>
<tr>
<td>double</td>
<td>fixpoint</td>
</tr>
<tr>
<td>single</td>
<td>fixpoint</td>
</tr>
<tr>
<td>fixpoint</td>
<td>fixpoint</td>
</tr>
<tr>
<td>double</td>
<td>boolean</td>
</tr>
<tr>
<td>single</td>
<td>boolean</td>
</tr>
<tr>
<td>fixpoint</td>
<td>boolean</td>
</tr>
</tbody>
</table>

#### 14.2.2.2 2-D Lookup Table

Note this block was previously named *Lookup Table (2-D)*. All block settings are supported. Supported data type combinations are listed in the table below (note that *fixpoint* includes all fixed-point data types and integer data types int8, uint8, int16, uint16, int32 and uint32).

<table>
<thead>
<tr>
<th>Supported Data Type Combinations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1st Input of LUT</strong></td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>single</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>single</td>
</tr>
<tr>
<td>single</td>
</tr>
<tr>
<td>single</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>bool</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>single</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>fixpoint</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>single</td>
</tr>
</tbody>
</table>

#### 14.2.2.3 n-D Lookup Table

This block was previously named *Lookup Table (n-D)*. Reactis supports a wide variety of n-D lookup tables, subject to restrictions which depend on the number of table dimensions. There are two levels of support, *native* and *via S-function*. 
Native support means Reactis will natively execute the n-D lookup table block and track coverage for it. Only n-D lookup tables which satisfy the following requirements are natively supported in Reactis:

1. The number of table dimensions is at most three, or the number dimensions is four and all input, output and internal data types are the same floating-point data type. Note that coverage tracking is only supported for n-D Lookup Table blocks with table dimensions two or less.

2. The interpolation and extrapolation methods are not cubic spline.

3. The types used in the table are restricted to the following: fixed-point types with a bias of zero and a slope that is a (negative) power of two, integer types, single- or double-precision floating-point types, Boolean.

Support via S-function means Reactis can execute the n-D lookup block but not track coverage. In order to execute the block, Reactis requires the sfun_lookupnd.mexw32 S-Function that comes with MATLAB R2007a or earlier. Starting with MATLAB R2007b, this S-Function is no longer included in the MATLAB distribution. However, if a sfun_lookupnd.mexw32 from R2007a or earlier is placed in Reactis’ search path, Reactis can use that S-Function even with R2007b and later.

n-D lookup tables which meet the following six requirements are supported via S-function in Reactis:

1. The input and output types are the same floating-point type.

2. The fraction data type is set to Inherit via internal rule.

3. The table data type is set to Inherit: same as output.

4. The breakpoint data type is set to Inherit: same as corresponding input.

5. All inputs are scalars.

6. If the table has more than one dimension, then the intermediate data type is set to Inherit: same as output.

14.2.2.4 Direct Lookup Table (n-D)

Only floating-point (double and single) data types are supported. This table is supported in Reactis via the sfun_nddirectlook.mexw32 S-Function that comes with MATLAB R2009a and earlier. Starting with MATLAB R2009b, this S-Function is no longer included in the MATLAB distribution. However, if a sfun_lookupnd.mexw32 from R2009a or earlier is placed in Reactis’ search path, Reactis can use that S-Function even with R2009b and later.

14.2.2.5 Lookup Table Dynamic

All parameters and the data type configurations listed for the “Lookup Table” block above are supported.
14.2.2.6 Prelookup

There are two levels of support for the Prelookup block in Reactis, native and via S-function. 

Native support means Reactis will natively execute the Prelookup block and track coverage for it. Reactis natively supports Prelookup blocks which use the following types: fixed-point types with a bias of zero and a slope that is a (negative) power of two, integer types, single- or double-precision floating-point types, Boolean.

Support via S-function means Reactis can execute the Prelookup block but not track coverage. In order to execute the block, Reactis requires the `sfun_idxsearch.mexw32` S-Function that comes with all versions of MATLAB.

All parameters are supported. The following data type restrictions apply:

- Only floating-point (double and single) input data types are supported.
- Only “int32” and “uint32” are supported as the data type of the “index” output.
- Only “single” and “double” are supported as the data type of the “fraction” output.

14.2.2.7 Interpolation using Prelookup

There are two levels of support for the Interpolation block in Reactis, native and via S-function. 

Native support means Reactis will natively execute the Interpolation block and track coverage for it. Only Interpolation blocks which satisfy the following requirements are natively supported in Reactis:

1. The number of table dimensions is at most three. Note that coverage tracking is only supported for n-D Lookup Table blocks with table dimensions two or less.
2. The interpolation and extrapolation methods are not cubic spline.
3. The types used in the table are restricted to the following: fixed-point types with a bias of zero and a slope that is a (negative) power of two, integer types, single- or double-precision floating-point types, Boolean.

Support via S-function means Reactis can execute the Interpolation block but not track coverage. In order to execute the block, Reactis requires the `sfun_kflookupnd.mexw32` S-Function that comes with all versions of MATLAB.

All parameters are supported. The following data type restrictions apply:

- Only floating-point (double and single) data types are supported for “fraction” inputs.
- Data types of all “fraction” inputs must be the same as the output data type.
- Only “int32” and “uint32” are supported as the data type of the “index” input.
- Table data type must match output data type
- Intermediate results data type must match output data type.
## 14.2.3 Table of Supported Blocks

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Commonly Used Blocks</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus Creator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Bus Selector</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data-Type Conversion</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Demux</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Discrete-Time Integrator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Gain</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Ground</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Import</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Integrator</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Logical Operator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Mux</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Outport</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Relational Operator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Saturation</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Scope</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Switch</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Terminator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Vector Concatenate</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td><strong>Continuous</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Derivative</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Integrator Limited</td>
<td>R2010a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Integrator Second-Order</td>
<td>R2010a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Integrator Second-Order Limited</td>
<td>R2010a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>PID Controller</td>
<td>R2009b</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>PID Controller (2 DOF)</td>
<td>R2009b</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>State-Space</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Transfer Fcn</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Transport Delay</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Variable Time Delay</td>
<td>R2007b</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Variable Transport Delay</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Zero-Pole</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td><strong>Discontinuities</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Backlash</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Coulomb &amp; Viscous Friction</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Dead Zone</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Dead Zone Dynamic</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Hit Crossing</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Quantizer</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Introduced</td>
<td>Reactis Support</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------</td>
<td>------------</td>
<td>-----------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Rate Limiter</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Rate Limiter Dynamic</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Relay</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Saturation</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Saturation Dynamic</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Wrap To Zero</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td><strong>Discrete</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| Delay             | R2011b     | Y               | In R2011b, the Integer Delay block was extended and renamed to the Delay block. Reactis requires:  
|                   |            |                 | • Delay length source must be Dialog  
|                   |            |                 | • Initial condition source must be Dialog  
|                   |            |                 | • External reset must be None  
<p>|                   |            |                 | • Input processing must be Elements as channels (sample based) or Inherited |
| Difference        | R14        | Y               |                                                                      |
| Discrete Derivative | R14      | Y               |                                                                      |
| Discrete Filter   | pre-R14    | Y               |                                                                      |
| Discrete FIR Filter | R2008a  | Y               |                                                                      |
| Discrete PID Controller | R2009b | N               |                                                                      |
| Discrete PID Controller (2 DOF) | R2009b | N               |                                                                      |
| Discrete State-Space | pre-R14 | Y               |                                                                      |
| Discrete Transfer Fcn | pre-R14 | Y               |                                                                      |
| Discrete Zero-Pole | R14        | Y               |                                                                      |
| Discrete-Time Integrator | pre-R14 | Y               |                                                                      |
| First-Order Hold  | R14        | N               |                                                                      |
| Memory            | pre-R14    | Y               |                                                                      |
| Tapped Delay      | R14        | Y               |                                                                      |
| Transfer Fcn First Order | R14 | Y               |                                                                      |
| Transfer Fcn Lead or Lag | R14 | Y               |                                                                      |
| Transfer Fcn Real Zero | R14 | Y               |                                                                      |
| Unit Delay        | pre-R14    | Y               |                                                                      |
| Zero-Order Hold   | pre-R14    | Y               |                                                                      |
| <strong>Logic and Bit Operations</strong> |         |                 |                                                                      |
| Bit Clear         | pre-R14    | Y               |                                                                      |
| Bit Set           | pre-R14    | Y               |                                                                      |
| Bitwise Operator  | pre-R14    | Y               |                                                                      |
| Combinatorial Logic | pre-R14 | Y               |                                                                      |
| Compare To Constant | pre-R14 | Y               |                                                                      |
| Compare To Zero   | pre-R14    | Y               |                                                                      |
| Detect Change     | pre-R14    | Y               |                                                                      |
| Detect Decrease   | pre-R14    | Y               |                                                                      |
| Detect Fall Negative | pre-R14 | Y               |                                                                      |
| Detect Fall Nonpositive | pre-R14 | Y               |                                                                      |
| Detect Increase   | pre-R14    | Y               |                                                                      |
| Detect Rise Nonnegative | pre-R14 | Y               |                                                                      |
| Detect Rise Positive | pre-R14 | Y               |                                                                      |
| Extract Bits      | R14        | Y               |                                                                      |</p>
<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interval Test</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Interval Test Dynamic</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Logical Operator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Relational Operator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Resettable Delay</td>
<td>R2012b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Shift Arithmetic</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Variable Integer Delay</td>
<td>R2012b</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td><strong>Lookup Tables</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-D Lookup Table</td>
<td>pre-R14</td>
<td>Y</td>
<td>Previously named LookUp Table. See Section 14.2.2.1</td>
</tr>
<tr>
<td>2-D Lookup Table</td>
<td>pre-R14</td>
<td>Y</td>
<td>Previously named LookUp Table (2-D). See Section 14.2.2.2</td>
</tr>
<tr>
<td>Cosine</td>
<td>pre-R14</td>
<td>Y</td>
<td>Only output formula cos</td>
</tr>
<tr>
<td>Direct Lookup Table (n-D)</td>
<td>pre-R14</td>
<td>Y</td>
<td>See Section 14.2.2.4</td>
</tr>
<tr>
<td>Interpolation using Prelookup</td>
<td>pre-R14</td>
<td>Y</td>
<td>See Section 14.2.2.7</td>
</tr>
<tr>
<td>Lookup Table Dynamic</td>
<td>R14</td>
<td>Y</td>
<td>See Section 14.2.2.5</td>
</tr>
<tr>
<td>n-D Lookup Table</td>
<td>pre-R14</td>
<td>Y</td>
<td>See Section 14.2.2.3</td>
</tr>
<tr>
<td>Prelookup</td>
<td>pre-R14</td>
<td>Y</td>
<td>See Section 14.2.2.6</td>
</tr>
<tr>
<td>Sine</td>
<td>pre-R14</td>
<td>Y</td>
<td>Only output formula sin</td>
</tr>
<tr>
<td><strong>Math Operations</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abs</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Algebraic Constraint</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Bias</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Complex to Magnitude-Angle</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Complex to Real-Imag</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Divide</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Dot Product</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Find Nonzero Elements</td>
<td>R2010a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Gain</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Magnitude-Angle to Complex</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Math Function</td>
<td>pre-R14</td>
<td>Y</td>
<td>Except Hermitian</td>
</tr>
<tr>
<td>Matrix Concatenate</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>MinMax</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>MinMax Running Resettable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Permute Dimensions</td>
<td>R2007a</td>
<td>Y</td>
<td>2 dimensions only</td>
</tr>
<tr>
<td>Polynomial</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Product</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Product of Elements</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Real-Imag to Complex</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Reciprocal Sqrt</td>
<td>R2010a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Resshape</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Rounding Function</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sign</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Signed Sqrt</td>
<td>R2010a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Introduced</td>
<td>Reactis Support</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------</td>
<td>-----------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>Sine Wave Function</td>
<td>pre-R14</td>
<td>Y</td>
<td>Using simulation time</td>
</tr>
<tr>
<td>Slider Gain</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sqrt</td>
<td>R2010a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Squeeze</td>
<td>R2007a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Subtract</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Sum of Elements</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Trigonometric Function</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unary Minus</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Vector Concatenate</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Weighted Sample Time Math</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

**Model Verification**

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assertion</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Discrete Gradient</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Dynamic Gap</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Dynamic Lower Bound</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Dynamic Range</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Dynamic Upper Bound</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Input Resolution</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Static Gap</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Static Lower Bound</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Static Range</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Check Static Upper Bound</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

**Model-Wide Utilities**

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Support Table</td>
<td>R2007a</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>DocBlock</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Model Info</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Time-Based Linearization</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Trigger-Based Linearization</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

**Ports & Subsystems**

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>CodeReuse Subsystem</td>
<td>R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Configurable Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Enable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Enabled Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Enabled and Triggered Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>For Each Subsystem</td>
<td>R2010a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>For Iterator Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Function-Call Feedback Latch</td>
<td>R2011a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Function-Call Generator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Function-Call Split</td>
<td>R2010a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Block</td>
<td>Introduced</td>
<td>Reactis Support</td>
<td>Notes</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>Function-Call Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>If</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>If Action Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>In Bus Element</td>
<td>R2017a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Inport</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Model</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Model Variant</td>
<td>R2009b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Out Bus Element</td>
<td>R2017a</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Outport</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Switch Case</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Switch Case Action Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Trigger</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Triggered Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Variant Subsystem</td>
<td>R2010b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>While Iterator Subsystem</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td><strong>Signal Attributes</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus to Vector</td>
<td>R2007b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Type Conversion</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Type Conversion Inherited</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Type Duplicate</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Type Propagation</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Type Scaling Strip</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>IC</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Probe</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Rate Transition</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Signal Conversion</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Signal Specification</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Weighted Sample Time</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td><strong>Signal Routing</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bus Assignment</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Bus Creator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Bus Selector</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Store Memory</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Store Read</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Data Store Write</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Demux</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Environment Controller</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>From</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Goto</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Goto Tag Visibility</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Index Vector</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Manual Switch</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Merge</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
### 14.2. SIMULINK

