How to Contact Reactive Systems:

**Web**  www.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 for C User’s Guide

Copyright © 2011-2019 Reactive Systems, Inc. All rights reserved.

This document and the Reactis for C 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. Other names are trademarks or registered trademarks of their respective holders.

ISBN 978-1-257-91214-8
CONTENTS

5.4 Include Search Path ............................................. 47
5.5 Libraries ......................................................... 48
5.6 Defines .......................................................... 49
  5.6.1 Instrumentation Code ....................................... 50
  5.6.2 General ....................................................... 50
5.7 Generating Stubs ................................................. 51
5.8 Pathnames ....................................................... 51

6 The Reactis Harness Editor ...................................... 53
  6.1 Creating a new harness ....................................... 54
  6.2 General ......................................................... 58
  6.3 Inputs .......................................................... 59
  6.4 Outputs ........................................................ 60
    6.4.1 Tolerance Methods .................................... 61
  6.5 The Tolerance Editor Dialog ................................ 62
  6.6 Configuration Variables ..................................... 62
  6.7 Names .......................................................... 63
  6.8 Properties ..................................................... 64
  6.9 Types .......................................................... 64
  6.10 The Type Editor Dialog ..................................... 67
    6.10.1 Using the Type Editor .................................. 67
    6.10.2 Labeled Window Items ................................. 69
  6.11 Error Checking ............................................... 71
  6.12 Coverage Metrics ............................................ 72
  6.13 Excluded Coverage Targets ................................. 74

7 Reactis Coverage Metrics ....................................... 75
  7.1 Statement Coverage .......................................... 75
  7.2 Function Coverage ........................................... 75
  7.3 Function Call Coverage ...................................... 75
  7.4 Decision Coverage .......................................... 76
  7.5 Condition Coverage .......................................... 76
  7.6 MC/DC Coverage .............................................. 76
  7.7 MCC Coverage ................................................ 77
  7.8 Boundary Value Coverage ................................... 78
  7.9 Assertion Coverage .......................................... 78
  7.10 Boundary Value Coverage .................................. 79
  7.11 Coverage Highlighting ...................................... 79
  7.12 Excluding Coverage Targets ............................... 79
    7.12.1 Disabling a Coverage Metric .......................... 80
    7.12.2 Excluding Individual Targets ....................... 80

8 Reactis Simulator ............................................... 83
  8.1 Labeled Window Items ........................................ 83
  8.2 Menus .......................................................... 87
  8.3 Creating Test Execution Reports ............................ 90
    8.3.1 Test Data Plots ......................................... 92
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.4 Specifying the Simulation Input Mode</td>
<td>93</td>
</tr>
<tr>
<td>8.4.1 User Input Mode</td>
<td>95</td>
</tr>
<tr>
<td>8.4.2 Test Input Mode</td>
<td>97</td>
</tr>
<tr>
<td>8.5 Tracking Data-Item Values</td>
<td>97</td>
</tr>
<tr>
<td>8.5.1 The Watched-Variable List</td>
<td>99</td>
</tr>
<tr>
<td>8.5.2 Scopes</td>
<td>99</td>
</tr>
<tr>
<td>8.5.3 Distribution Scopes</td>
<td>101</td>
</tr>
<tr>
<td>8.5.4 Difference Scopes</td>
<td>101</td>
</tr>
<tr>
<td>8.6 Tracking Code Coverage</td>
<td>102</td>
</tr>
<tr>
<td>8.6.1 The Coverage Summary Dialog</td>
<td>102</td>
</tr>
<tr>
<td>8.6.2 Coverage Information in the Main Panel</td>
<td>103</td>
</tr>
<tr>
<td>8.6.3 The Coverage Report Browser</td>
<td>107</td>
</tr>
<tr>
<td>8.7 Exporting and Importing Test Suites</td>
<td>107</td>
</tr>
<tr>
<td>8.7.1 Exporting Test Suites</td>
<td>107</td>
</tr>
<tr>
<td>8.7.2 Importing Test Suites</td>
<td>108</td>
</tr>
<tr>
<td>8.8 Updating Test Suites</td>
<td>109</td>
</tr>
<tr>
<td>9 Reactis Tester</td>
<td>111</td>
</tr>
<tr>
<td>9.1 The Tester Launch Dialog</td>
<td>111</td>
</tr>
<tr>
<td>9.2 The Progress Dialog</td>
<td>114</td>
</tr>
<tr>
<td>10 Reactis Validator</td>
<td>117</td>
</tr>
<tr>
<td>10.1 Validator Objectives</td>
<td>118</td>
</tr>
<tr>
<td>10.1.1 Assertions</td>
<td>118</td>
</tr>
<tr>
<td>10.1.2 User-Defined Coverage Targets</td>
<td>118</td>
</tr>
<tr>
<td>10.2 Use Cases of Validator Objectives</td>
<td>118</td>
</tr>
<tr>
<td>10.3 Instrumentation Code</td>
<td>121</td>
</tr>
<tr>
<td>10.3.1 Advantages and Disadvantages of Instrumentation Code</td>
<td>121</td>
</tr>
<tr>
<td>10.3.2 Identifying Instrumentation Code</td>
<td>121</td>
</tr>
<tr>
<td>10.3.3 Adding Instrumentation Code</td>
<td>122</td>
</tr>
<tr>
<td>10.3.4 Editing Instrumentation Code</td>
<td>122</td>
</tr>
<tr>
<td>10.3.5 Cutting, Copying, and Pasting Instrumentation Code</td>
<td>123</td>
</tr>
<tr>
<td>10.3.6 Verifying the Location of Instrumentation Code</td>
<td>123</td>
</tr>
<tr>
<td>10.3.7 Moving Instrumentation Code</td>
<td>124</td>
</tr>
<tr>
<td>10.3.8 Removing Instrumentation Code</td>
<td>124</td>
</tr>
<tr>
<td>11 The Reactis Test-Suite Browser</td>
<td>125</td>
</tr>
<tr>
<td>11.1 Labeled Window Items</td>
<td>126</td>
</tr>
<tr>
<td>11.2 Menus</td>
<td>127</td>
</tr>
<tr>
<td>11.3 Test Step Filter Editor</td>
<td>129</td>
</tr>
<tr>
<td>12 The Reactis Coverage-Report Browser</td>
<td>133</td>
</tr>
<tr>
<td>12.1 Labeled Window Items</td>
<td>133</td>
</tr>
<tr>
<td>12.2 Menus</td>
<td>135</td>
</tr>
<tr>
<td>12.3 Exporting Coverage Reports</td>
<td>135</td>
</tr>
<tr>
<td>12.4 Coverage Report Contents</td>
<td>136</td>
</tr>
<tr>
<td>13 Preparing Code for Use with Reactis for C</td>
<td>139</td>
</tr>
<tr>
<td>13.1 Basic type sizes and alignments</td>
<td>139</td>
</tr>
<tr>
<td>13.2 Unsupported C Features</td>
<td>140</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Reactis® for C offers automated testing, debugging, and validation for software implemented with the C programming language. Reactis for C consists of three main components: Reactis Tester, Reactis Simulator, and Reactis Validator.

Reactis Tester automatically generates test suites from C code. The test suites provide comprehensive yet concise coverage of different test-quality metrics. Each test case in a test suite consists of a sequence of input vectors as well as the responses to those inputs generated by the program. These tests may be used for a variety of purposes, including:

**Detecting runtime errors.** Generation and execution of the tests often detects runtime errors that can occur as a program executes.

**Regression testing.** Tests may be generated from one version of a program and run on a different version in order to check conformance between the two versions.

Reactis for C enables users to maximize the effectiveness of their testing while reducing time and effort.

Reactis Simulator enables users to visualize program execution. Simulator’s user interface is similar to those of traditional debuggers: it allows you to step through the execution of code by hand, set break points, and study values of variables. Simulator also supports reverse execution, the replay of tests generated by Reactis Tester, the graphical display of different coverage metrics, and the fine-tuning of Tester-generated test suites.

Reactis Validator performs automated searches of programs 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 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 cruise control always disengage when the brake pedal is pressed?
- Will a plane’s thrust-reverser ever engage while the aircraft is airborne?
- Will an x-ray machine ever deliver a dangerous dose of radiation?
1.1 Support and Feedback

RSI welcomes user feedback and questions. To ask questions, make suggestions, or report suspected bugs, users may call RSI’s help line at (+1) 919-324-3507 or send e-mail to help@reactive-systems.com. When sending e-mail, users are encouraged to include the “System Info” information for their Reactis for C installation in question; this can be obtained by selecting the Help → About menu item from the top-level Reactis for C window, then clicking Copy To Clipboard and pasting the information into the e-mail message.
Chapter 2

Installing Reactis for C

Reactis for C is available on Windows® XP, Vista, 7 and 10. To run Reactis for C, a system should satisfy the following minimum requirements:

- At least 1GB RAM (more is required for large programs)
- At least 60 MB free disk space
- An installed Ethernet card

The Reactis for C install process differs slightly depending on whether you have purchased a node-locked or floating license. Note, that evaluation licenses are node-locked.

For floating licenses, you (or someone in your organization) will also need to install the Reactis License Manager.

2.1 Installing with a Node-Locked License

To install Reactis for C, perform the following steps:

1. Execute the self-installing executable reactis4c-setup-V2019.exe and follow the instructions. You may always 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 reactis4c-setup-V2019.0.n.exe where $n$ is a patch release number. Download address:

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

2. To obtain a Reactis for C license, first find the (incomplete) license file rslicense.dat. This file can be found in the directory where Reactis for C was installed. The default location for the license file is either:

   C:\Program Files\Reactis for C V2019\rslicense.dat

   or, if 32-bit Reactis for C is installed on a 64-bit version of Windows:
The location of rsilicense.dat can be found by starting Reactis. When Reactis is started without a license, the full pathname of rsilicense.dat is displayed, as shown by highlighted item 1 in Figure 2.1. Once you have located rsilicense.dat, send a copy to help@reactive-systems.com. You will receive a response containing a completed license as an attachment. When you receive this e-mail, replace the original rsilicense.dat with the completed license.

2.2 Installing with a Floating License

If your organization already has a server running the Reactis License Manager, then you may install Reactis for C on your computer (the client) as follows. Note that in this case you will need the name or IP address of the server running the license manager; your network or system administrator can supply you with this information.

1. On the client machine, run the installer as described in Step 1 of “Installing with a Node-Locked License”.  

1Windows® is a registered trademark of Microsoft Corporation.
2.2. INSTALLING WITH A FLOATING LICENSE

2. Invoke Reactis for C by selecting

Start → Programs → Reactis for C V2019 → Reactis

The dialog shown in Figure 2.2 will appear, enabling you to specify the server on which the Reactis License Manager is running. To specify a server, click the Add button and then enter the IP address or name of the server. If your organization has more than one server running the Reactis License Manager, you may enter as many of these as you wish.

Figure 2.2: Configuring Reactis for C to access a Reactis License Manager running on a remote server.
2.3 Installing the Reactis License Manager

See the Reactis License Manager User’s Guide, available at:

2.4 Performing a Silent Install

The Reactis for C installer supports a silent install (installing Reactis for C 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 for C 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 for C API</td>
</tr>
</tbody>
</table>

- /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 for C 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
- 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.

```
C:\> reactis4c-setup-V2018.exe /SILENT /LOADINF="mysetup.inf"
```

will install Reactis for C without any user prompts using the values saved in `mysetup.inf`. Options given on the command line will override parameters stored in the information file.

• `/HELP` or `/?` or `/H` Display help information for the silent install command line switches.
Chapter 3

Getting Started with Reactis for C

This chapter provides a quick overview of Reactis for C. It contains a brief description of each main component of the tool suite in the form of a “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 automobile cruise control application that is included in the Reactis for C distribution.

§1 To prepare for the tour, copy some files from the Reactis for C distribution to a local folder:

1. Create a folder r4ctour

2. Copy the files cruise.c, cruise.h, cruise_main.c, cruise_validator.c, cruise_validator.h, cruise_validator.rsm from the Reactis for C examples directory, default location:

C:\Program Files (x86)\Reactis for C V2019\examples

to your r4ctour working folder. Do not copy cruise.rsm and cruise.rsh, because those will be reconstructed in the following steps.

3.1 Quick Start

A typical usage scenario proceeds according to the following steps. The guided tour in the remainder of this chapter describes these operations in more detail.

1. Start Reactis for C.

2. Select File → New Build File... to open the Reactis Build File Editor and create a Reactis Build File (.rsm file) that lists the C source files that comprise your project.

¹Reactis Build Files were previously called Reactis Makefiles, hence the file suffix .rsm. The more generic “Build File” notation was adopted to distinguish from the specific Unix/GNU Make build environment.
3. Select Edit → New Harness... to create an execution harness. An execution harness defines a specific portion of your code to be tested. The primary components of a harness are an (optional) initialization function, an (required) entry function, a set of inputs, and a set of outputs. These components determine the structure of the tests constructed by Reactis for C, which performs the following actions as it constructs a test:

(a) Call the initialization function.
(b) Select a value for each input.
(c) Execute the entry function with the given inputs.
(d) Record the inputs and resulting outputs as a step in the test.
(e) If the program has not yet been fully exercised, goto step (b) to add another test step.

The inputs and outputs are either global variables or parameters of the entry function.

4. Select Edit → Inputs... to specify the set of inputs and optionally constrain the set of values an input should assume during testing. For example, you can indicate that a given input should fall within a range (give min/max values) or specify that it should come from an enumerated set of values.

5. Select Test Suite → Create... to generate a test suite (collection of tests) for your execution harness. The resulting test suite will be stored in a Reactis Test Suite file (.rst file). During test creation, runtime errors may be flagged for investigation.

6. When test generation completes, dismiss the Reactis Tester dialog and then start Reactis Simulator by selecting Simulate → Simulator On/Off.
3.2. **Reactis for C Top Level**

The Reactis for C top-level window contains menus and a toolbar for launching testing and validation activities on C programs. Reactis for C is invoked as follows.

§2 Select Start → Programs → Reactis for C V2019 → Reactis for C from the Windows Start menu (or double-click on the desktop shortcut if you installed one).

You now see a Reactis for C window like the one as shown in Figure 3.1. A program to be tested is defined by a *Reactis Build File* (.rsm file) that lists the collection of C source files that comprise the program. The next step creates an .rsm file for the cruise control example.

![Build File Editor](image)

Figure 3.2: The Build File Editor after adding source files to `cruise.rsm`.

7. Load the newly-generated test suite by selecting Test Suite → Open... and then selecting the new .rst file.

8. Execute the whole test suite by selecting Simulate → Fast Run.

9. Examine the source code in the main panel looking for code marked in red to indicate that coverage targets were not exercised by the tests.
§3 Create cruise.rsm as follows:

1. To specify a name and location for the new build file, select File → New Build File..., then use the resulting file-selection dialog to navigate to the r4ctour folder created in the earlier step, specify a file name of cruise.rsm, and click the Save button.

2. The Build File Editor opens to specify the contents of cruise.rsm. In the editor, click the Add button beside the Source Files section. This will open a file selection dialog (titled Select Source Files). In the file selection dialog, specify cruise.c and cruise_main.c, then click Ok. At this point appearance of the Build File Editor should be similar to Figure 3.2.