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiport Switch</td>
<td>pre-R14</td>
<td>Y</td>
<td>'Data port for default case' must be 'Last data port' and 'Diagnostic for default case' must be 'Error'</td>
</tr>
<tr>
<td>Mux</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Selector</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Switch</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Vector Concatenate</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

#### Sinks

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Floating Scope</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Manual Variant Sink</td>
<td>R2016b</td>
<td>Y</td>
<td>You can configure Reactis to ignore To File blocks from the Settings dialog.</td>
</tr>
<tr>
<td>Outport</td>
<td>pre-R14</td>
<td>Y</td>
<td>You can configure Reactis to ignore To Workspace blocks from the Settings dialog.</td>
</tr>
<tr>
<td>Scope</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Stop Simulation</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Terminator</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>To File</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>To Workspace</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Variant Sink</td>
<td>R2016b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>XY Graph</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

#### Sources

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Band-Limited White Noise</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Chirp Signal</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Clock</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Counter Free-Running</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Counter Limited</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Digital Clock</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Enumerated Constant</td>
<td>R2009b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>From File</td>
<td>pre-R14</td>
<td>Y</td>
<td>.mat file must be in ‘Array’ format, not ‘MATLAB timeseries object’ format. Linear interpolation and extrapolation must be used.</td>
</tr>
<tr>
<td>From Workspace</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Ground</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Inport</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Manual Variant Source</td>
<td>R2016b</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Pulse Generator</td>
<td>pre-R14</td>
<td>Y</td>
<td>'Pulse type' must be 'sample based' and 'Time (t)' must be 'Use simulation time'</td>
</tr>
<tr>
<td>Ramp</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Random Number</td>
<td>pre-R14</td>
<td>Y</td>
<td>This block may only be used in virtual sources.</td>
</tr>
<tr>
<td>Repeating Sequence</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Repeating Sequence Interpolated</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Repeating Sequence Stair</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Signal Builder</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Signal Generator</td>
<td>pre-R14</td>
<td>Y</td>
<td>'Wave form' of 'random' not supported</td>
</tr>
<tr>
<td>Sine Wave</td>
<td>pre-R14</td>
<td>Y</td>
<td>'Time (t)' must be 'Use simulation time'</td>
</tr>
<tr>
<td>Step</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
## CHAPTER 14. PREPARING MODELS FOR USE WITH REACTIS

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uniform Random Number</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Variant Source</td>
<td>R2016b</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

### User-Defined Functions

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fcn</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Interpreted MATLAB Function</td>
<td>pre-R14</td>
<td>N</td>
<td>Previously named MATLAB Fcn</td>
</tr>
<tr>
<td>Level-2 MATLAB S-Function</td>
<td>pre-R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>MATLAB Function</td>
<td>pre-R14</td>
<td>Y</td>
<td>Previously named Embedded MATLAB Function. Only a subset of the Embedded MATLAB language is supported. See Section 14.4 for details.</td>
</tr>
<tr>
<td>MATLAB System</td>
<td>R2013b</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>S-function</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>S-function Builder</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

### Additional Discrete

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-Point State-Space</td>
<td>R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>‘Transfer Fcn Direct Form II’</td>
<td>R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>‘Transfer Fcn Direct Form II Time Varying’</td>
<td>R14</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Enabled</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Enabled External IC</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Enabled Resettable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Enabled Resettable External IC</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay External IC</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Resettable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay Resettable External IC</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay With Preview Enabled</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay With Preview Enabled Resettable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay With Preview Resettable External RV</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay With Preview Resettable</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Unit Delay With Preview Enabled Resettable External RV</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

### Additional Math

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decrement Real World</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Decrement Stored Integer</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Decrement Time To Zero</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Decrement To Zero</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Increment Real World</td>
<td>pre-R14</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
14.2. SIMULINK

<table>
<thead>
<tr>
<th>Block</th>
<th>Introduced</th>
<th>Reactis Support</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment Stored Integer</td>
<td>pre-R14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stateflow</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chart</td>
<td>pre-R14</td>
<td>Y</td>
<td>See Section 14.3.</td>
</tr>
<tr>
<td>Truth Table</td>
<td>R14</td>
<td>Y</td>
<td>See Section 14.4 for supported subset of the underlying Embedded MATLAB language.</td>
</tr>
</tbody>
</table>

### 14.2.4 Simulink Extras

The following table lists the supported blocks from the “Simulink extras” library which is available in all MATLAB versions:

<table>
<thead>
<tr>
<th>Additional Discrete</th>
<th>Supported: Discrete Transfer Fcn (with initial outputs), Discrete Transfer Fcn (with initial states), Discrete Zero-Pole (with initial outputs), Discrete Zero-Pole (with initial states), Idealized ADC Quantizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsupported:</td>
<td></td>
</tr>
<tr>
<td>Additional Linear</td>
<td>Supported:</td>
</tr>
<tr>
<td>Unsupported:</td>
<td>State-Space (with initial outputs), Transfer Fcn (with initial outputs), Transfer Fcn (with initial states), Zero-Pole (with initial outputs), Zero-Pole (with initial states)</td>
</tr>
<tr>
<td>Additional Sinks</td>
<td>Supported:</td>
</tr>
<tr>
<td>Flip Flops</td>
<td>Supported: D Flip-Flop, DLatch, J-K Flip-Flop, S-R Flip-Flop</td>
</tr>
<tr>
<td>Unsupported:</td>
<td>Clock</td>
</tr>
<tr>
<td>Linearization</td>
<td>Supported:</td>
</tr>
<tr>
<td>Unsupported:</td>
<td>Switched derivative for linearization, Switched transport delay for linearization</td>
</tr>
<tr>
<td>Transformations</td>
<td>Supported:</td>
</tr>
<tr>
<td>Unsupported:</td>
<td>Cartesian to Polar, Cartesian to Spherical, Celsius to Fahrenheit, Degrees to Radians, Fahrenheit to Celsius, Polar to Cartesian, Radians to Degrees, Spherical to Cartesian</td>
</tr>
<tr>
<td>Unsupported:</td>
<td></td>
</tr>
</tbody>
</table>

### 14.2.5 TargetLink 2.2.1 Library Blocks

The following table lists the supported blocks from the dSPACE TargetLink library (version 2.2.1), if installed:

<table>
<thead>
<tr>
<th>Main TargetLink Library</th>
<th>Supported:</th>
</tr>
</thead>
</table>
## 14.2.6 TargetLink 2.3.1 Library Blocks

The following table lists the supported blocks from the dSPACE TargetLink library (version 2.3.1), if installed:

<table>
<thead>
<tr>
<th>Supported</th>
<th>Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main TargetLink Library</td>
<td>Interpolation (n-D) using PreLook-Up, Rate Limiter</td>
</tr>
<tr>
<td>Non-Linear</td>
<td>Dead Zone Positive, Dead Zone Negative, Saturation Dynamic, Backlash</td>
</tr>
<tr>
<td>Extras</td>
<td>Dead Zone, Dead Zone Dynamic</td>
</tr>
<tr>
<td>Bit Operations</td>
<td>U8 Bit Set, U16 Bit Set, U32 Bit Set, U8 Bit Clear, U16 Bit Clear, U32 Bit Clear, Split U16, Split U32, Combine U16, Combine U32, 8Bit Decoder, 8Bit Encoder, 16Bit Decoder, 16Bit Encoder, Bitwise Logical Operator</td>
</tr>
<tr>
<td>TargetLink-Supported Simulink Blocks</td>
<td>Please see list of Reactis-Supported Simulink Blocks</td>
</tr>
</tbody>
</table>
### 14.2. SIMULINK

<table>
<thead>
<tr>
<th><strong>Math</strong></th>
<th>Supported: Rounding Function, Transport Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Extras</strong></td>
<td>Supported: D Flip-Flop, D Latch, J-K Flip-Flop, S-R Flip-Flop, Preprocessor IF</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Bit Operations</strong></td>
<td>Supported: U8 Bit Set, U16 Bit Set, U32 Bit Set, U8 Bit Clear, U16 Bit Clear, U32 Bit Clear, Split U16, Split U32, Combine U16, Combine U32, 8Bit Decoder, 8Bit Encoder, 16Bit Decoder, 16Bit Encoder, Bitwise Logical Operator</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
</tbody>
</table>

#### 14.2.7 TargetLink 3.0 to 3.4 Library Blocks

The following table lists the supported blocks from the dSPACE TargetLink library (versions 3.0 to 3.4), if installed:

<table>
<thead>
<tr>
<th><strong>Main TargetLink Library</strong></th>
<th>Supported: Inport, Outport, Constant, Sum, Gain, Product, Logical Operator, Relational Operator, Fcn, Look-Up Table, Look-up Table (2-D), Direct Look-up Table (n-D), PreLook-Up Index Search, Interpolation (n-D) using PreLook-Up, Saturation, MinMax, Abs, Sign, Rate Limiter, Relay, Trigonometric Function, Math, Unit Delay, Discrete Transfer Fcn, FIR Filter, Discrete-Time Integrator, Discrete State-Space, Data Store Write, Data Store Memory, Data Store Read, Merge, Sink, Bus Inport, Bus Outport, Switch, Multiport Switch, Unit Delay (Reset Enabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Non-Linear</strong></td>
<td>Supported: Backlash, Dead Zone Positive, Dead Zone Negative, Saturation Dynamic, Dead Zone, Dead Zone Dynamic</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Math</strong></td>
<td>Supported: Rounding Function, Transport Delay</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Extras</strong></td>
<td>Supported: D Flip-Flop, D Latch, J-K Flip-Flop, S-R Flip-Flop, Preprocessor IF</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
<tr>
<td><strong>Bit Operations</strong></td>
<td>Supported: U8 Bit Set, U16 Bit Set, U32 Bit Set, U8 Bit Clear, U16 Bit Clear, U32 Bit Clear, Split U16, Split U32, Combine U16, Combine U32, 8Bit Decoder, 8Bit Encoder, 16Bit Decoder, 16Bit Encoder, Bitwise Logical Operator</td>
</tr>
<tr>
<td></td>
<td>Unsupported:</td>
</tr>
</tbody>
</table>
14.2.8 TargetLink 3.5 to 4.2 Library Blocks

The following table lists the supported blocks from the dSPACE TargetLink library (versions 3.5 to 4.2), if installed:

<table>
<thead>
<tr>
<th>Main TargetLink Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supported: Inport, Outport, Constant, Sum, Gain, Product, Logical Operator, Relational Operator, Fcn, Look-Up Table, Look-up Table (2-D), Direct Look-up Table (n-D), PreLook-Up Index Search, Interpolation (n-D) using PreLook-Up, Saturation, MinMax, Abs, Sign, Rate Limiter, Relay, Trigonometric Function, Math, Unit Delay, Discrete Transfer Fcn, Discrete-Time Integrator, Discrete State-Space, Data Store Write, Data Store Memory, Data Store Read, Merge, Sink, Bus Inport, Bus Outport, Switch, Multiport Switch, Unit Delay (Reset Enabled)</td>
</tr>
<tr>
<td>Unsupported: FIR Filter</td>
</tr>
<tr>
<td>Non-Linear</td>
</tr>
<tr>
<td>Supported: Backlash, Dead Zone Positive, Dead Zone Negative, Saturation Dynamic, Dead Zone, Dead Zone Dynamic</td>
</tr>
<tr>
<td>Unsupported:</td>
</tr>
<tr>
<td>Math</td>
</tr>
<tr>
<td>Supported: Rounding Function</td>
</tr>
<tr>
<td>Unsupported: Transport Delay</td>
</tr>
<tr>
<td>Extras</td>
</tr>
<tr>
<td>Supported: D Flip-Flop, D Latch, J-K Flip-Flop, S-R Flip-Flop, Preprocessor IF</td>
</tr>
<tr>
<td>Unsupported:</td>
</tr>
<tr>
<td>Bit Operations</td>
</tr>
<tr>
<td>Supported: U8 Bit Set, U16 Bit Set, U32 Bit Set, U8 Bit Clear, U16 Bit Clear, U32 Bit Clear, Split U16, Split U32, Combine U16, Combine U32, 8Bit Decoder, 8Bit Encoder, 16Bit Decoder, 16Bit Encoder, Bitwise Logical Operator</td>
</tr>
<tr>
<td>Unsupported:</td>
</tr>
<tr>
<td>TargetLink-Supported Simulink Blocks</td>
</tr>
<tr>
<td>Supported: Please see list of Reactis-Supported Simulink Blocks</td>
</tr>
<tr>
<td>Unsupported:</td>
</tr>
</tbody>
</table>

14.3 Stateflow

Reactis supports most of Stateflow. Some exceptions are the following unsupported features:

- Charts with the action language set to MATLAB are currently supported if they use the
language subset described in Section 14.4. Charts with “C” action language are supported.