3. Click the Add button beside the Libraries section, then in the resulting dialog select cruise_validator.rsm.

4. In the Build File Editor, click Ok to save your changes and dismiss the editor.

After creating the build file, the top-level window changes as shown in Figure 3.3. The panel to the right shows information about the build file. The panel to the left shows the source files in the project. In addition, the title bar now reports the build file currently loaded, namely cruise.rsm and the Reactis Harness Library (.rsh file) cruise.rsh that contains test harnesses for testing and validation. Harnesses are created and managed within Reactis for C as explained in the next section.

Figure 3.3: Reactis for C after creating cruise.rsm.
3.3 The Harness Editor

Reactis for C does not modify any source files in a program. Instead the tool stores project-specific information that it requires in a Reactis Harness Library (.rsh file). The primary way for you to view and edit the data in these files is via the Reactis Harness Editor, which is described briefly in this section and in more detail in Chapter 6.

§4 The next stop in the guided tour explains how to create a harness.

1. Select the Edit → New Harness... menu item.

2. In the Create Harness dialog, enter cruise_main in the Harness name entry box.

3. Click the ... button beside the Entry function entry box.

4. The resulting dialog lists the functions from your program that are candidates for becoming the entry function for the harness under construction. Select function cruise_main from the list and click Ok.

5. In the Create Harness dialog, click Ok to close the dialog and create the new harness.

6. Note that in the main window title bar a * appears next to cruise.rsh to indicate it has been modified. Also (cruise_main) is also shown in the title bar to indicate that the new harness is the currently selected harness. Select File → Save Harness Library.

Next, we will open the the Reactis Harness Editor to examine the inputs that were created for the new harness. When you create a new harness, by default, all pass by value parameters to the entry function are considered inputs, while all pass by reference parameters to the entry function and any return value are considered outputs. You can then modify the set of inputs and outputs inferred from the entry function or tag global variables as inputs or outputs. The default type for each parameter is the base type inferred for the parameter from the program. If you add or remove a parameter to your entry function you can synchronize your .rsh file with the new version of the entry function by selecting Tools → Synchronize from the Reactis Harness Editor.

§5 To open the harness editor, select Edit → Inputs....

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

The Reactis Harness Editor has seven panes:
General. View the harness name, initialization function, and entry function. Modify general parameters such as the sample rate.

Inputs. Add and remove items to the list of inputs and specify type information used by Reactis for C to constrain the set of values fed into inputs during simulation, test-case generation, and C code validation.

Outputs. Add and remove items to the list of outputs.

Configuration variables. Add and remove items to the list of configuration variables, global variables tagged by the user to be changed in between tests in a test suite (but not during a test). The set of values a configuration variable may be assigned during testing may be specified with a type exactly as inputs are.

Error Checking Specify the action to take when an anomalous condition occurs, such as a numeric overflow.

Coverage Metrics Specify which coverage metrics will be tracked during simulation and the details of each metric.

Excluded Coverage Targets Examine targets which are excluded from coverage tracking, and re-enable tracking for some targets.

The types that may be specified are the base Reactis for C 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.
3.3. THE HARNESS EDITOR

Complete range of base type: By default the type associated with an input is the base type inferred from the program. Allowed base types along with their corresponding C types are as follows:

<table>
<thead>
<tr>
<th>Reactis Base Type</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>bool</td>
</tr>
<tr>
<td>int8</td>
<td>char</td>
</tr>
<tr>
<td>uint8</td>
<td>unsigned char</td>
</tr>
<tr>
<td>int16</td>
<td>short int</td>
</tr>
<tr>
<td>uint16</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>int32</td>
<td>int</td>
</tr>
<tr>
<td>uint32</td>
<td>unsigned int</td>
</tr>
<tr>
<td>int32</td>
<td>long int</td>
</tr>
<tr>
<td>uint32</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>single</td>
<td>float</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
</tbody>
</table>

Subrange of base type: $t[i, j]$, where $t$ is a base type, and $i$ and $j$ are elements of type $t$, with $i$ being a lower bound and $j$ an upper bound.

Subrange with resolution type: $t[i : j : k]$, where $t$ 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 $t$. The allowed values that a variable 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 a variable 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: $t\{e_1, \ldots, e_n\}$, where $t$ is a base type and $e_1, \ldots, e_n$ are either elements of type $t$, or expressions of form $v : w$, where $v$ is an element of type $t$ 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 a variable having an enumeration type. If the probability weight is omitted it is assumed to be “1”. For example, an input 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. See Chapter 6 for more details.

Delta type: $t \delta [i, j]$, where $t$ is either a base type, a range type, or a resolution type, and $i$ and $j$ are elements of the underlying base type of $t$. 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.

Table 3.3 gives examples of types and the values they contain.
### Table 3.1: Type examples.

<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; inputs of this type can increase or decrease by at most 1 in successive simulation steps.</td>
</tr>
</tbody>
</table>

In this step we will add constraints for inputs to limit the values selected for those inputs during test generation.

1. Select the row for input onOff.
2. Right-click on the row and select Edit....
3. In the resulting dialog, select radio button Set of specific values, click Add.
4. In the Enter Values dialog enter 0, 1, click Ok.
5. In the Type for 'onOff' dialog click Ok to save the changes and close the dialog.
6. Select Edit → Copy to copy the type we just configured for onOff.
7. Select the row for accelResume, select Edit → Paste to restrict accelResume to the values 0 and 1 as we did for onOff.
8. Repeat the paste operation for cancel, decelSet, brake, gas.
9. Select the row for inactiveThrottleDelta, right-click on the row and select Edit..., select Set of specific values, click Add, enter values: −0.1, 0.0, 0.1, accept the changes.
10. Select the row for drag, right-click on the row and select Edit..., select Subrange of base type, enter min and max values of −0.01 and 0.01. In the Delta section select radio button Limits and enter min and max allowed changes of −0.001 and 0.001. This delta constraint specifies that the drag value can increase or decrease by at most 0.001 in between any two consecutive steps.

After adding all the constraints, the result should be similar to Figure 3.5.

Configuration variables are global variables in your program that you specify should be
allowed to change in between tests but not during a test. In the next step of the tour we add a configuration variable.

§7

1. In the harness editor, select the Configuration Variables pane.

2. Select Edit → Add..., then in the Select variables dialog, choose variable InitialSpeed, then click Ok.

3. In the harness editor, make sure the InitialSpeed row is selected, then right-click and select Edit....

4. In the Type for 'InitialSpeed' dialog, select Subrange of base type, specify min of 25, max of 50, and click Ok.

Finally, save the changes and exit the harness editor.

§8 From the harness editor's toolbar select File → Save to save the harness changes then select File → Exit to close the Reactis Harness Editor.

3.4 Instrumentation Code

The final step before simulating and testing is to add some instrumentation code, which will contain code to test specific requirements of the program.
CHAPTER 3. GETTING STARTED WITH REACTIS FOR C

Figure 3.6: The C Code Editor is used to add instrumentation code to the program under test.

Figure 3.7: Instrumentation code is identified by a colored background.

§9 Right-click on line number 106 (click directly on the line number, not on the line itself) of cruise.c and select Add Instrumentation Code. A window titled C Code Editor will appear. Type (or copy and paste) the following into the main panel of the editor:

```c
/* Assertion: SpdCheck */
reactis_assert(cruise_assert_spdCheck(*active, speed, dSpeed));

/* User Defined Target: DeactivateActivate */
reactis_target(cruise_target_deactivateactivate(onOff, *active, decelSet, speed));
```

After the text has been entered, the appearance of the dialog should be similar to Figure 3.6. Click the Ok button to close the editor. The instrumentation code should now appear after line 106 of cruise.c, as shown in Figure 3.7.

You will now need to add instrumentation code in three more places.
§10 After line 4 of cruise.c, add the following:

```c
#include "cruise_validator.h"
```

After line 1 of cruise_main.c, add the following:

```c
#include "cruise_validator.h"
```

After line 46 of cruise_main.c, add the following:

```c
/* User-defined target: Active30 */
reactis_target(cruise_target_active30(*active));

/* User-defined target: LowSpeedOn */
reactis_target(cruise_target_lowspeedon(onOff, *speed));

/* Assertion: Brake */
reactis_assert(!(brake && *active));

/* Assertion: Speed */
reactis_assert(cruise_assert_speed(*speed, *active));
```

Note that there is an asterisk after cruise.rsm in the title bar. This indicates that there are unsaved changes to the instrumentation code. Save the changes now.

§11 Select Edit → Save instrumentation code

Congratulations! You are now ready to do some simulation and testing.

3.5 Simulator

Reactis Simulator provides a rich array of facilities for viewing the execution of programs. To continue with the guided tour:

§12 Click the (enable Simulator) button in the toolbar to start Reactis Simulator.

This causes a number of the toolbar buttons that were previously disabled to become enabled.

Simulator performs simulations in a step-by-step manner: for each call to the entry function, values are generated for each input, and after the entry function has executed, the values of the outputs are recorded. You can control how Simulator computes input values with the Source-of-Inputs Dialog in the toolbar (see Figure 3.8) as follows. The tool may be directed to:
CHAPTER 3. GETTING STARTED WITH REACTIS FOR C

1. generate values randomly (this is the default);

2. query the user;

3. read inputs from a Reactis test suite. Such tests 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.

The next part of the guided tour illustrates the use of each of these input modes. Inter- spersed with this discussion are asides on coverage tracking, data-value tracking, and other useful features.

Figure 3.8: The Source-of-Inputs Dialog allows you to control how Reactis Simulator generates values for top-level inputs.

3.5.1 Generating Random Inputs

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

 hexadecimal ] In the panel on the left of the Simulator window, click on cruise.c in order to display source code in that file. Then, click the ▶️ (Run/Pause Simulation) button. When the simulation time (shown in the bottom right corner of the window) reaches 5, click the ▶️ button again to pause the simulation.

During simulation, lines of code are highlighted in green as they execute. 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 halt the simulation by clicking the ▶️ (Run/Pause Simulation) button.

3.5.2 Tracking Code Coverage

While Simulator is running you may track coverage information regarding which parts of your program 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.
Figure 3.9: The Reactis coverage tracking features convey which parts of a program 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.

§14 **Select menu item** Coverage → Show Summary.

A dialog summarizing coverage now appears, and elements of the program not yet exercised are drawn in red, as shown in Figure 3.9. If you see no red in the main panel, make sure Coverage → Show Details is selected. Note that poor coverage is not uncommon with random simulation. A statement is drawn if red if it has not executed. A thin red line over a decision (e.g. Boolean expression of an if statement) indicates the decision has never evaluated to true. A thick red line over a condition (atomic predicate in a decision) indicates that the condition has not evaluated to true. Similarly, underlining indicates whether decisions and conditions have evaluated to false. 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 as follows:
§15 Perform the following.

1. In the main panel displaying cruise.c, scroll until the function `Mode` is displayed.

2. Right click on the decision `g_dsMode==M_OFF && onOff` and select View coverage details.

Details for decision coverage, condition coverage, MC/DC and MCC are conveyed using the tables shown in Figure 3.10. The left table (Decision tab) in this figure gives information for the decision — `g_dsMode==M_OFF && onOff` — which includes two conditions — `g_dsMode==M_OFF` and `onOff`. 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 relevant situation has not yet been encountered. 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 met the MC/DC criterion 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:test/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.

The right table (MCC tab) in Figure 3.10 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 3.10.

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 program elements have been covered along with the test and step where they were first exercised. A simulation run, and associated coverage statistics, may be reset by clicking the (reset) button in the toolbar.

### 3.5.3 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.

§16 Click the button in the toolbar 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 for C 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:

§17 Click on the Source-of-Inputs Dialog.

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 last test in the suite:

§18 Do the following.

1. Select the last test in the test suite in the dialog then click the Ok button.

2. Click the Run/Pause Fast Simulation button.

If you look at the bottom-right corner of the window, you can see that the test is being executed, although the results of each execution step are not displayed graphically. When the test execution completes, the exercised parts of the C code 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.

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 in the Source-of-Inputs Dialog with the left mouse button.
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.

You may also use the Source-of-Inputs Dialog to change the name of a test. To do so, select the test by single clicking on it, then click on the name and, when the cursor appears, type in a new name and press return.

### 3.5.4 Tracking Values of Data Items

As shown in Figure 3.11, when Simulator is paused at a breakpoint, you may view the current value of a given variable by hovering over the variable with your mouse cursor.

![Figure 3.11: Hovering to query the value of a variable and when it was last updated.](image)

You may also select data items whose values you wish to track during simulation using the `watched-variable` and `scope` facilities of Simulator.

**§19** Navigate to the Cruise function in `cruise.c`, then right-click on the variable “mode” and select **Add to Watched** from the resulting pop-up menu.

The bottom of the Simulator window now changes to that indicated in Figure 3.12. 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.

**§20** Repeatedly click `▶` (step into) to execute a statement at a time. Observe how the value displayed in the watched variable panel changes from `[out of scope]`, to `[not initialized]`, to a value.
3.5. SIMULATOR

Figure 3.12: The watched-variable panel in Simulator displays the values of data items, as does hovering over a data item with the mouse.

Scopes display the values a given global variable has had at the end of each step since the beginning of the current simulation run. To open a scope:

§21 In the main panel, scroll to the top of cruise.c, right-click in variable g_dsMode and select Open Scope from the resulting pop-up menu.

Figure 3.13: A scope window plotting the mode of the cruise control during a test. 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.

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

3.5.5 Querying the User for Inputs

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

§23 Select User Guided Simulation from the Source-of-Inputs Dialog.
Figure 3.14: The **Next Input Values** dialog enables you to hand-select input data.

Upon selecting this input mode, a **Next Input Values** dialog appears, as shown in Figure 3.14, that allows you to specify the input values for the next simulation step. Each input has a row in the dialog containing five columns; these determine the next input value for the corresponding parameter as follows.

1. The input name.

2. The column 2 pull-down menu has entries that determine how the next value for the parameter is specified:
   - **Random**: Randomly select the next value for the input from the type given for the input in the .rsh file.
   - **Entry**: Specify the next value with the text-entry box in column three of the row.
   - **Panel**: Open a separate dialog to control the input.

3. If the pull-down menu in column two is set to “Entry”, then the next input value is taken from this text-entry box.

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

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

When **Run Simulation** (▶) or **Run Fast Simulation** (▶) is selected, the input values specified are used for each subsequent simulation step until the simulation is paused.
3.5. SIMULATOR

3.5.6 Other Features

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

- The $\uparrow$ button executes a single step, i.e. a single execution of the harness entry function.
- The $\downarrow$ button executes a single C statement, perhaps stepping into the body of a function being called.
- When execution is paused at a function call, the $\uparrow$ button executes the function, then pauses after the function returns.
- The $\Rightarrow$ button completes the execution of the currently executing function.
- The $\leftarrow$ button steps back by a single C statement, perhaps stepping back into a function that was called immediately before the current statement.
- When execution is paused after a function call, the $\leftarrow$ button steps back without stepping into the function.
- The $\Rightarrow$ button steps back to the statement that called the currently executing function.
- The $\lln$ button, which causes the simulation to go back (undo) a single step or go back to the beginning of the current step.
- The $\lll$ button causes the simulation to go back multiple steps.

You may specify the number of steps taken when $\uparrow$, $\downarrow$, or $\ll$ are pressed by adjusting the number in the text-entry box to the right of $\downarrow$. 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. When 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.

Breakpoints 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. The $\bullet$ symbol is drawn to the right of a line number when a break point is set. During a simulation run, whenever a breakpoint is hit Simulator pauses immediately. When paused at a breakpoint, you may proceed either by:

- executing one or more statements at a time with $\downarrow$, $\ll$, $\lll$
- reverse executing one or more statements at a time with $\lll$, $\ll$, $\lll$
- completing the current simulation step by clicking $\uparrow$.
- returning to the beginning of the current step by clicking $\lll$.