- Implicit “enter”, “exit” and “change” events
- Range limits for variables.
- “ml.” name-space operator and “ml()” function call.
- Embedded MATLAB code. Reactis supports the language subset described in Section 14.4.
- Using Stateflow keywords as variable names. The Stateflow keywords are: at, after, before, change, du, during, enter, en, entry, every, ex, exit, in, on, ml, send, abs, acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, labs, ldexp, log, log10, min, max, pow, rand, rem, sin, sinh, sqrt, tan, tanh, int8, int16, int32, uint8, uint16, uint32, double, boolean.
- The explicit type cast function (“cast”).
- The address operator (&) is supported only in calls to external C functions.
- The pointer operator (*) is supported only inside a literal C code section.
- Charts without trigger or sample time are supported only if the model has a fixed sample time.
- In some cases, the detection of inner transitions fails for odd-shaped transitions (for example, single transition segments which leave and reenter a state).
- The temporalCount operator.
- Atomic Subcharts (introduced in R2010b).

14.4 Embedded MATLAB

Reactis V2012 introduced black-box support for Embedded MATLAB. This includes:

**Simulink**
- MATLAB Function block
- Truth Table block

**Stateflow**
- MATLAB functions
- Truth Table functions with MATLAB language option
- MATLAB action language in Stateflow

Note that a subset of the full Embedded MATLAB language is supported. To use models containing Embedded MATLAB in Reactis, your model must only use the language subset supported by Reactis. Note that the same language restrictions apply no matter which of the five model constructs listed above are used to incorporate Embedded MATLAB into your model. To avoid confusion, we use the following abbreviations in the following discussion:
Embedded MATLAB (EML)  the subset of MATLAB supported by MathWorks for code generation.

Reactis Embedded MATLAB (REML)  the subset of EML supported by Reactis.

REML is under active development towards the ultimate goal of supporting a very large subset of EML. Please send requests to Reactive Systems (help@reactive-systems.com) if there are unsupported EML features you would like to use.

The V2015 release of Reactis included a new product: the Reactis for EML Plugin. It integrates with Reactis to offer white-box testing of the EML portions of a model. See Chapter 17 for a description of the Reactis for EML Plugin. If not using the new product, you can still test models containing EML; however, exercising coverage targets within the EML will not be an objective of test generation and tracking this coverage will not be available. Whether in white-box or black-box mode for EML, Reactis supports the same subset of EML. This subset is defined in Section 17.4.
Chapter 15

Model Semantics

A number of different tools interpret how Simulink / Stateflow models execute. These include Reactis, the MathWorks Simulation environment, and various autocode tools. In this chapter we discuss issues related to the semantics of Simulink / Stateflow models. In particular we focus on how model execution affects coverage tracking.

15.1 Conditional Input Branch Execution

A number of factors influence whether or not a Simulink block executes during a given simulation step. These include whether or not the block (or a subsystem containing the block) has a sample time and whether or not the block resides in a conditionally executed subsystem. Conditional input branch execution is an optimization supported by Simulink that offers an additional way to determine when blocks execute by disabling them in some cases. Beginning with V2007, Reactis also supports this feature.

To understand how conditional input branch execution works, consider the simple model shown in Figure 15.1. When input $c$ is greater than or equal to zero, the Switch block outputs RelOp1 and ignores RelOp2. Conversely, when $c$ is less than zero, the Switch block outputs RelOp2 and ignores RelOp1. The basic idea behind conditional input branch execution is to not execute blocks whose output is ignored.

This optimization does not change the outputs computed by the model, but it will speed
up execution since some blocks are not executed. In the case of Reactis, the optimization has an important impact on coverage tracking: the coverage targets associated with a block are not covered during any simulation step in which the block is ignored due to conditional input branch execution. For example, in Figure 15.1, neither the true nor false branch of RelOp2 are covered when \( c \) is greater than or equal to zero.

In addition to switch blocks, conditional input branch execution also occurs when Multiport Switch blocks or non-virtual subsystems are executed. The non-control inports of Multiport Switches and the inports of non-virtual subsystems each serve as the root of an ignorable branch just as each non-control inport of a switch block does. We call these inports ignorable inports.

In Figure 15.1, each ignorable branch consists of a single block wired to an ignorable inport; however, in general, an entire group of blocks might be ignored. The set of blocks included in an ignorable branch is computed as follows. A block \( B \) is included in an ignorable branch \( IB \) if it meets each of the following conditions:

- \( B \) is a direct-feedthrough block. Such blocks do not maintain a state, instead their output values can be computed as a function only of their inputs. Examples of direct-feedthrough blocks include Gain, Sum, Logical Operator, and Relational Operator blocks. The following blocks are not direct-feedthrough blocks: Unit Delay, Discrete Integrator, Discrete State Space, Hit Crossing, Memory, Relay, Integer Delay, and S-Function (if there are ports that are not direct-feed-through).

- \( B \)'s outports only feed into either:
  - the ignorable port at the root of \( IB \); or
  - the inports of blocks in \( IB \).

- \( B \) has the same underlying execution schedule as the block containing the ignorable inport at the root of \( IB \). Two blocks have the same underlying execution schedule if:
  - both are triggered by same trigger; and
  - both are enabled by same enable port; and
  - both have the same sample time (possibly inherited).

In Simulink, a model is configured to use conditional branch execution in two ways.

- For switch and Multiport Switch blocks, the Simulation Parameters dialog is used (see Figure 15.2). The following steps are required:
  1. Load your model in Simulink.
  2. Select Simulation \( \rightarrow \) Configuration Parameters... menu item.
  3. Click the Optimization tab.
  4. Check the box for Conditional input branch execution (item 1 in Figure 15.2).

- Non-virtual subsystems are configured on a subsystem-by-subsystem basis, by performing the following steps:
  1. Load your model in Simulink.
  2. Right-click on a non-virtual subsystem and select Subsystem parameters...
3. Check the box for Propagate execution context across subsystem boundary.

Conditional input branch execution in Reactis is controlled independently for each model. The steps required to change the setting are as follows:

1. In the main Reactis window, select Edit → General....

2. The Reactis info file editor will appear with the General... tab selected. This tab contains a parameter Conditional input branch execution, which can be set to On, Off, or Inherited from model settings.
The base Reactis product treats S-Functions and Stateflow custom C code as black boxes. Models containing such C code can be processed, but the C source code is inaccessible. Hence, there is no way to perform basic debugging operations such as single-stepping, viewing variable values, setting breakpoints, or browsing the source code during simulation. Furthermore, targets cannot be created within the C source code when generating tests. The Reactis for C Plugin integrates seamlessly with Reactis to offer white-box testing and validation of the C-code portions of models (S-Functions and C code called from Stateflow). Together, Reactis and Reactis for C Plugin automate the generation of test data from, and validation of models constructed with Simulink, Stateflow, and C code.

A separate license is required to use the Reactis for C Plugin. If your organization does not yet own such a license, please contact sales@reactive-systems.com to inquire about purchase options or to obtain a demo license for the product. The Reactis for C Plugin requires base Reactis to run.
16.1 Quick Start

1. In the main Reactis window, select File \rightarrow Global Settings... and then display the Reactis for C tab. Make sure the Enable white-box analysis of C code check-box is selected.

2. To subject an S-Function to white-box analysis, do the following:
   (a) Right-click on the S-Function in the main Reactis panel and select Assign RSM File
   (b) Accept the default file name
   (c) Click Yes to create a new \texttt{.rsm} file
   (d) Add the C source files that implement the S-Function using the dialog.

3. Stateflow custom C code will be processed automatically when the Reactis for C Plugin is enabled. No \texttt{.rsm} file is necessary.

4. Navigate to an S-Function in the hierarchy panel to display the C code for the S-Function. C code called from Stateflow will be listed at the bottom of the hierarchy panel in the C Libraries section in a subsection called Stateflow User Sources.

5. The C code can be observed from Simulator and targets within the C code will be used by Tester to generate test suites that aim to maximize coverage of the C code.

6. Selecting Edit \rightarrow C Code will bring up the C code tab in the Info File Editor. This dialog displays the locations of all C components in the model. From here \texttt{.rsm} files can be added, edited or removed by right-clicking. See section 16.2 for more details.

16.2 Configuring Reactis for C

16.2.1 Global Settings

Figure 16.1 shows the Reactis for C Plugin tab of the Reactis Global Settings dialog that lets you configure some aspects of the Reactis for C Plugin. The purpose of each dialog element is as follows:

1. **Enable white-box analysis of C code.** Enable or disable the Reactis for C Plugin.

2. **Display line numbers.** Toggle the display of line numbers in the main Reactis panel when displaying C source files. The Background Color button allows you to select the background color of screen region where the line numbers are displayed.

3. **Display line grid.** Toggle the highlighting of every other line a different color. The Background Color button allows you to select which color is used.

4. **Highlight comments.** Toggle the highlighting of comments within C source code. The Text Color button allows you to select the color of comment text.

5. **Highlight inactive code.** Toggle the highlighting of code which has been disabled by preprocessor directives such as \#ifdef. The Text Color button allows you to select the color of inactive code text.

6. **Source File Editor.** This allows you to specify an external editor that can be launched from Reactis to edit C source files.
16.2.2 Model-Specific Settings

16.2.2.1 Error Checking Pane

Figure 16.2 shows the model-specific settings for C code. The effect of each of these settings is explained below.

1. **On signed integer overflow.** This item determines the behavior of the C-Plugin when overflow occurs during a signed integer computation. The effect of integer overflows outside of the C-Plugin are not affected. There are three possible settings:

   - **Wrap over.** Emulate typical C runtime behavior and accept the hardware-produced result.
   - **Produce warning.** Emulate typical C runtime behavior and accept the hardware-produced result, but also generate a warning.
   - **Produce error.** Stop execution and generate an error message.

   Integer overflows are discussed in more detail in Section 16.10.

2. **On unsigned integer overflow.** This item determines the behavior of the C-Plugin when overflow occurs during a signed integer computation. The effect of integer overflows outside of the C-Plugin are not affected. There are three possible settings:
Figure 16.2: The Error Checking pane of the Reactis Info File Editor with the C code settings shown.

Wrap over. Emulate typical C runtime behavior and accept the hardware-produced result.

Produce warning. Emulate typical C runtime behavior and accept the hardware-produced result, but also generate a warning.

Produce error. Stop execution and generate an error message.

Note that C specifies that unsigned integer calculations should wrap around when a result is too large to fit, and many programs are designed with this behavior in mind.

3. **On overflow during conversion from unsigned to signed integer.** This item determines the behavior of the C-Plugin when an overflow occurs while converting an unsigned integer to a signed integer. There are four possible settings:

Wrap over. Emulate typical C runtime behavior and accept the hardware-produced result.

Produce warning. Emulate typical C runtime behavior and accept the hardware-produced result, but also generate a warning.

Produce error. Stop execution and generate an error message.

Inherit from “On signed integer overflow.” Behave according to the setting of On signed integer overflow.

Integer conversions are discussed in more detail in Section 16.10.

4. **On overflow during conversion from floating-point to integer.** This item determines the behavior of the C-Plugin when an overflow occurs while converting a floating-point value to integer. There are four possible settings:

Wrap over. Emulate typical C runtime behavior and accept the hardware-produced result.

Produce warning. Emulate typical C runtime behavior and accept the hardware-produced result, but also generate a warning.
Produce error. Stop execution and generate an error message.

Inherit from “On signed integer overflow.” Behave according to the setting of On signed integer overflow (the previous setting).

Integer conversions are discussed in more detail in Section 16.10.

5. When detecting Inf or NaN values. This setting determines what action is taken when one of the values NaN, +inf, or -inf is produced by an expression whose input values do not include NaN, +inf, or -inf. Possible settings are:

Do nothing. Ignore NaN and infinite values and allow execution to continue.
Produce warning. Generate a warning message and allow execution to continue.
Produce error. Stop execution and generate an error message.

Inherit from “When detecting Inf or NaN values.” Behave according to the setting of When detecting Inf or NaN values in the Simulink and Stateflow section of this pane.

6. On invalid pointer creation. This item determines what happens when a pointer expression yields an invalid address. There are three possible settings:

Produce error. Stop execution and generate an error message.
Produce warning. Generate a warning message and allow execution to continue.
Do nothing. Ignore the invalid pointer. An error will occur if the pointer is used to access memory (which might never happen).

Details on the use of this option are provided in Section 16.11.1.

7. When an empty struct/union type is created. This settings determines what happens when a struct or union type is defined which does not have any members. There are three possible settings:

Produce error. Stop execution and generate an error message.
Produce warning. Generate a warning message and allow execution to continue.
Do nothing. Ignore the empty structure type. Errors may occur if variables of the empty type are used at runtime.

8. When an undefined function is called. This settings determines what happens when a function which has been declared but not defined is called. There are two possible settings:

Return zero. Act as if the function has no side-effects and returns a value of zero. This allows unit testing to be done without generating stub code for missing functions.
Produce error. Stop execution and generate an error message.

9. When an extern variable is undefined. This settings determines what happens when an extern variable is declared but never defined. There are three possible settings:

Initialize to zero. Define the variable with an initial value of zero.
Produce warning. Define the variable with an initial value of zero, but also generate a warning.
**Produce error.** Stop execution and generate an error message.

9. **Infinite loop timeout (seconds.)** Maximum time for an S-function or C custom code to complete. If the computation exceeds this limit, Reactis assumes there is an infinite loop and the simulation is then terminated and an error message printed. A value of zero disables the timeout.