In either case, after reaching the end of the current simulation step, all other navigation buttons in the toolbar become enabled.
3.6 Tester

Tester may be used to generate a test suite (a set of tests) automatically from a C program as shown in Figure 3.15. The tool identifies coverage targets in the C code and aims to maximize the number of targets exercised by the generated tests.

![Diagram of C Program to Reactis Tester to Test Suite]

Figure 3.15: Reactis Tester takes a C program as input and generates a test suite.

To start Tester:

§24 Select the Test Suite → Create menu item.

This causes the Tester launch dialog shown in Figure 3.16 to appear. The first section of the dialog (Preload Files) lists previously-generated test suites to be pre-loaded. The second section (Run for) determines how long tester should run. The third section (Coverage Objectives) lists the metrics which Tester will focus on during the targeted testing phase. 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 3.16, is to run Tester for 20000 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. In most cases either the first or second option is preferred. For a description of the third option, see Chapter 9.
3.7. THE TEST-SUITE BROWSER

The Coverage Objectives section contains check boxes which are used to select the kinds of targets Tester will focus on during the targeted testing phase. Chapter 7 describes the different types of coverage tracked by Reactis.

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

§25 Click the Create Suite button.

The Tester progress dialog, shown in Figure 3.17, 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 test suite into the Test-Suite Browser (see below), or Reactis Simulator.

![Reactis Tester launch dialog.](image)

Figure 3.16: The Reactis Tester launch dialog.

3.7 The Test-Suite Browser

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

§26 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 3.18 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 toolbar. The main panel in the browser window shows the indicated test as a matrix, in which the first column gives the names of the harness inputs and outputs and each subsequent column lists the values for each input or output for the corresponding simulation step. The simulation time is displayed in the output row labeled “...t...”. The buttons on the toolbar 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 offers distribution scopes to display the entire set of values assumed by an input or output during a single test or set of tests.
3.7. **THE TEST-SUITE BROWSER**

Figure 3.18: The Reactis Test-Suite Browser.

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

1. **Left click on the row for input** `drag`.

2. **Right click on the row and select** Open Distribution Scope (current test).

A dialog similar to that shown in Figure 3.19 appears. In the figure, each value assumed by the input `drag` is represented by a yellow dot.

Figure 3.19: Values assumed by input `drag` during a test.
3.8 Validator

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

**Assertion checking.** You can instrument your code with *assertions*, which monitor code behavior for erroneous scenarios. If Reactis 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 code 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 code behavior; however, their purpose is to determine when a desired test case has been constructed, so that the test may be included in a test suite.

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).

The following methods can be used to track assertions:

- Add standard C `assert` statements in the code and enable the Create coverage targets for `assert()` statements setting in the Reactis Settings dialog. Reactis will then track the `assert` calls as Reactis assertions instead of raising an exception if the assertion fails.

- Add `reactis_assert` statements in your code and add a directive:

  ```c
  #include <reactis_validator.h>
  ```

  at the top of the source file. Reactis will count the assertion as violated if the argument to the `reactis_assert` call is zero.

To track user-defined targets, add `reactis_target` statements in your code and add a directive:

- Add `reactis_target` statements in your code and add a directive:

  ```c
  #include <reactis_validator.h>
  ```

  at the top of the source file. Reactis will count the target as covered if the argument to the `reactis_target` call is non-zero.
Chapter 4

The Reactis for C Top-Level Window

The next several chapters of this manual contain detailed descriptions of different components of Reactis for C. This chapter concentrates on the functionality available in the top-level window when Simulator is turned off, or disabled. Simulator is disabled when Reactis for C is first invoked; it may also be explicitly turned off by clicking button (window item 11 in Figure 4.1). Clicking button (window item 12) turns Simulator on; the functionality of the Simulator-on mode is described in Chapter 8.

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

An annotated screen shot of the Reactis for C top-level window may be found in Figure 4.1. This section describes the functionality of the numbered items in this figure, while the section following discusses the workings of the pull-down menus.

The numbers below refer to the labels in Figure 4.1.

1. The hierarchy panel shows the C files in the project and how they are related. Clicking on the ▼ to the left of an item displays the children of the item. Clicking on the ▲ to the left of an item hides the children of an item. Clicking on the name of a file displays the file contents in the main panel (window item 2). Pressing the [F2] key causes the parent of the currently displayed item to be displayed.

Right-clicking on an item in the hierarchy panel opens a pop-up menu from which the item can be edited. Note that editing is only allowed when Simulator is disabled.

2. The main panel displays the currently selected C code or build file. You may interact with the main panel in a number of different ways using the mouse including hovering, double-clicking on items, or right-clicking in various parts of the panel. Some operations in the main panel are only available when Simulator is enabled (see Chapter 8 for a description of those operations). The following mouse operations are available when Simulator is disabled:

When a build file is displayed in the main panel, clicking...

- on the Edit link in the project heading will open the Reactis Build File Editor to modify the .h file which defines the project.
- on a C file name will display the file contents in the main panel.
- on the Edit link next to a file name will open the file in the editor of your choice. See Section 4.3.1 for instructions on how to specify which editor to use.

When a C source files is displayed in the main panel, left-clicking...

- on the line number of a line of instrumentation code will select the line for subsequent cut or copy operations. See Section 10.3.5 for more details.

When a C source files is displayed in the main panel, right-clicking...

- on a line number will open a pop-up menu which lists options for adding and editing instrumentation code. See Section 10.3.4 for more details.

When the C source code is too big to display completely in the main panel, scroll bars appear for repositioning. 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.
One final note: pressing the [F2] key within this panel causes the parent of the currently displayed item to be displayed.


4. Open an existing Reactis build file.

5. Edit the currently selected Reactis build file or C file.

6. Undo the most recent change to the instrumentation code.

7. Redo the most recent change to the instrumentation code.

8. Go back in the history of displayed items.

9. Go forward in the history of displayed items.

10. Display the parent of the currently displayed item.

11. Disable Simulator.

12. Enable Simulator.

4.2 Menus

Some of the top-level menu items are described here; the rest are discussed in Chapter 8, which is devoted to Reactis Simulator.

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. The file menu contains the following entries.

New Build File... Create a new build file.

Open Build File... Load an existing build file into Reactis for C.

Save Instrumentation Code... Save all instrumentation code (in the corresponding build file).

Save Build File As... Rename and save the current build file.

Close Build File. Close the currently loaded build file.

Reload Build File. Reload the currently loaded project.

New Harness Library... Create a new harness library.

Open Harness Library... Load an existing harness library.

Extract Harness Library... Extract an .rsh file from an .rtp file. Reactis for C Tester may be configured to store launch parameters and the .rsh file used for a given run in a Reactis Tester Parameter file (.rtp file). Selecting this menu item retrieves the .rsh file from the .rtp file.
CHAPTER 4. THE REACTIS FOR C TOP-LEVEL WINDOW

Save Harness Library. Save the current .rsh file.

Save Harness Library As... Rename and save the current .rsh file.

Global Settings... Opens dialog to adjust Reactis for C global settings. Section 4.3 describes the use of this dialog.

Default Harness Library Settings... Opens a dialog to specify the default settings to be used when creating a new .rsh file. The dialog includes settings which specify:

- how a program executes (e.g. promotion of float values to double),
- error checking (e.g. flagging overflows or NaN values),
- coverage settings that specify which coverage metrics are used and how the coverage metrics are configured (e.g. the criteria for determining when an expression is a decision),

Whenever a new .rsh file is created, the initial settings match the default values. These settings can be changed via the Reactis Harness Editor. See Section 4.4 for more details on the Default Harness Library Settings dialog.

Exit Reactis for C. Stop Reactis for C.

Edit. This menu includes entries used to manipulate .rsm files and .rsh files. Note that these files may be modified only when Simulator is disabled. Therefore, when Simulator is enabled the information may be viewed but not changed. The Reactis Harness Editor is described in more detail in Chapter 6.

Build File... Open the Reactis Build File Editor.

Find... Launches a text search of the C files comprising your project. If a C file is currently displayed in the main panel, then only that C file will be searched. If an .rsm file is displayed in the main panel, then all C files named in the build file as well as those in any libraries referenced will be searched.

New Harness... Create a new harness in the currently loaded harness library.

General... View or modify basic properties of the current harness.

Inputs... Constrain the values arriving at inputs.

Outputs... Specify the set of outputs.

Configuration Variables... Manipulate configuration variables, global variables that may change in between tests but not during a test.

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 the current harness library. When 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.
4.2. MENUS

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.

**Undo** Undo the most recent change to the instrumentation code.

**Redo** Redo the most recently undone change to the instrumentation code.

**Cut** Copy the currently selected lines of instrumentation code to the clipboard and then delete them.

**Copy** Copy the currently selected lines of instrumentation code to the clipboard.

**Paste** Replace the currently selected lines of instrumentation code with the contents of the clipboard.

**View.** 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 8.2 for descriptions of these items.

**Back.** Go back in the history of what is displayed in the main panel.

**Forward.** Go forward in the history of what is displayed in the main panel.

**Go to Parent.** Cause the parent of the currently displayed item to be displayed in the main panel.

**Expand Tree.** Causes the entire tree in the hierarchy panel to be expanded.

**Collapse Tree.** Causes the entire tree in the hierarchy panel to be collapsed.

**Select C Source Font...** Select font for displaying C source code in the main panel.

**Show Recent Errors...** Display recent error messages encountered for the current project.

**Clear Recent Errors.** Remove all entries from the list of recent errors.

**Simulate.** 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 8.2.

**Simulator on/off.** Toggles whether Simulator is enabled or disabled. Implements the same behavior as labeled window items 11 and 12 in Figure 4.1.

**Test Suite.** 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 8.2.

**Create...** Launch Reactis Tester. See Chapter 9 for details.

**Browse...** Launches the Test-Suite Browser by first opening a file-selection dialog to allow user to indicate which test suite is to be browsed. See Chapter 11 for details.
Coverage. No items are enabled when Simulator is disabled.

Window. Used to switch between projects currently loaded in Reactis for C.

Help. The Help menu offers the following choices:

- **Contents.** Go to the table of contents in the in-tool documentation.
- **Index.** Go to the index in the in-tool documentation.
- **Frequently Asked Questions.** Go to the Frequently Asked Questions section of the in-tool documentation.
- **Release Notes.** Display the release notes for the current Reactis for C version.
- **Top Level.** Go to the section of the in-tool documentation that describes the Reactis top-level window.
- **Check for Updates...** Check the Reactive Systems website for a version of Reactis for C more recent than than the currently executing version.
- **About.** Open a dialog displaying the Reactis for C version and other configuration information. The dialog includes a Copy To Clipboard button to transfer the information to the Windows Clipboard. When requesting assistance, sending this information to Reactive Systems via email often facilitates the efficient delivery of support.

---

### 4.3 Reactis Global Settings

Selecting **File → Global Settings...** opens the Reactis for C Global Settings dialog, from which the behavior of Reactis for C can be adjusted. The settings are partitioned into four groups: **General**, **C Code**, **User Info**, and **License**. Each group is displayed in a separate tabbed pane as described below.

---

#### 4.3.1 General Settings

The **General** settings tab shown in Figure 4.2 includes the following items.

1. **GUI Language.** Language used in the Reactis for C GUI.
2. **Documentation Language.** Language used in the Reactis for C documentation.
3. **Source file encoding.** Character set encoding used by source files.
4. When creating a test suite, also create a parameters file (extension .rtp). Instructs Reactis Tester to store launch parameters and the .rsh file used for a given run in a Reactis Tester Parameter file (.rtp file). Subsequently:
   - the .rsh file may be extracted from the .rtp file by selecting File → Extract Harness Library...;
4.3. REACTIS GLOBAL SETTINGS

- the Tester parameters may be loaded into the Tester launch dialog by clicking the Load button.

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

5. Automatically check for updates (once a day). Instructs Reactis for C to check once per day whether updates to Reactis for C are available for download. If updates are found you will be asked if you would like to download and install the patch. To the right of this check box, the Proxy button summons a dialog from which a proxy server for Reactis updates is specified, and the Check Now button performs an immediate check for updates. Note: this feature can be disabled at install time; in which case this check box, the Proxy button, and Check Now button will not appear in the dialog.

6. Enable logging. Enables 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 by provided by the Reactis support team if you are asked to create a log file.
4.3.2 C Code

Figure 4.3: The C Code tab of the Global Settings dialog. The numbered items in this figure configure the behavior of Reactis for C as follows:

1. **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 the region where the line numbers are displayed.

2. **Display line grid.** Toggle the highlighting of every other line with a contrasting background color. The Background Color button allows you to select which color is used.

3. **Highlight instrumentation line numbers.** Toggle the highlighting of the line numbers of source code lines which contain instrumentation code with a contrasting background color. The Text Color button allows you to select which color is used.

4. **Highlight instrumentation code.** Toggle the highlighting of source lines which contain instrumentation code with a contrasting background color. The Text Color button allows you to select which color is used.
5. **Highlight comments.** Toggle the highlighting of comments in the source code with a contrasting text color. The Text Color button allows you to select which color is used.

6. **Highlight inactive code.** Toggle the highlighting of code which was disabled by a conditional compilation directive such as `#ifdef` with a contrasting text color. The Text Color button allows you to select which color is used.

7. **Source File Editor.** This allows you to specify which editor will be used to edit C source files.

### 4.3.3 User Info Settings

The User Info settings tab is shown in Figure 4.4. When Reactis for C is configured to use a remote license server as described in the next section, info contained in this panel is submitted to the server when Reactis for C 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 tab of the Global Settings dialog as described below, or using the standalone License Monitor utility included in the Reactis for C distribution. This utility may be invoked by selecting Reactis for C V2019 → License Manager → License Monitor from the Windows Start menu.

![Figure 4.4: The Global Settings dialog with tab User Info selected.](image)
4.3.4 License Settings

The **License** tab is shown in Figure 4.5 and 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 tab 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 lower portion of the tab displays information for each License Manager in the list. This information includes the status of the License Manager, the total number of licenses, the number of licenses currently being used, the IP addresses currently occupying licenses, and any info (names, phone numbers) available for users actually using the licenses.

![Figure 4.5: The Global Settings dialog with tab License selected.](image)

Each of the window items labeled in Figure 4.5 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:

**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, Reactis Model Inspector 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. 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.

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

5. Add a new License Manager to the list.

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:

**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 settings tab of the person occupying the license.

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

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

### 4.4 Default Harness Library Settings

The Default Harness Library Settings dialog lets you specify default settings to be used when creating new .rsh files. Whenever a new .rsh file is created, these settings are imported into the new .rsh file. Subsequently the settings for a harness library are modified using the Reactis Harness Editor. The dialog includes the following tabbed panes:

**General** settings which control how a program is executed.

**Error Checking** settings which specify when errors should be flagged (overflows, NaN values, etc.)

**Coverage** settings that specify which coverage metrics are collected for a program under test and how the coverage metrics are configured.

See Chapter 6 for a description of each setting.
Chapter 5

The Reactis Build File Editor

Reactis for C requires the C source files that implement a program as well as the source files for any libraries referenced by the program. The mechanism for listing these C source files is described in the rest of this section.

The collection of C source files that implement a program is specified using a Reactis Build File (.rsm file \(^1\)), which is created and modified using the editor shown in Figure 5.1. An .rsm file specifies how Reactis for C should compile a program for testing. Additionally, a separate .rsm file is required for each library referenced by a program. The .rsm files for the program and any libraries referenced by the program are defined in the same way. In the following discussion we denote the program/library compiled from an .rsm file as the library under construction (LUC).