### 16.2.2.2 Coverage Metrics Pane

![Figure 16.3: C-Specific coverage metrics.](image)

Figure 16.3 shows the C-specific coverage metrics settings within the **Coverage Metrics** pane of the Info File Editor. There are four settings:

1. **C Statements** Enables or disables tracking of the *C Statement* coverage metric.
2. **C Functions** Enables or disables tracking of the *C Function* coverage metric.
3. **C Function Calls** Enables or disables tracking of the *C Function Call* coverage metric.
4. **Decision metric** This setting determines which expressions in a C program are counted as decisions for tracking Decision, Condition, MC/DC, and MCC coverage. There are three options:
   - Expressions which control if/while/for statements.
   - Expressions which control if/while/for statements or the ? operator.
   - All non-trivial Boolean statements except static initializers.

All of the above metrics are explained in Section 16.5.

### 16.2.2.3 C Code Pane

Reactis for C settings are also available in the **C Code** pane. The pane also displays a list of locations in the current model that reference C code (S-Functions and Stateflow custom code) and gives you the ability to configure associated *.rsm* files.

Each of the window items labeled in Figure 16.4 is interpreted and used as follows:
16.2. CONFIGURING REACTIS FOR C

Figure 16.4: Viewing list of C Code components in a model using the C Code pane of the Info File Editor.

1. **S-Function source code coverage tracking.** This menu item configures the setting for tracking coverage within the C code in S-Functions. This setting may be overridden for an individual S-Function within the .rsm file for the S-Function. The supported settings are:

   - **Off.** Turn off coverage tracking within S-Functions.
   - **Cumulative.** When there are multiple instances of the same S-Function in a model, track coverage cumulatively for all instances.
   - **Separate.** When there are multiple instances of the same S-Function in a model, track coverage separately for the instances.

2. **Create and use cache files for C code.** In order to analyze C code, the Reactis for C Plugin must convert the source file into a Reactis object file (.rso file). Enabling this feature causes .rso files to be stored on disk (in the folder where the C code resides). Subsequently, a C source file will only be recompiled if the C source file is modified, or if a new version of Reactis is installed.

3. **Promote ‘float’ values to ‘double’ to reduce round-off error.** When checked, single-precision floating-point values will always be converted to double-precision prior to any calculation, which reduces round-off error in some cases. For strict C99 compliance, this should be disabled (not checked).

4. **Type.** This indicates the type of C code used in the model. These types are described in Section 16.3 and 16.4.

5. **S-Function Name.** This gives the name of the S-Function used, where applicable. (As discussed in section 16.3 and 16.4, S-Function Libraries, Stateflow code, and Stateflow code libraries do not have associated S-Function names).
6. **RSM File.** This is the Reactis Make file (.rsm) that describes the necessary C files, includes, libraries, search paths and defines (as described in Section 16.3).

7. **Location.** This is the path location of the C code instance. If the type is Stateflow Custom C code, it can be accessed from any Stateflow chart in the model, so here it is labeled 'global'.

By right-clicking on a row of this dialog, you can bring up a menu to add, edit, or remove .rsm files. These functions are the same ones presented when you right-click on an .rsm file in the model hierarchy panel.

Double-clicking on any line with a named S-Function will cause the Reactis main window to jump to that location in the model hierarchy, and briefly highlight that system. Double-clicking on a line whose location is 'global' will only highlight the C code library in the hierarchy panel.

Clicking on each column heading will sort the rows alphabetically according to that column. In the example, some S-Function name and .rsm file columns have "n/a" (not applicable) in various locations. As explained in 16.4, code referenced from Stateflow does not have an associated S-Function, nor is an .rsm file required. Libraries that are referenced from Stateflow or S-Functions require an .rsm file, but do not have separate S-Functions.

---

### 16.3 S-Function C Code

Simulink does not require you to specify the C source files that implement an S-Function; instead only a name is given. For white-box testing, the Reactis for C Plugin requires the C source files that implement an S-Function as well as the source files for any libraries referenced by the S-Function. Since the .slx file does not include this information, the Reactis for C Plugin has a mechanism for listing the C source files which comprise an S-Function. This mechanism is described in the rest of this section.

The collection of C source files that implement an S-Function is specified using a Reactis Make File (.rsm file), which is created and modified by the editor shown in Figure 16.5, Figure 16.7, and Figure 16.8. An .rsm file is created to specify how the Reactis for C Plugin should compile the S-Function. Additionally, a separate .rsm file is constructed for each library referenced by the S-Function. The .rsm files for the S-Function and any libraries referenced by the S-Function are defined in the same way. In the following discussion we denote the library compiled from an .rsm file as the *library under construction* (LUC). The .rsm file editor consists of 5 panes, each of which is described in the following subsections.

#### 16.3.1 Source Files

Figure 16.5(a) shows the Source Files pane of the .rsm file editor, which lists the C source files which implement the LUC. The labeled items in Figure 16.5(a) are used as follows:

1. This button is used to add one or more source files to the LUC (see Section 16.3.1.1).
2. This button converts all currently selected source files to relative path format.
3. This button converts all currently selected source files to absolute path format.
4. This button removes all currently selected source files from the LUC.
16.3. S-FUNCTION C CODE

Figure 16.5: The (a) Source Files and (b) Include Search Path panes of the .rsm file editor.

6. These buttons let you reorder items in the list.
7. This button opens the help dialog.
8. This button generates stub source code for missing functions and variables.
9. This button builds the LUC.
10. This button closes the .rsm file editor and saves any pending modifications.
11. This button closes the .rsm file editor without saving any pending modifications.

16.3.1.1 Adding Source Files

When adding source files to an .rsm file, you may choose to either (1) enter the name of a file directly, (2) locate the file by browsing the filesystem, or (3) use the batch add facility, as shown in Figure 16.6. The labeled items in Figure 16.6 are used as follows:

1. This button opens the Select All Source Files in Folder dialog.
2. The directory containing the source files is entered here.
3. This button opens a browser dialog which is used to select the directory containing the source files (item 2).
4. When checked, only those source files which match one of the file selection patterns (item 5) will be selected.
5. File selection patterns are entered here. Within a pattern, asterisk (*) matches any (possibly empty) sequence of characters and semicolon (;) separates patterns. A filename is a match if it matches any of the listed patterns. For example, the pattern sequence a*.c;b*.c will select any filename which starts with a or b and has extension .c.
6. When checked, subfolders will be searched for matching files.
7. When checked, subfolders which match one of the directory exclusion patterns (item 8) will not be searched.

8. Directory exclusion patterns are entered here. Within a pattern, asterisk (*) matches any (possibly empty) sequence of characters and semicolon (;) separates patterns. A directory is excluded if it matches any of the listed patterns. For example, the pattern sequence ".;slprj" will exclude any directory which starts with a period or is named slprj.

### 16.3.2 Include Search Path

Figure 16.5(b) shows the Include Search Path pane of the .rsm file editor, which lists the directories to be searched when processing a #include directive. The labeled items in Figure 16.5(b) are used as follows:

1. This button is used to add one or more directories to the LUC.
2. This button converts all currently selected directories to relative path format.
3. This button converts all currently selected directories to absolute path format.
4. This button removes all currently selected directories from the LUC.
5. These buttons are used to move the currently selected directories up or down in the list. The list is searched from top to bottom when processing a #include directive.
6. This button opens the help dialog.
7. This button generates stub source code for missing functions and variables.
8. This button builds the LUC.
9. This button closes the .rsm file editor and saves any pending modifications.

10. This button closes the .rsm file editor without saving any pending modifications.

Figure 16.7: The (a) Libraries and (b) Defines panes of the .rsm file editor.

16.3.3 Libraries

Figure 16.7(a) shows the Libraries pane of the .rsm file editor, which lists the libraries used by the LUC. The labeled items in Figure 16.7(a) are used as follows:

1. This button is used to add one or more libraries to the LUC.

2. This button opens the currently selected library in a new instance of the .rsm file editor. The current editor instance is suspended until the new instance of the editor is closed.

3. This button converts all currently selected libraries to relative path format.

4. This button converts all currently selected libraries to absolute path format.

5. This button removes all currently selected libraries from the LUC.

6. This button opens the help dialog.

7. This button generates stub source code for missing functions and variables.

8. This button builds the LUC.

9. This button closes the .rsm file editor and saves any pending modifications.

10. This button closes the .rsm file editor without saving any pending modifications.
16.3.4 Defines

Figure 16.7(b) shows the Defines pane of the .rsm file editor, which lists the macro definitions which are to be defined at the start of compilation of the LUC. Macros are defined using expressions of the form:

- \textit{name=value}, in which case instances of the identifier \textit{name} will be replaced by \textit{value} during preprocessing, or
- \textit{name}, in which case \textit{name} will be replaced by the literal 1 during preprocessing.

There is no mechanism provided to define macros which accept arguments or use macro operators such as stringification or token catenation. This is similar to the way the -D option works on most C compilers.

The labeled items in Figure 16.7(b) are used as follows:

1. This button is used to add one or more macro definitions to the LUC.
2. This button opens a editor from which the currently selected macro definition can be changed.
5. This button removes all currently selected macro definitions from the LUC.
6. This button opens the help dialog.
7. This button generates stub source code for missing functions and variables.
8. This button builds the LUC.
9. This button closes the .rsm file editor and saves any pending modifications.
10. This button closes the .rsm file editor without saving any pending modifications.

16.3.5 General

Figure 16.8 shows the General pane of the .rsm file editor, which contains a few general settings for the LUC. The labeled items in Figure 16.8 are used as follows:

1. This button makes the LUC a static library. \textit{Static library} is the default and recommended setting for libraries. This setting can only be changed when the library is first created.
2. This button makes the LUC a dynamic library.
3. If (2) is selected, this checkbox makes the pointers used to implement dynamic linking visible, and may be useful in cases where code has been written that directly accesses the pointers to library objects.
4. If (3) is selected, the prefix used for dynamic library pointers will appear here and can be edited if desired.
5. This pulldown menu determines the coverage tracking for the library. The possible choices are:
16.3. S-FUNCTION C CODE

Inherited. Inherit the setting from the S-Function source code coverage tracking setting in the C Code pane of the Reactis Info File Editor. Note that the Inherited option is only available for S-Function .rsm files.

Off. Turn off coverage tracking within the LUC.

Cumulative. When there are multiple instances of the LUC in a model, track coverage cumulatively for all instances.

Separate. When there are multiple instances of the LUC in a model, track coverage separately for the instances. Note that the Separate option is only available for S-Function .rsm files.

6. This button opens the help dialog.

7. This button generates stub source code for missing functions and variables.

8. This button builds the LUC.

9. This button closes the .rsm file editor and saves any pending modifications.

10. This button closes the .rsm file editor without saving any pending modifications.

16.3.6 Pathnames

The path to each source file or library can be either relative or absolute. The location of a source file whose path is relative is determined by appending the relative path to the directory containing the .rsm file. For example, if the .rsm file C:\models\cruise\cruise.rsm references
the source file `..\csrc\ramp.c`, the absolute path to `ramp.c` will be `C:\models\csrc\ramp.c`. For convenience, the .rsm file editor provides buttons for converting paths from relative to absolute form, and vice-versa. To use one of these buttons, first select a source file or library by clicking on it, and then click on either the Relative or Absolute button to the right of the window containing the source file name or library name.

16.3.7 S-Function Requirements

All source code (for both an S-Function and any referenced libraries) must be available. In addition, all code must satisfy the requirements listed in Section 14.2.1.

16.4 Stateflow Custom C Code

![Diagram of a Stateflow model with custom C code highlighted.](image)

Figure 16.9: A model containing Custom C Code before an .rsm file has been specified for `ctest10_code_lib.lib`.

The Reactis for C Plugin will automatically determine which C source files are called from Stateflow. This is possible because Stateflow requires you to specify the C source files in the Target Options dialog and stores the names of the files in the .slx file for a model. The Reactis for C Plugin automatically processes any custom C code files. User input is only required in cases where custom C code links to external libraries.

When a model is opened, the C files called from Stateflow will be listed at the bottom of the Reactis hierarchy panel in the C Libraries section, subsection Stateflow User Sources. This is illustrated in Figure 16.9. In this case, the Reactis for C Plugin has determined that the file `ctest10_code.c` is called from Stateflow and has added it to the tree of model components, along with the library `ctest10_code_lib.lib`. Note the error icon next to `ctest10_code_lib.lib`. This icon indicates that user input is required to tell the Reactis for C Plugin what sources belong to the library.

---

1 The Target Options dialog is opened from the Stateflow window by selecting Tools → Open Simulation Target... and then clicking the Target Options button.
After selecting `ctest10_code_lib.lib`, the message shown in Figure 16.10 is displayed. This message indicates that it will be necessary to create an `.rsm` file because the `.slx` file does not list the sources used to build `ctest10_code_lib.lib`. To specify the sources, right-click on the library model name in the hierarchy panel (in this case `ctest10_code_lib.lib`) and select `Assign RSM File`. A dialog window similar to the one in Figure 16.11 will appear. The process of creating the `.rsm` file is nearly the same as the process for creating S-Function `.rsm` files discussed in section 16.3. The only two differences are (1) that the editor window does not contain a library section, and (2) there is no coverage selection dialog. No coverage selection dialog is presented because coverage data in custom C code called from Stateflow is always collected cumulatively.

Figure 16.10: The Reactis for C Plugin requesting information about the source code used to compile a library.