5.1 Common Buttons

The .rsm file editor has 5 buttons which are always enabled regardless of which pane is selected. These buttons are labeled 6-10 in Figure 5.1, and are used as follows:

6. **Help.** Open the help dialog.
7. **Stub.** Generate stub source code for missing functions and variables.
8. **Build.** Build the LUC.
9. **Ok.** Close the .rsm file editor and save any pending modifications.
10. **Cancel.** Close the .rsm file editor without saving any pending modifications.

5.2 Item Selection

Four of the Reactis Build File Editor panes contain lists of items, such as source files or directories. Within each of these panes, a item listed within the pane can be selected by

---

\(^1\)Reactis Build Files were previously called Reactis Makefiles, hence the file suffix .rsm. The more generic “Build File” notation was adopted to distinguish from the specific Unix/GNU Make build environment.
left-clicking on the line containing the item. Selected items are distinguished by a blue background. A contiguous group of multiple items can be selected by left-clicking on the first item in the sequence and then holding down the shift button while left-clicking on the last item in the sequence. This will select the two items clicked-on plus all items appearing between them. A non-contiguous group of items can be selected by left-clicking on the first item and then holding down the control (Ctrl) key while left-clicking on subsequent items.

---

5.3 Sources

Figure 5.1 shows the Source Files pane of the .rsm file editor, which lists the C source files which implement the LUC. The contents of this pane are modified using the buttons labeled 1-4 in Figure 5.1:

1. **Add.** Add one or more source files to the LUC.

2. **Relative.** Convert all currently selected source files to relative path format.

3. **Absolute.** Convert all currently selected source files to absolute path format.

4. **Remove.** Remove all currently selected source files from the LUC.

See Section 5.8 for an explanation of relative and absolute pathnames. See Section 5.2 for an explanation of how source files within the Source Files pane are selected.
5.4 Include Search Path

Figure 5.2 shows the Include Search Path pane of the .rsm file editor, which lists the directories to be searched when processing a #include directive. The list is searched from top to bottom when processing a #include directive. The labeled items in Figure 5.2 are used as follows:

1. **Add.** Add one or more directories to the search path for the LUC. When Add is clicked, a dialog will appear in which you can either enter the name a directory directly or browse the file-system for a directory. In addition to the directory name, the dialog contains a check-box labeled *Recursively add all subdirectories*. If this box is not checked, then only the selected directory will be added to the search list. If *Recursively add all subdirectories* is checked, then, in addition to the directory itself, all directories contained within the directory will be added, as well as all the subdirectories of those directories, and so on, until every directory underneath the chosen directory has been added.

2. **Relative.** Convert all currently selected directories to relative path format. Relative pathnames are explained in Section 5.8. Directory selection is explained in Section 5.2.

3. **Absolute.** Convert all currently selected directories to absolute path format. Absolute pathnames are explained in Section 5.8. Directory selection is explained in Section 5.2.

4. **Remove.** Removes all currently selected directories from the search path. Directory selection is explained in Section 5.2.

5. These buttons are used to reorder the search list:
Move the selected directories up 1 position within the search list.
Move the selected directories down 1 position within the search list.
Move the selected directories to the start of the search list.
Move the selected directories to the end of the search list.

See Section 5.8 for an explanation of relative and absolute pathnames. See Section 5.2 for an explanation of how directories within the Include Search Path pane are selected.

5.5 Libraries

Figure 5.3: The Libraries pane of the .rsm file editor.

Figure 5.3 shows the Libraries pane of the .rsm file editor, which lists the libraries used by the LUC. The labeled items in Figure 5.3 are used as follows:

1. **Add.** Add one or more libraries to the LUC. A library is referenced via the .rsm file used to build the library.

2. **Edit.** Open 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. **Relative.** Convert all currently selected libraries to relative path format.

4. **Absolute.** Convert all currently selected libraries to absolute path format.

5. **Remove.** Removes all currently selected libraries from the LUC.

See Section 5.8 for an explanation of relative and absolute pathnames. See Section 5.2 for an explanation of how libraries within the Libraries pane are selected.
5.6 Defines

Figure 5.4 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:

- `name=value`, in which case instances of the identifier `name` will be replaced by `value` during preprocessing, or
- `name`, in which case `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 5.4 are used as follows:

1. **Add**. This button is used to add one or more macro definitions to the LUC.
2. **Edit**. This button opens a editor from which the currently selected macro definition can be changed.
3. **Remove**. This button removes all currently selected macro definitions from the LUC.

See Section 5.2 for an explanation of how macro definitions within the **Defines** pane are selected.
5.6.1 Instrumentation Code

Figure 5.5 shows the Instrumentation Code pane of the .rsm file editor. This pane lists the location of every segment of instrumentation code that is stored within the build file. Instrumentation code whose location needs to be verified are highlighted with red text. Double clicking on any location within the pane causes the corresponding instrumentation code to be displayed in the main window and highlighted with a yellow flashing background for a few seconds.

The labeled items in Figure 5.4 are used as follows:

1. **Highlight.** The button displays the currently selected instrumentation code in the main window and highlights it with a yellow flashing background for a few seconds. Highlighting can also be done by double-clicking on the location in the Instrumentation Code panel.

5.6.2 General

Figure 5.6 shows the General pane of the .rsm file editor. The labeled item in Figure 5.6 is used as follows:

1. **Coverage Tracking.** This drop-down menu controls the coverage tracking for the LUC. The possible choices are:
   - **On** Turn on coverage tracking within the LUC.
   - **Off** Turn off coverage tracking within the LUC.
5.7 Generating Stubs

The **Stub** button (item 7 in Figure 5.1) generates a source file which contains definitions for global variables and functions which are declared in the code, but are *not* defined.

A variable is defined if an instance of the variable exists which does not include the **extern** keyword.

A function is defined if an instance with a body containing the code for that function exists.

In the generated source file, all variables are initialized to numeric zero (0 for integers, 0.0 for floats, structure fields are individually initialized to numeric zero), and all functions ignore their arguments and return a value of zero (using the same rules as variables). The generated source file may require manual editing before it will compile, particularly if complex types or typedefs are used. In particular, **#include** directives may need to be added to the top of the source file.

Also note, that, if you want to stub missing variables with an initial value of zero and stub missing functions with functions that return zero, then it is easier to open the **Error Checking** pane of the Reactis Harness Editor and set **When an extern variable is undefined** to **Initialize to zero** and **When an undefined function is called** to **Return zero**. See Section 6.11 for details.

5.8 Pathnames

The path to a source file or directory 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:\testing\cruise\cruise.rsm

references the source file ..\csrc\ramp.c, the absolute path to ramp.c will be

C:\testing\csrc\ramp.c.

For convenience, the .rsm file editor provides two 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 or library.
Chapter 6

The Reactis Harness Editor

Reactis does not modify your C source files. Instead the tool stores project-specific information that it requires in a Reactis Harness Library (.rsh file). The primary way for users to view and edit the data in these files is via the Reactis Harness Editor, as described in this chapter. The key components of a harness are inputs, outputs, an initialization function, and an entry function. To construct a test for a harness, Reactis for C first calls the initialization function. It then repeatedly performs the following steps:

1. Selects values for each harness input.
2. Calls the entry function with the selected inputs.
3. Records the generated outputs.

Each harness library contains one or more harnesses. As shown in Figure 6.1, the harness selector in the toolbar of the Reactis Harness Editor enables the user to select a harness from those defined in the currently loaded .rsh file, to create a new harness, or to delete a harness. The currently selected harness is displayed in parentheses beside the .rsh file name in the title bars of the main window and harness editor.