Figure 16.11: Creating an `.rsm` file for a library used by custom C code called from Stateflow.
16.4.1 Stateflow Custom Code Requirements

The Reactis for C Plugin supports analysis of Stateflow custom code for which all C source code is available. The source files need not necessarily reside in the same directory as the model. Reactis will automatically search the directories in the Reactis path if a custom code source file or library is not found in the model directory. This behavior emulates the behavior of MATLAB. Regardless of what directory a library is found in, the .rsm file editor will start in the model directory when assigning the .rsm file associated with the library. If the .rsm file does not yet exist, the file system tree displayed as part of the open dialog window can be used to select a directory other than the model directory in which to create the .rsm file. For example, it could be desirable to create the .rsm file for a library which shared between models in the directory containing the source files used to build the library.

16.5 C Code Coverage Metrics

Reactis uses a number of different coverage metrics to measure how thoroughly a test or set of tests exercises a model. In general, coverage metrics record how many of a given class of coverage targets (model features such as Stateflow states, C program statements, etc.) have been executed at least once. Coverage metrics may be visualized using Simulator and are central to test generation and model validation using Tester and Validator.

Chapter 6 describes the coverage metrics that are tracked within the Simulink and Stateflow portions of models. Those metrics include some which are specific to Simulink, some which appear only in Stateflow, and others which are generic and can appear in either Simulink or Stateflow. Three of the generic metrics are also tracked in the C code portions of models by the Reactis for C Plugin. These are decision coverage, condition coverage, and modified condition / decision coverage (MC/DC).

**Decision coverage** tracks whether each decision in a program has evaluated to both true and false. The program elements that Reactis for C identifies as decisions can be configured via the Decision metric setting in the C Code tab of the Info File Editor (item 4 in Figure 16.3). There are three possible Decision metric settings:

**Expressions which control if/while/for statements.** Only expressions which appear within the parentheses following the if, while, or for keywords are decisions. Boolean expressions appearing in other contexts, such as the expression \( x < 100 \) in the assignment statement \( b = x < 100 \), are not decisions.

**Expressions which control if/while/for statements or the ? operator.** In addition to expressions which appear within the parentheses following the if, while, or for keywords, the first argument of the C ? operator is also a decision. For example, in a program which contains the statement \( x = y<0? -y : y \), \( y < 0 \) is a decision.

**All non-trivial Boolean expressions except static initializers.** Under this criterion, any Boolean expression which does not appear in a static initializer and also is not a simple assignment of one Boolean variable to another, is a decision. For example, if a function \( f \) contains the assignment \( b = x < 100 \) (where \( b \) is a local variable of \( f \)), then \( x < 100 \) is a decision.

Note that the argument of a switch statement is not a decision.
**Condition coverage** tracks whether each condition in a program has evaluated to both true and false.

For the definition of MC/DC coverage, please see Chapter 6.

In addition to the generic coverage metrics discussed above, the Reactis for C Plugin also tracks three C-specific coverage metrics. These are:

**C Statements.** This metric has one target for each statement in a C program. A target is covered when the corresponding statement has been executed at least once.

**C Functions.** This metric has one target for each function in a C program. A target is covered when the corresponding function has been executed at least once.

**C Function Calls.** This metric has one target for each function call expression in a C program. A target is covered when the corresponding function call expression has been executed at least once.

### 16.5.1 Visualizing Coverage in C Code

When **Coverage → Show Details** in Simulator is selected, unexercised coverage targets in C code are reported visually as shown in Figure 16.12. Any unexecuted C statement is rendered in red. If a function has not been called, its name is rendered in red. If a function call expression has not been executed, the name of the function within the call expression is rendered in red.

If a decision has not evaluated to true it has a thin red overline. If a decision has not evaluated to false it has a thin red underline. If a condition has not evaluated to true it has a thick red overline. If a condition has not evaluated to false it has a thick red underline. If a decision has not met the MC/DC or MCC criteria, then the text of the decision is displayed in red. MC/DC and MCC coverage details associated with a decision (Figure 16.13) may be displayed by right clicking on the decision and selecting **View Coverage Details**. For a description of this dialog, please see Chapter 6.

Lines containing unexercised targets are distinguished by a red bar which is drawn just to the right of the line number of that line.

### 16.6 C Code in the Main Reactis Window

As shown in Figure 16.14, an `.rsm` file may be assigned, edited, or removed by right-clicking on the S-Function in the main panel when Simulator is disabled. After an `.rsm` file is assigned, the C source files for an S-Function are listed in the hierarchy panel and RSM appears in the upper right corner of the S-Function in the main panel.

Figure 16.15 shows that a C file may be displayed in the main panel by clicking on the filename in the hierarchy panel. Line numbers and the grid may be turned on or off from the Reactis for C tab of the Global Settings dialog.

### 16.7 C Code and Reactis Simulator

When the Reactis for C Plugin is enabled, Simulator seamlessly steps into Stateflow C code and S-Functions that have `.rsm` files assigned. The Simulator controls in the tool bar work in the same way within C code as they do for the Simulink and Stateflow portions of models.
Figure 16.12: Coverage highlighting in C code.

Figure 16.13: View the MC/DC-related coverage details associated with a decision by right-clicking on the decision and selecting View Coverage Details.

Five additional buttons are available to step through C source code. The behavior of those five and the Step Into button (also available when stepping through Simulink / Stateflow) is described below:

- **Reverse Step Out Of.** Step backwards to just before the currently executing function was called.

- **Reverse Step Over.** When paused after a function call, clicking this button steps back to just before the function was called.
Figure 16.14: Assigning .rsm files to S-Functions.

Figure 16.15: Clicking on a C filename in the hierarchy panel displays the C file in the main panel.
Reverse Step Into. Step backwards one statement. If a function call just returned, step back to the last statement of the function.

Step Into. Executes until either the current line of source code completes or a function is called.

Step Over. Executes until the current line of source code completes, including all function calls performed during its execution.

Step Out Of. Executes until the current function completes.

Other buttons continue to operate the same as when not using the Reactis for C Plugin. For example, the step button ( ▼ ) reads all top-level inputs of the model and then executes the blocks (possibly including C code), updating the top-level outputs of the model with the results of executing the step.

As shown in Figure 16.16, a breakpoint may be toggled on lines within a C file which contain an executable C statement by double-clicking just to the right of the line number, or by right-clicking to the right of the line number and selecting Toggle Breakpoint. When trying to set a breakpoint in C code on a line that does not contain an executable statement, Reactis remaps the breakpoint to a different line according to the following rules:

- If the attempted breakpoint line is within a multi-line executable statement, the breakpoint is placed at the first line of multi-line statement.
- If the attempted breakpoint line is on a line with no executable statement (e.g. a comment or a variable declaration), then the breakpoint is placed on the first subsequent line that contains an executable statement.

Note that the following are not executable statements:

- type declarations,
- function prototypes,
- static variable declarations with or without an initializer,
- declarations of uninitialized local variables,
- preprocessor commands, and
- lines which are empty or only contain comments.

When execution reaches a line containing a breakpoint, execution will he paused before the line is executed. While paused, hovering over many identifiers in the source code will cause information about the identifier to be displayed. The information displayed depends on the type of the identifier:

- If the identifier is an instance of a macro, the expansion of the macro will be displayed.
- If the identifier is a variable, information about the type and value of the variable will be displayed. In addition, pertinent source code locations, such as the location where the variable was last modified, will be displayed.
### 16.8 C Code and Reactis Tester

When the Reactis for C Plugin is enabled, Tester automatically tries to exercise the coverage targets (described in Section 16.5) in Stateflow custom C code and S-Functions that have `.rsm`

---

<table>
<thead>
<tr>
<th>Type</th>
<th>sizeof</th>
<th>alignof</th>
</tr>
</thead>
<tbody>
<tr>
<td>char, bool, void</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>short int</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>int, long int, float</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>long long int, double, long double*</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

*long double is not currently supported – it is mapped to type double.

Table 16.1: Sizes and alignments of the common C types, in bytes.

- If the identifier is a function, the function type, the argument types and values, the location where the function is defined and the current call stack will be displayed.
- If the identifier is a user-defined type, the type details will be displayed.
- If the identifier is a pointer, the value pointed to is displayed.

For a description of how Simulator visually presents coverage information please see Section 16.5.1.

---

The sizes and alignments of the common built-in C types under the Reactis for C Plugin are listed in Table 16.1. These match the behavior of GCC (GNU C Compiler) and most other x86 C compilers. Integers are stored using a little endian byte order (the least-significant byte is stored first).
files assigned. The launch dialog includes a check-box to enable or disable C statement coverage.

16.9 C Code and Reactis Validator

Reactis Validator can check requirements on top-level S-Function inputs and outputs. To do so, Validator objectives are inserted within the subsystem which contains the S-Function. The Reactis for C Plugin does not currently support the insertion of Validator objectives within the C code.

16.10 Integer Overflows

The C-Plugin supports detection and flagging of overflows during operations on integer values. This feature can be turned on or off by changing the value of the On integer overflow configuration parameter, as discussed in Section 16.2.

When On integer overflow is set to Wrap Over, the C-Plugin emulates the behavior of typical hardware, in which the most significant bits of a value which is too large are dropped until the value is small enough to be represented. The result of an unsigned wrapped-over overflow is typically equal to the large result modulo $2^n$, where $n$ is the width (in bits) of the container type.

When signed values overflow, the sign bit may change, producing results much smaller or larger than if no overflow occurred. For example, in most 32-bit environments, adding the maximum signed int value ($0x7fffffff$) to itself produces the wrapped-over result $-2$.

When On integer overflow is set to Error, the C-Plugin will stop execution and display an error dialog window whenever an integer computation overflows. When a signed integer addition, subtraction, multiplication or left shift operation is performed, the result is considered to have overflowed when it is too large or too small to be represented.

Note that according to the C language specification, no overflow occurs during unsigned computations. Instead, the result is computed by wrapping over the large value, using the modulo approach mentioned in the previous paragraph. This has the effect of truncating (in binary) the larger value. For example, assigning the value $0x1234u$ to an unsigned char variable is the same as assigning the value $0x34$.

When a value of one integer type is converted to a value of a different integer type, an overflow can only occur when the destination type is signed. This means that the statements

```c
int x = -1;
unsigned int y = x;
```

will not produce an overflow. Instead, $y$ will receive the value $0xffffffff$. However, if a third statement is added:

```c
int x = -1;
unsigned int y = x;
x = y;
```

then an overflow will occur, because the value of $y$ ($0xffffffff$) is larger than the maximum positive value that can be represented by $x$ ($0x7fffffff$). To avoid such errors while
catching arithmetic overflows, set \textit{On integer conversion overflow} to \textit{Do nothing}. Alternatively, the following macro can be used to convert an unsigned int to an int without causing an overflow:

```
#define UINT_TO_INT(x) \ 
  (((x) <= INT_MAX)? (int)(x) : \ 
    (((int)((x)^INT_MIN))+INT_MIN))
```

The values \texttt{INT\_MAX} and \texttt{INT\_MIN} are defined in the system header file \texttt{limits.h}.

The C-Plugin uses C99 rules for determining the signedness of expressions, which affects when overflows may occur. For integer constants, the type of the constant is determined by radix of the constant and the presence of a \texttt{U} or \texttt{u} suffix, as determined by the following rules:

1. Any integer constant with a \texttt{U} or \texttt{u} suffix is unsigned.
2. Any decimal constant without a \texttt{U} or \texttt{u} suffix is signed.
3. Any non-decimal constant \texttt{X} without a \texttt{U} or \texttt{u} suffix will be assigned the first type that can represent \texttt{X} from the following list: int, unsigned int, long long int, unsigned long long int. This means that \texttt{0xffffffff} will be assigned type unsigned int and \texttt{4294967295} will be assigned type long long int even though the two values are equivalent. In the latter case, if the literal is changed to \texttt{4294967295U}, it will be assigned type unsigned int.

The C99 rules for evaluating integer expressions state that values of any type smaller than \texttt{int}, including unsigned types, are first converted to type \texttt{int} before any subsequent evaluations are performed. This forces values of type unsigned short and unsigned char to become signed, so that, for example, adding two variables of type unsigned short will produce a signed int result.

When two integer values are combined in a binary expression, the width and signedness of the result are determined by two rules:

1. If one type is larger than the other, then the larger type is chosen.
2. If both types have the same size, then the result is unsigned if either type is unsigned.

Operator arguments are converted to the result type before the expression is evaluated. This means that the expression \((\texttt{int}(-1)) > (\texttt{unsigned int}1)\) will evaluate to 0 (false).

---

16.11 Memory Errors

Whenever a pointer is used to access memory, the Reactis for C Plugin performs a safety check to ensure that pointer is valid. There are two steps in the validity check:

1. \textit{Spatial validity}. A pointer is not allowed to access memory outside of the bounds of the data object it originally pointed to (henceforth called the \textit{referent}). For example, a pointer to an array can only be used to access memory within the bounds of the array. If a pointer is dereferenced and it points outside the bounds of its referent, then a \textit{spatial memory error} occurs.
2. **Temporal validity.** The referent of a pointer must not have been deallocated prior to an access attempt. Pointers to heap objects are no longer temporally valid once the referent has been deallocated by a call to `free()`. Pointers to local variables of a function are no longer valid once the function has returned. Data stored in static memory, such as variables declared outside of a function scope are never deallocated and are hence always temporally valid. Attempts to access a temporally invalid referent result in a *temporal memory error*.

Note that it is possible that a program which appears to function correctly actually contains a memory error. For example, a function that sums an array may have an error in the iteration termination condition which causes the function to read 1 element past the end of the array. If the element which follows the end of the array happens to always be zero, this error may have never been detected. At some point in the future, a change to the program may cause the invalid memory access to return a non-zero value. Even worse, there may be rare runtime conditions under which the memory access returns a non-zero value, causing intermittent program malfunctions which are difficult to reproduce and diagnose. Hence it is best to detect and fix such errors even though they may not seem to be presently causing a problem.

### 16.11.1 Invalid Pointer Creation

The Reactis for C Plugin can produce an error or warning whenever a pointer expression produces an invalid pointer. This can help determine the source of a memory error. However, there are cases where invalid pointers are produced which are never dereferenced, such as the following program fragment:

```c
int A[10], B[10], i, *a = A, *b = B;
while (i < 10) {
    *(a++) = *(b++);
}
```

During the last iteration of the above loop, the pointers `a` and `b` are both assigned invalid pointer values. This is not a problem because the invalid pointer values are never used. For such programs, the *On invalid pointer creation* setting should be set to *Do nothing* or *Produce warning*.

### 16.12 Other Runtime Errors

In addition to memory errors and overflows, the Reactis for C Plugin also detects the errors listed below. The C standard states that the results of these operations are undefined.

- **Divide by zero.** An attempt to compute the quotient or remainder of a fraction in which the divisor is zero.

- **Invalid shift.** An attempt to shift a value by an amount which is either (a) greater than or equal to the width of the value, or (b) a negative amount.
16.13 Stub File Generation

The .rsm file editor includes a stub file generator, which produces a source file that contains definitions of missing functions and variables. The stub generator is launched by clicking on the Stub button. Note that the generated stub file initially does not contain any type definitions and will not compile until all required type definitions have been added to the file, which is typically done by adding #include statements to the stub file with a text editor.

16.14 Unsupported C Features

The Reactis for C Plugin supports a very large subset of the C99 standard. The following features, however, are not currently supported:

- Variable-length arrays (e.g., int f(int n) { char s[n]; }).
- ANSI Trigraphs.
- Universal character codes (a \u or \U escape which can be followed by a UCS-2 or UCS-4 hex code).
- Signals.
- Types _Complex and _Imaginary.
- Type long double.
- Wide character types.
- File I/O during simulation.
- Designated initializers (e.g., struct { int x; int y; } s = { .y = 5 };).
- Non-constant static initializers (e.g., static int A[1] = { f(x); });).

The following restrictions apply to white-box analysis of S-Functions:

- All general restrictions for S-Function support as mentioned in Section 14.2.1.
- MATLAB external interface functions dealing with sparse matrices (mxCreateSparse, mxCreateSparseLogicalMatrix, mxGetIr, mxGetJc, mxSetIr and mxSetJc) are not supported.
- MATLAB external interface functions dealing with imaginary data (mxGetImagData, mxGetPi, mxSetImagData, mxSetPi) are not supported.

In Stateflow custom C code, pointers to functions are not currently supported. The following runtime library functions are currently unimplemented:

complex.h
   cabsf cargf cimagf crealf cabs cimag creal cabsf cargl cimagl creall cacoshf casinhf catanhf ccoshf cexpf conjf cpowf cprojf csinf csinhf csqrtf ctanf ctanhf cacos cacosf cacosl cacosl
ctan ctanh cacosl casinl casinhl catanl catanhl ccosl ccoshl cexp1 clogl conjl cpowl cprojl
csinl csinhl csqrtl ctanl ctanh

dlfcn.h
dlopen dlslm dlclose dlerror

cntl.t.h
creat cntl open close read write lseek unlink

math.h
asinh asinhf asinhl acosh acoshf acoshl atanh atanhl erf erff erfl erfc erfcf erfl fam
fmaf faml hypot hypotf hypotl lgamma lgammaf lgammal lgammaf lgammal expm1
expm1f expm1l ilogb ilogbf ilogbl log1p log1pf log1pl rint rintf rintl

setjmp.h
sigsetjmp siglongjmp

signal.h
raise

stdio.h
freopen fseek ftell rewind fgetpos fsetpos ungetc fscanf scanf vfprintf vscanf vscanf vs-
scanf fread fwrite feof ferror clearerr rename remove tmpfile tmpname

stdlib.h
atexit system execl execlp execle execvp execve bsearch qsort

unistd.h
access alarm brk chdir chroot chown close confstr crypt ctermid cuserid dup dup2 encrypt
fchown fchdir fdatasync fork fpathconf fsync ftruncate getcwd getdtablesize getegid geteuid
getgid getgroups gethostid getlogin getlogin_r getpagesize getpass getpgid getpgrp getsid
getws getuid getwd isatty lchown link lockf lseek nice pathconf pause pipe pread
pthread_atfork pwrite read readdir remdir sbk atfork pwrite read readdir rmdir setgid setgid
setpgid setpgid setregid setreuid setsid setuid sleep swab symlink sync sysconf tcgetpgrp
tcsetpgrp truncate ttyname ttyname_r ualarm unlink usleep vfork write

utime.h
utime

wchar.t.h
wcscat wcsncat wcscmp wcsncmp wcsncpy wcslen wcschr wcsrchr wcspn wcsp-
spn wcpbrk wcstok wcslstr wcsxfrm wmemchr wmemcmp wmemmove
wmemset wmemset wchar getwc getwchar ungetwc fgetwc fgetws wcsanf wcsanfl wcsanf fputwc putw-
cputwchar fputws fwprintf wprintf vfprintf vscanf vswprintf vswscanf vswscanf

wctype.h
mbrlen bto wc mbtowc wcscnt wctomb mbsinit isalnum isalpa iswcntrl iswdigit iswxdigit
iswgraph iswpunct iswprint iswlower iswupper iswspace towlower towupper wctype isw-
type wctrans towtrans
16.15 Supported C library functions

The following C library functions are currently provided by the Reactis for C Plugin.

**assert.h**
- assert

**ctype.h**
- isalnum
- isalpha
- isascii
- iscntrl
- iscsym
- iscsymf
- isdigit
- isxdigit
- isprint
- isgraph
- ispunct
- islower
- isupper
- isspace
- isblank
- toint
- tolower
- toupper

**errno.h**
- strerror

**fenv.h**
- fegetround
- fesetround

**math.h**
- acos
- sqrt
- sin
- cos
- tan
- sinh
- cosh
- tanh
- asin
- acosl
- asinf
- atan
- atan2
- exp
- exp2
- pow
- ln
- log
- log2
- log10
- sinh
- cosh
- tanh

**memory.h**
- bcopy
- memccpy

**setjmp.h**
- longjmp
- setjmp

**stdarg.h**
- va_start
- va_arg
- va_copy
- va_end

**stddef.h**
- offsetof

**stdlib.h**
- malloc
- calloc
- realloc
- free
- abs
- labs
- llabs
- div
- ldiv
- lldiv
- atof
- atoi
- atol
- atoll
- strtof
- strtod
- strtold
- strtol
- strtoul
- strtoull
- exit
- _Exit
- abort
- getenv
- bsearch
- qsotr
- mblen
- mbtowc
- mbstowcs
- wcstombs
- rand
- srand

**string.h**
- memchr
- memcmp
- memmove
- memset
- strlen
- strchr
- strcpy
- strncpy
- strcat
- strncat
- strcmp
- strncmp
- strspn
- strcspn
- strpbrk
- strtok
- strcoll
- strxfrm
- strdup
- strndup

**tgmath.h**
- sin
- cos
- tanh
- sinh
- cbrt
- ln
- log
- log2
- log1p
- fabs
- ceil
- floor
- nearbyint
- rint
- lrint
- round
- lround
- llround
- floor
- round
- trunc
- atan2
- pow
- copysign
- fdim
- hypot
- fmin
- fmax
- fmod
- remainder
- nextafter
- nexttoward
- frexp
- ldexp
- modf
- scalbn
- scalbnl
- nrmquo
Chapter 17

Reactis for EML Plugin

The V2015 release of Reactis included a new product the Reactis for EML Plugin, which integrates with Reactis to offer white-box testing of the Embedded MATLAB (EML) portions of a model. When using the plugin, Reactis Tester will generate tests which attempt to exercise targets in EML code and Reactis Simulator will track and display the coverage status of EML code. The V2016 release of Reactis extends Simulator functionality to include basic debugging of EML code, including breakpoints, single-stepping, and the ability to view variable values.

Note that if not using the Reactis for EML Plugin, you can still test models with EML code, but the EML portions of a model will be treated as block boxes. They will be executed but coverage within EML code not be tracked or targeted by generated tests, and EML code will not be visible within Reactis Simulator.

EML can be introduced into a model in several ways:

- **Simulink**
  - MATLAB Function block
  - Truth Table block
- **Stateflow**
  - MATLAB functions
  - Truth Table functions with MATLAB language option

Note that a subset of the full Embedded MATLAB language is supported. To use models containing EML in Reactis, your model must only use the language subset supported by Reactis. Note that the same language restrictions apply no matter which of the four model constructs listed above are used to incorporate EML into your model. To avoid confusion we use the following abbreviations in the following discussion:

- **Embedded MATLAB (EML)** the subset of MATLAB supported by MathWorks for code generation.
- **Reactis Embedded MATLAB (REML)** the subset of EML supported by Reactis.

REML is under active development toward the ultimate goal of supporting a very large subset of EML. Please send requests to Reactive Systems (help@reactive-systems.com) if there are unsupported EML features you would like use.

Whether in white-box or black-box mode for EML, Reactis supports the same subset of EML. This subset is defined in Section 17.4.
17.1 Enabling the Reactis for EML Plugin

Enable the plugin by selecting File → Global Settings..., selecting the Reactis for EML tab, and then in the resulting dialog (Figure 17.1) select Enable white-box analysis of Embedded MATLAB code (requires license for EML Plugin).

![Figure 17.1: The Settings dialog with tab Reactis for EML selected.](image)

The other controls in this tab let you toggle the display of line numbers in the main panel, set the background color of the line number bar, and toggle and set the color for drawing a grid (alternating the background color of each line) in the main panel.

As shown in Figure 17.2, after enabling the EML Plugin, the EML code contained in a model element will be displayed in the Reactis main panel when the element is selected in the hierarchy panel. Since Simulink stores the EML code within the model’s .slx file, no special configuration is required within Reactis. Reactis automatically extracts the EML code from the .slx file. If this EML code from the .slx file calls external functions residing outside the .slx file in separate .m files, then the external functions should be identified within Reactis as described in Section 17.2.
17.2 External EML Functions

Simulink supports calling functions stored in .m files external to a model. To call such a function from a MATLAB Function block (or any EML embedded in the model e.g. Stateflow, Truth Tables, etc.) the .m file must reside in a folder which is included in the MATLAB path.

Reactis also supports external EML functions, but you must identify the .m files containing external functions called from your model. You enumerate the external EML functions from the External EML Functions pane of the Reactis Info File Editor (Opened by selecting Edit → External EML Functions from the Reactis main window). After you list the functions here, Reactis lists them in the External EML Functions section of the hierarchy panel as shown in Figure 17.3. Additionally, Reactis will identify coverage targets withing the external functions, attempt to exercise those targets when generating tests, and allow you to step into the external functions when executing your model in Reactis Simulator.

17.3 EML Coverage Metrics

Reactis uses a number of different coverage metrics to measure how thoroughly a test or set of tests exercises a model. In general, coverage metrics record how many of a given class of coverage targets (model features such as Stateflow states, EML program statements, etc.) have been executed at least once. Coverage metrics may be visualized using Simulator and are central to test generation and model validation using Tester and Validator.

Chapter 6 describes the coverage metrics that are tracked within the Simulink and State-
flow portions of models. Those metrics include some which are specific to Simulink, some which appear only in Stateflow, and others which are generic and can appear in either Simulink or Stateflow. Three of the generic metrics are also tracked in the EML code portions of models by the Reactis for EML Plugin. These are decision coverage, condition coverage, and modified condition / decision coverage (MC/DC).

**Decision coverage** tracks whether each decision in a program has evaluated to both true and false. The program elements that the Reactis for EML Plugin identifies as decisions are the conditional expressions in if statements and while loops.

**Condition coverage** tracks whether each condition (atomic Boolean expression) in a decision has evaluated to both true and false.

For the definition of MC/DC coverage, please see Chapter 6.

Finally, in addition to these generic coverage metrics, the Reactis for EML Plugin also tracks statement coverage – whether or not each EML statement has been executed at least once.

### 17.3.1 Tracking Coverage in EML Code

In Reactis Simulator, when **Coverage → Show Details** is selected, unexercised coverage targets in EML code are reported visually as shown in Figure 17.4. Any unexecuted EML statement is rendered in red. If a decision has not evaluated to true it has a thin red overline. If a decision has not evaluated to false it has a thin red underline. If a condition has not evaluated to true it has a thick red overline. If a condition has not evaluated to false it has a thick red underline. If a decision has not met the MC/DC criteria, then the text of the decision
17.3. EML COVERAGE METRICS

is displayed in red. The MC/DC-related coverage details associated with a decision (Figure 17.5) may be displayed by right clicking on the decision and selecting View Coverage Details. For a description of this dialog, please see Chapter 6.

Lines containing unexercised targets are distinguished by a thin red bar which is drawn just to the right of the line number of that line.

Figure 17.4: Coverage highlighting in EML code.

Figure 17.5: View the MC/DC-related coverage details associated with a decision by right-clicking on the decision and selecting View Coverage Details.

17.3.2 Debugging EML Code

When the EML Plugin is enabled, Simulator will step seamlessly into EML code, as shown in Figure 17.6. The numbered items in Figure 17.6 are the primary debugging operations supported by Simulator. These operations are:

1. **Breakpoints.** A breakpoint may be toggled on lines within EML code which contain an executable statement by double-clicking on the line number, or by right-clicking on the
line number and selecting Toggle Breakpoint.

2. **Single Stepping.** EML code may be stepped through one statement at a time by clicking on the mini-step ( ⌈ ) button or by selecting Simulate → Step Into.

3. **Hovering.** When execution is paused within EML code, hovering on a variable causes the value of the variable to be displayed. If execution has not yet passed the point where the variable was initialized, no value is displayed.

When execution is stopped between simulation steps, hovering on a coverage target in EML code causes the coverage status to be displayed.

4. **Watched Variables.** When execution is paused within EML code and a variable is active, the variable may be added to the watched variable panel by right clicking on the name of the variable and selecting Add to Watched.

### 17.4 Subset of Embedded MATLAB Supported by Reactis

This section describes the subset of EML that constitutes REML. REML supports the following aspects of EML. Note that in some cases not all aspects of a feature are supported. For example, a function might not support all numbers and types of input arguments.

#### 17.4.1 Syntax

While REML generally supports the rich EML notation for defining matrices, REML does impose some syntax restrictions. The most prominent restriction is that if any element of a row
is a non-trivial expression, then the row must be delimited with commas instead of spaces. For example, if \( a = 5 \) and \( b = 4 \) then \([ a + b -4 -3 ]\) is a valid EML matrix which evaluates to \([ 9 -4 -3 ]\). The REML syntax does not allow this expression. Instead it requires it to be written \([ a + b, -4, -3 ]\).

### 17.4.2 Types

- double, single, uint8, int8, uint16, int16, uint32, int32, logical, string \(^1\), enumerations \(^2\), structures \(^3\)

### 17.4.3 Control flow

- if statements
- while loops
- for loops
- switch statement
- break statement
- continue statement
- return statement

### 17.4.4 User-defined functions, Subfunctions, and External Functions

- local and persistent variables

### 17.4.5 Array indexing

- scalar indexing, e.g. \( x(4) \)
- vector indexing, e.g. \( x(2:5) \) or \( x(2:end) \) or \( x(:,3) \)

### 17.4.6 Logical operators

- \&, |, \&\&, ||, ~
- and, or, xor, not, any, all

### 17.4.7 Relational operators

- <, <=, ==, ~=, >=, >

### 17.4.8 Math operators

- +, -, *, /, ^

---

\(^1\)Support for strings is currently very limited.

\(^2\)defined by Simulink.IntEnumType classes

\(^3\)Simulink.Bus inputs or outputs
17.4.9 Math functions

- sin, cos, tan
- csc, sec, cot
- asin, acos, atan, atan2
- exp, log, log10
- sinh, cosh, tanh
- sqrt
- fix, floor, ceil, round
- mod, rem
- sign, abs, deg2rad

17.4.10 Statistical functions

These functions support vector and matrix inputs, but not higher dimensional arrays.

- mean, var, std

17.4.11 Matrix operations

- transpose (\( \) )
- addition, subtraction
- multiplication, division
- diag, expm, eye, inv, norm, ones, zeros, repmat, rcond, cond
- \( A^n \) if \( A \) is a square matrix and \( n \) is an integer

17.4.12 Element-wise array operations

- .*, ./, .\, .^

17.4.13 Vector operations

- dot, length, max, min, polyfit, polyval, prod, size, sort, sum

17.4.14 Bit operation

- bitset, bitget, bitand, bitor, bitxor, bitcmp, bitshift

17.4.15 Cast operations

- double, single, uint8, int8, uint16, int16, uint32, int32, logical
17.4.16 String operations

- strcmp

17.4.17 Miscellaneous functions

- class
- true, false
- intmin, intmax
- isempty, isequal, isfinite, isinf, isnan
- struct
- erf
- eps
Chapter 18

The Reactis API

The Reactis Application Programming Interface (API) enables users to access much of the tool's functionality from MATLAB or from C programs. This chapter provides a brief introduction to the use of the API. For detailed descriptions of all functions available in both the MATLAB and C bindings, please see:

https://www.reactive-systems.com/api.msp

18.1 Using the Reactis API from MATLAB

It is easy to access the Reactis API from the MATLAB command line, from MATLAB scripts, or from MATLAB functions using a library of MATLAB functions included in the Reactis distribution. To use this API, add folder \lib\api\MATLAB\reactis within the Reactis installation directory to the MATLAB search path. If you used the default settings during the Reactis installation then the correct folder is:

C:\Program Files\Reactis V2020\lib\api\MATLAB\reactis

or (if running a 32-bit Reactis within a 64-bit version of Windows):

C:\Program Files (x86)\Reactis V2020\lib\api\MATLAB\reactis

After adding the folder, you can get information about the Reactis API functions via the regular MATLAB help functionality. For example, typing “help reactis” on the MATLAB command line will list information about all Reactis API functions. Detailed information for each function can be accessed by typing “help [functionname]” or “doc [functionname]”.

Some examples for using the API can be found in the following folder within the Reactis installation directory: \lib\api\MATLAB\reactis\examples

18.2 Using the Reactis API from C Programs

18.2.1 Quick Start

The following usage scenario highlights the most important functions in the API and explains the order in which to call them. A more detailed sample program apitest.c is included in the distribution to demonstrate more aspects of how to use the API.

Assume you want to do the following:
• Create a test suite for a model
• Export the test suite in CSV format to run it in a hardware in the loop environment.

You will need to call the following functions to accomplish this task:

1. Call rsOpen to receive a RsHandle value which all other API functions require as a parameter. In the following, the term RsHandle will refer to the handle returned by this call.

2. Call rsTester passing the RsHandle, a model file name and other parameters according to the documentation of rsTester. This will create a test suite and return an RsTestSuite value (an abstract data structure representing the generated test suite).

3. If the rsTester call in the previous step fails, call rsGetLastError to retrieve a description of the problem that caused the call to fail. This can be done if any of the API functions fail.

4. Call the rsGetCoverageCriteria functions to retrieve coverage of the test suite that was just created.

5. Call rsSimOpen passing the RsHandle, and a model file name. This will return a RsSim value which serves as a handle to the newly created Simulator session.

6. Call rsSimExportSuite passing the RsSim handle returned by rsSimOpen, the RsTestSuite value returned by rsTester and a filename with a .csv suffix. This will export the test suite created in step two in the CSV format supported by Reactis.

7. Call rsSimClose passing the RsSim handle returned by rsSimOpen. rsSimClose will free all memory allocated by the Simulator functions.

8. Call rsClose, passing the RsHandle as an argument. This will free all memory allocated by the Reactis API.

18.2.2 Compiling a C or C++ program with the Reactis API

You can compile and run an application that uses the Reactis API as follows.

• Insert #include "reactis.h" at the top of your C (or C++) code. This compiler directive specifies the names, parameters and return values of the Reactis API functions.

• Add a suitable .lib library file to your linker options to specify to the linker how to find the Reactis API functions in the “libreactis.dll” dynamic link library. Unfortunately, different compilers use different .lib file formats. The Reactis distribution contains .lib files suitable for a few compilers:

  Microsoft Visual C++ .net, 7.0, 8.0, 2005 and 2008 The correct library file for this compiler is libreactis_vc.lib
  GCC The Gnu C compiler included in distributions like Cygwin or MinGW accepts the Microsoft .lib file format.
  Use libreactis_vc.lib
MathWorks “mex” compiler If you want to use Reactis API functions in code compiled with the “mex” compiler tool distributed with MATLAB, use the `libreactis_lcc.lib` library file.

If you are using a compiler that is not listed, you can try the following:

- Some compilers include tools that can create `.lib` files from a given DLL. Check whether that is true for your compiler. If so, create a `.lib` library from `libreactis.dll` and use that.
- Other compilers accept the Microsoft `.lib` file format. Try adding `libreactis_vc.lib` to your linker input settings.
- Send an email to help@reactive-systems.com describing your compiler type and version. We will try to generate a matching “.lib” file as soon as possible.

• Compile your application. The way to do this is highly dependent on your compiler and application code. For example, to compile the “apitest.c” example using GCC, type

  ```bash
gcc -o apitest.exe apitest.c libreactis_vc.lib
  ```

• Add the Reactis “lib\api” folder (e.g., `c:\Program Files\Reactis\lib\api`) to your Windows search path. This is necessary so your application can locate the `libreactis.dll` library. Alternatively, you can copy `libreactis.dll` into the folder where your application’s executable file is located.

### 18.2.3 Reactis API Files for C Programming

The following is a description of files distributed with Reactis that are related to the Reactis API. You can find these files in the `lib\api` subfolder in your Reactis distribution:

- **reactis.h** Header file containing declarations for the API functions.
- **libreactis_vc.lib** Library file suitable for Microsoft Visual C++ and GCC.
- **libreactis_lcc.lib** Library file suitable for MathWorks `mex` compiler.
- **libreactis.lib** Same as `libreactis_vc.lib`
- **apitest.c** Example program to illustrate how to use the API functions.
- **apitest.exe** Compiled version of apitest.c
- **libreactis.dll** Dynamic Link Library containing implementations of the API functions.
Chapter 19

The Reactis License Manager

The Reactis License Manager now has its own user guide which can be found at:

CHAPTER 19. THE REACTIS LICENSE MANAGER
Appendix A

File Types and Extensions

This appendix lists the file extensions of the different file types that Reactis and the Reactis for C Plugin use and create.

A.1 File Types Used or Created by Reactis

.mdl, .slx  Model files created in Simulink. Reactis reads model files but does not modify them.

.rsi  Reactis Info File. Stores information about models used in Reactis, for example:

- Validator objectives added to a model.
- .rsm files assigned to S-Functions within a model.
- Parameters (set via the Edit menu in Reactis) which control model testing and execution.

Reactis automatically creates a .rsi file when a model is loaded the first time. The base name of the created .rsi file matches the model file name. The .rsi file is created in the same folder as the model file, unless specified otherwise in Global Settings → Files.

.rst  ReactisTest Suite. Contains test data created by Reactis Tester or Reactis Simulator. Can be loaded in Reactis Simulator to run the tests on a model.

.csv  Comma Separated Value files. Reactis Simulator can import data from .csv files into a .rst test suite or export data from a .rst test suite to a .csv file.

.m  MATLAB script. Reactis can export a test suite to .m script format in order to run the test suite on the model in Simulink (see Section 12.2).

.mat  MATLAB binary data. Reactis can export a test suite to .m script format in order to run the test suite on the model in Simulink (see Section 12.2).

.rsp  Reactis profile file. Stores temporary information about models such as:

- Recent error messages about the model
- History of parameters used for the model in Tester
- Watched variables in Simulator
- Currently opened scopes in Simulator
• Currently enabled coverage highlighting settings in Simulator

Reactis automatically creates a .rsp file when a model is loaded the first time. The base name of the created .rsp file matches the model file name. The .rsp file is created in the same folder as the model file, unless specified otherwise in Global Settings → Files.

.rtp Reactis Tester parameter file. If the corresponding option in Edit → General is enabled, Reactis generates a .rtp file every time a test suite is created via the Reactis GUI. The .rtp file contains all information necessary to re-create the same test suite, including a copy of all settings in the current .rsi file. Note that .rtp files can only be used to re-create test suites within the same major version of Reactis. When using a .rtp file with a different version of Reactis, the re-created test suite may differ from the original.

.mwi Reactis model cache file. If the corresponding option in Edit → General is enabled, Reactis creates .mwi files for every model when either Simulator or Tester are starting up. The .mwi file contains an imported version of the model’s content, allowing Reactis to skip the Importing MDL file... phase of Tester or Simulator.

---

A.2 File Types Used or Created by the Reactis for C Plugin

.c, .h C source and header files. Reactis reads these files but does not modify them.

.rsm Reactis Makefile. Stores information about how to build an S-Function or static/dynamic library. To create a .rsm file, right-click on an S-Functions and selecting Assign RSM file, then either select an existing file or enter a new file name. See Section 16.3 for more information about creating .rsm files.

.rso C object file. If Create and use cache files for C code is checked in Edit → C Code then Reactis will create a .rso file for each .c input file.

.rsl Dynamic library cache file. If Create and use cache files for C code is checked in Edit → C Code then Reactis will create a .rsl file for each S-Function or dynamic library defined by a .rsm file.

.rsls Static library cache file. If Create and use cache files for C code is checked in Edit → C Code then Reactis will create a .rsls file for each static library defined by a .rsm file.

.rsxm Cache of initial Reactis for C Plugin machine state. If Create and use cache files for C code is checked in Edit → C Code then Reactis will create a .rsxm file for the model. This file contains the complete initial state of the Reactis for C Plugin and therefore speeds up Reactis Simulator and Tester startup time.
Appendix B

Frequently Asked Questions

Please see https://reactive-systems.com/faq.msp to view the Reactis frequently asked questions.
Appendix C

Revision History

Different versions of Reactis are labeled as shown in Figure C.1 and described below.

Figure C.1: Version labels begin with a “V” and include three parts: a year, a major release number, and a patch release number. The parts are numbers separated by decimal points. By convention, trailing zeros are omitted.

Major Releases. A new version of Reactis is released at the start of each year and labeled by a “V” followed by the four-digit year, for example V2018. Each label for an intra-year release includes a suffix consisting of a decimal point followed by a major release number; for example V2018.1, V2018.2, etc. will label the releases during 2018 that follow V2018.

Beta Releases. Reactive Systems sometimes makes beta releases available to customers interested in evaluating the newest features of Reactis. Beta releases do not undergo as much testing as major releases do. By convention, beta releases have odd numbered major release numbers. For example, V2018.1, and V2018.3 denote beta releases.

Patch Releases. Both stable and beta releases may be patched. The label for a patch release is constructed by extending the label for the major release to be patched with a suffix that includes a second decimal point and a patch release number. For example:

V2018.0.1 denotes the first patch release for V2018
V2017.2.3 denotes the third patch release for V2017.2
APPENDIX C. REVISION HISTORY

C.1 Patches Mailing List Archive

Patches to Reactis are posted to the Reactive Systems website between major releases. To view a summary of recent changes, please view the archives of the Reactis Patches mailing list available at:

https://www.reactive-systems.com/mlists.msp?lid=2

C.2 V2020 (17 July 2020)

The V2020 release of Reactis includes the bug fixes that were included in patch releases through V2019.2.3 plus the new features listed below.

C.2.1 Newly Supported Simulink Features

The following Simulink features are supported in Reactis V2020:

- MATLAB R2020a.
- Stateflow charts configured to use the MATLAB action language.
- State Writer and State Reader blocks.
- Initialize Function and Reset Function blocks.
- Reset event input ports in referenced models.

C.2.2 Other improvements

- Added the `pre()` function, which is used to access the value of a variable from a previous simulation step from within a Validator expression objective.
- Test suite export dialog preserves settings from previous export.

C.2.3 Reactis for C Plugin

- Added support for macros which access structure members from within a Stateflow action.

C.2.4 Reactis for EML Plugin

- For a matrix \( A \), Reactis previously supported the transpose operator \( A' \). V2020 adds support for performing the operation with the `transpose` function: `transpose(A)`.
- New trigonometric functions: `csc`, `sec`, `cot`.
- For an enumerated type `EnumType`, support:
  - the notation `EnumType(EnumVal)` which is equivalent to `EnumType.EnumVal`
  - the notation `EnumType(n)` to denote the element of `EnumType` with underlying integer value `n`. 
• Support $A \sim n$ where $A$ is a square matrix and $n$ is an integer.
• Other new functions: deg2rad, cond

### C.3 Previous Major Release Dates

Major releases of Reactis prior to V2019.2 have occurred on the following dates:

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2019.2</td>
<td>20 December 2019</td>
</tr>
<tr>
<td>V2019</td>
<td>28 June 2019</td>
</tr>
<tr>
<td>V2018.2</td>
<td>20 December 2018</td>
</tr>
<tr>
<td>V2018</td>
<td>27 June 2018</td>
</tr>
<tr>
<td>V2017.2</td>
<td>22 December 2017</td>
</tr>
<tr>
<td>V2017</td>
<td>7 July 2017</td>
</tr>
<tr>
<td>V2016.2</td>
<td>22 December 2016</td>
</tr>
<tr>
<td>V2016</td>
<td>20 June 2016</td>
</tr>
<tr>
<td>V2015.2</td>
<td>18 December 2015</td>
</tr>
<tr>
<td>V2015</td>
<td>30 June 2015</td>
</tr>
<tr>
<td>V2014.2</td>
<td>19 December 2014</td>
</tr>
<tr>
<td>V2014</td>
<td>25 June 2014</td>
</tr>
<tr>
<td>V2013.2</td>
<td>17 December 2013</td>
</tr>
<tr>
<td>V2013</td>
<td>14 August 2013</td>
</tr>
<tr>
<td>V2012.2</td>
<td>19 December 2012</td>
</tr>
<tr>
<td>V2012</td>
<td>14 June 2012</td>
</tr>
<tr>
<td>V2011.2</td>
<td>20 December 2011</td>
</tr>
<tr>
<td>V2011</td>
<td>22 June 2011</td>
</tr>
<tr>
<td>V2010.2</td>
<td>19 November 2010</td>
</tr>
<tr>
<td>V2010</td>
<td>8 February 2010</td>
</tr>
<tr>
<td>V2009.2</td>
<td>22 July 2009</td>
</tr>
<tr>
<td>V2009</td>
<td>16 January 2009</td>
</tr>
<tr>
<td>V2008</td>
<td>10 June 2008</td>
</tr>
<tr>
<td>V2007</td>
<td>31 October 2007</td>
</tr>
<tr>
<td>V2006.2</td>
<td>6 November 2006</td>
</tr>
<tr>
<td>V2006</td>
<td>25 January 2006</td>
</tr>
<tr>
<td>V2005</td>
<td>18 May 2005</td>
</tr>
<tr>
<td>V2004.2</td>
<td>17 December 2004</td>
</tr>
<tr>
<td>V2004</td>
<td>3 March 2004</td>
</tr>
<tr>
<td>V2003.2</td>
<td>16 October 2003</td>
</tr>
<tr>
<td>V2003</td>
<td>4 February 2003</td>
</tr>
<tr>
<td>V2002</td>
<td>3 June 2002</td>
</tr>
</tbody>
</table>
Index

.m files, 160, 217–219
.mat files, 161, 218
.mdl files, 219
.rsi file, 12, 65
.rsi files, 11, 66, 79, 140
.rsm files, 43, 252
.rsp files, 132, 150
.rst files, 128, 134, 149, 171, 193, 196
.rtp files, 46, 91, 172, 196
.slx files, 12, 65

absolute tolerances, 89
analyzing subsystems, 52
assertions, 65

branch coverage, 103
breakpoint, 129
breakpoints, 29

c code, 48
callbacks, 48
Child State Exit via Parent Transition, 112
colors in diagrams, 166
command line parameters, 7
condition combination coverage, 113
conditional input branch execution, 90, 247
configuration variables, 48, 51, 79, 133, 230
configuration variable sets, 81, 133
in S-Functions, 230
counters, 176, 185, 224
coverage, 103
color, 166
cumulative subsystem, 125
excluding targets, 48
reporting, 135, 199
specifying metrics, 48
tracking, 18
coverage metrics, 103
branch coverage, 103
condition, 113
condition action coverage, 111
conditional subsystems, 103
CSEPT, 112
decision, 113
integer saturation, 119
lookup table, 108
MC/DC, 113
MCC, 113
state coverage, 111
transition action coverage, 111
user-defined targets, 177
coverage objectives, 103
CSEPT, 112, 157, 159, 160, 167
cumulative subsystem coverage, 125
data items
displaying values, 129
tracking values, 22
desktop models, 127
decision
default model-specific settings, 47, 63
delta types, 79
diagram objectives, 176
difference scopes, 134, 149
disabling
Simulator, 41
distribution scopes, 132, 149, 207
Download, 3
drawing
styles, 166
editing tolerances, 89
Embedded MATLAB, 245, 279
EML, 245, 279
enabling
Simulator, 41
errors
runtime, 94

303
specifying severity, 48
executing test suites in Simulink, 216
exporting
  .mat file, 161
  .m file, 160
  comma separated values, 161
  CSV, 161
to MATLAB, 216
expression objective, 176
external .m files, 281
external EML functions, 281
extracting subsystems, 52
FAQ, 297
fast simulation, 21
feedback
  report to RSI, 2
file format
  m, 160, 217–219
  mat, 161, 218
  mdl, 219
  rsi, 11, 12, 65, 66, 79, 140
  rsm, 43, 252
  rsp, 132, 150
  rst, 128, 134, 171, 193, 196
  rtp, 46, 91, 172, 196
  slx, 12, 65
files
  .mwi, 91
  .rsp, 150
  .rst, 21, 149
  extensions, 295
find, 47
Frequently Asked Questions, 297
functional testing, 26
global settings, 46
help facility, 68
  menu, 50
history, 299
importing
  comma separated values, 162
  CSV, 162
  Inf, 94, 255
import types, 69
import values
  constraining, 48, 69
from test, 128, 149
from user, 141
probabilities, 79
querying from user, 25
reading from test, 21
imports
  constraining, 48
  types, 48
imports source
  pull-down menu, 128
input mode
  user, 25, 141
inputs
  plotting, 139
install
  silent, 5
installation, 3
Installer, 3
interpolateoff, 217
interpolation, 217
interruptible timers, 224
lines
  color, 166
linking to requirements document, 190
lookup table coverage, 108
LUC, 258
m files, 160, 217–219
MAC address, 60
mat files, 161, 218
MATLAB
  configuration, 57
  invocation by Reactis, 57
  path, 92
  workspace, 227
max tolerances, 89
MC/DC, 19, 113, 157, 167
MCC, 20, 113, 158
mdl files, 219
min tolerances, 89
models
  decomposing, 226
default settings, 63
file dependencies, 49
hierarchy, 129
preparing, 9
printing, 54
INDEX

search path, 49
searching, 47
simulating, 17, 127
multi-block decision coverage, 97, 115, 129
multiple condition coverage, 20, 113
NaN, 94, 255
natural language requirements, 190
outport tolerances, 88, 93, 131, 149, 154, 218
outputs
tolerances, 48, 88, 93, 131, 149, 154, 218
outputs
comparing simulation vs. test, 134
plotting, 139
updating to reflect model changes, 134, 164
overflow, 94
path
MATLAB, 92
MATLAB, 58
Reactis, 58
global, 58
model-specific, 49
paths, 49, 58, 92
port types, 69
printing models, 54
probabilities of input values, 79
profile, 132, 150
Reactis
installing, 3
Reactis Embedded MATLAB, 245, 279
Reactis Info File, 12, 65
Reactis profile, 132
relative tolerances, 89
REML, 245, 279
reports
coverage, 135, 199
test execution, 133, 135
requirements
natural language, 190
requirements documents
linking to, 190
resolution, 15
resolution types, 79
revision history, 299
rsi file, 12, 65
rsi files, 11, 66, 79, 140
rsm files, 43, 252
rsp files, 132, 150
rst files, 128, 134, 171, 193, 196
rtp files, 46, 91, 172, 196
running test suites in Simulink, 216
timeout errors, 94
S-functions, 1, 101, 103, 220, 229, 248, 258
scopes, 22, 149
adding data item, 150
difference, 134, 149, 154
distribution, 149, 207
saving currently opened, 132, 150
searching
models, 47
test suites, 208
settings
C loop detection timeout, 256
cache files, 91
coverage, 48
default model-specific, 47, 63
error checking, 48
file dependencies, 49
global, 46, 55
loop detection timeout, 97
model execution, 48
model-specific search path, 49
multiple trigger event outputs, 96
outport tolerances, 48
propagating changes, 91
Stateflow outport merge, 91
tolerance, 93, 218
undirected local event broadcast, 96
signal group, 153
signals
tracing, 43, 131
silent install, 5
simulation, 17, 127
fast, 21
user guided, 25, 141
simulation probabilities, 79
Simulator, 17, 127
disabling, 41
enabling, 41
Simulink, 229
executing test suites in, 216
running test suites in, 216
supported blocks, 234
slx files, 12, 65
Source-of-Inputs dialog, 17, 128
start from command line, 7
starting Reactis, 7
Stateflow, 244
Stub file
  generating, 275
subsystems
  analyzing, 45, 52
  extracting, 45, 52
support
  report to RSI, 2
TargetLink
  supported blocks
    version 2.2.1, 241
    version 2.3.1, 242
    version 3.0 to 3.4, 243
    version 3.5 to 4.2, 244
  synchronizing import ranges, 67
technical support
  report to RSI, 2
test points, 44
test points, 48, 82
  plotting, 139
  updating to reflect model changes, 134, 164
test suites
  adding a test, 29, 134
  browsing, 50
  executing in Simulator, 21, 128, 149
  executing in Simulink, 216
  executing in Simulator, 215
  executing on source code, 220
  exporting, 160
  filtering, 208
  importing, 162
  searching, 208
Tester, 169
  launch dialog, 169
  launching, 49
tests
  executing, 128
  execution reports, 133, 135
  generating, 169
timer objectives, 176, 185
  placement, 186
timers, 176, 185
  interruptible, 224
targets, 224
tolerances
  absolute, 89
  editing, 89, 131
  max, 89
  methods, 89
  min, 89
  output, 88, 93, 131, 149, 154, 218
  relative, 89
tracing
  signals, 43
TUC, 70
type under construction, 70
types
  in .rsi files, 69
  of inputs, 48
  synchronizing with TargetLink, 67
UDTs, 65, 177
uncovered
  color, 166
unit testing, 45, 52
unreachable
  color, 166
update outputs, 134, 164
update test points, 134, 164
update test suite to reflect model changes, 164
user input mode, 25, 141
user-defined targets, 65, 177
Validator, 175
  objectives, 14, 48, 65, 175
  diagram, 176
  expression, 176
  timer, 176, 185
Validator objectives, 48
variables
  displaying value during simulation, 22, 129
  distribution, 149
  watched, 22, 132, 149
version, 51, 68, 201, 299
view profile, 132, 150
virtual sources, 177
watch list
  saving, 150
watch lists
  saving, 132
watched variables, 132
saving current watch list, 132, 150