The Reactis Harness Editor contains seven panes used to control different aspects of testing:

General. Displays the initialization function, entry function, harness name, sample rate, and some additional settings which control simulation. The initialization function is called once per test, and the entry function is called once per step during simulation and testing.

Inputs. A set of variables used as inputs. A new value is generated for each input prior to taking a step during simulation or test-generation. The user may constrain the set of allowed values for each input.

Outputs. A set of variables used as outputs. After each step the values of the outputs are recorded.

Configuration Variables. A set of variables which may change prior to the start of a test, but not during a test.
CHAPTER 6. THE REACTIS HARNESS EDITOR

Figure 6.1: The Reactis Harness Editor.

Error Checking Settings which specify the action to take when an anomalous condition, such as a numeric overflow, occurs.

Coverage Metrics Settings that specify which coverage metrics will be tracked during simulation and the details of each metric.

Excluded Coverage Targets A set of coverage targets which are excluded from coverage tracking.

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

Although not necessary, the default naming convention assumes that the .rsm file and .rsh file of a project share the same base name; for example, if the .rsm file is named cruise.rsm, then the name of the associated .rsh file is assumed to be cruise.rsh. A .rsh file named differently may be associated with a program by loading the build file in Reactis and selecting File → Select Harness Library....

6.1 Creating a new harness

To create a new harness, either select Edit → New Harness... within the main Reactis for C window or select the Create new harness... option in the harness selector drop-down box. This will bring up the dialog shown in Figure 6.2.

The numbered items in Figure 6.2 let you specify a harness as follows:

1. **Harness name.** Specify a name for the harness being created in this entry box.
2. **Initialization function** Each harness has an *initialization function* – a function that Reactis for C will call once per test, before the first step of the test is taken. The initialization function can either be set to **None**, or the name of a function can be entered into the text entry box.

3. **Entry function** This text entry box lets you specify an *entry function*, which Reactis for C will call for every step in a test. To construct a test for a harness, Reactis for C repeatedly:
   
   (a) Selects values for each harness input.
   (b) Calls the entry function with the selected inputs.
   (c) Records the generated outputs.

4. ... Clicking this button opens a dialog that lets you select an entry function from a list of candidate functions.

5. ... Clicking this button opens a dialog that lets you select an initialization function from a list of candidate functions. Only functions which have no arguments and do not return a value can to be used as initialization functions.

6. **Create separate input/output for each structure member** This menu determines how structures in the test harness are handled and has two choices: (1) split the structure into a separate input or output for each member of the structure, or (2) create a single input or output for the entire structure.
7. **Create new harness.** Creates a new harness with the minimum set of inputs and outputs, as inferred from the entry function’s parameters and return value type:
   - Each pass-by-value argument to the entry function is added as an input.
   - Each pass-by-reference argument to the entry function is added as an output
   - If the entry function’s return value type is non-void then an output is added for the return value.

8. **Copy existing harness.** Copies inputs, outputs and configuration variables from an existing harness in the current harness library. This can be useful when defining harnesses for different entry functions that share some of the inputs, outputs and configuration variables.

9. When **Copy existing harness** is selected, this menu lets you specify the harness to copy.

10. **Create harness from .rsi file** Copy inputs, outputs, and configuration variables from an .rsi file created with Reactis for Simulink for a Simulink model. This option is useful if test cases for the model have been created with Reactis for Simulink and C code was automatically generated from the model. When a harness is constructed with this option, tests generated from the model can be directly run on the C code with Reactis for C.

11. **RSI file.** If copying harness info from an .rsi file, enter the .rsi file name in this text box.

12. ... Open a file selection dialog to specify an .rsi file from which to copy harness info.

13. **Variable Renaming** The names of Simulink inports and outports might not match exactly with the names of corresponding variables in the C code. The Variable Renaming section of the dialog provides a way to map the names to resolve any differences. Using the text boxes in this section, you can specify common prefixes or suffixes for the items in the .rsi file and/or the C code. For each inport/outport/configuration variable in an .rsi file, the following process is used to try to map it to an item in the C code:

   (a) $itemname_1 =$ the name of the inport/outport/configuration variable as found in the .rsi file.
   (b) $itemname_2 =$ the result of removing the prefix specified in the RSI File prefix column from the beginning of $itemname_1$.
   (c) $itemname_3 =$ the result of removing the suffix specified in the RSI File suffix column from the end of $itemname_2$.
   (d) $itemname_4 =$ the result of adding the prefix specified in the C Code prefix column to the beginning of $itemname_3$.
   (e) $itemname_5 =$ the result of adding the suffix specified in the C Code suffix column to the end of $itemname_4$. 

(f) In the C code, if a global variable or parameter of the entry function is named \texttt{itemname}_5 then it will be added as an input/output/configuration to the harness. Its harness name will be \texttt{itemname}_1 and it will map to C code item \texttt{itemname}_5. If \texttt{itemname}_5 does not match any global variable or entry function parameter, then Reactis will produce a warning that a match could not be found.

Figure 6.3 shows some mapping examples:

**Example A** Mapping inport/outport names with prefixes or postfixes to C variables:
- Simulink inports named “SLin\_name” will map to “name” in the C code.
- Simulink outports named “name\_SLout” will map to “name” in the C code.

**Example B** Mapping inport/outport names to C variables with prefixes or postfixes:
- Simulink inports named “name” will map to “Cin\_name” in the C code.
- Simulink outports named “name” will map to “name\_Cout” in the C code.

**Example C** Mapping with prefixes or postfixes in both inport names and C variables: Simulink inports named “SLPrefix\_name\_SLSuffix” will map to “CPrefix\_name\_CSuffix” in the C code.

14. **Automatically match embedded coder functions/variables.** If the C code was produced using MATLAB Embedded Coder, checking this box will automatically apply renaming rules which match model names to code names.

15. **Help.** Clicking this button opens the help dialog.

16. **Ok.** Clicking this button causes the harness creation to proceed.

17. **Cancel.** Clicking this button closes the dialog without creating a harness.
6.2 General

The General tab of the harness editor is shown in Figure 6.4. The settings controlled by this tab are labeled 1 through 5 in Figure 6.4.

1. **cruise_main**. This pull-down menu is used to select the current harness, create a new harness, or delete the current harness. The displayed value is the name of the current harness.

2. **Initialization Function**. The initialization function is called at the start of each test.

3. **Sample rate**. The sample rate is the number of seconds to add to the time (\( \_\_t\_\_ \)) at the end of each step. When performing back-to-back testing, the test harness sample rate will typically match the sample rate of the model.

4. **Create and use cache files for C code**. When checked, Reactis will generate a Reactis object file for each C source file. These files have extension .rso and are stored in the directory containing the C source file from which they were compiled. Subsequently, the C source file will only be recompiled if it is modified or if a new version of Reactis is installed.

5. **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).

6. Set Test Execution Report defaults. This button opens a dialog which controls the default settings for test execution reports.

### 6.3 Inputs

The **Inputs** tab (see Figure 6.5) of the harness editor enables you to define a set of inputs. Inputs are either parameters to the entry function or global variables which are designated as inputs by selecting **Edit → Add...** and then choosing from the candidate list. Parameters which are passed by reference can also be designed as inputs via the same mechanism.

![Figure 6.5: The Inputs tab of the harness editor defines the set of inputs and limits on the values that the inputs can assume during simulation and test generation.](image)

The **Inputs** tab contains six columns for each input value. These are numbered 1 through 6 in Figure 6.5. The purpose of each column is as follows:

1. **#** The number of the input (1...N).

2. **Name** The name of an input is usually, but not always, the same as the name of the variable which holds the input value. See Section 6.7 for details.

3. **Variable** The name of the variable which holds the input value.

4. **File** The name of source file in which the variable which holds the input value is defined.
5. **Properties**  A code which describes how the input is passed through the harness and stored in the program under test. See Section 6.8 for details.

6. **Type**  This field specifies the set of values that can be assigned to the input. See Section 6.9 for details.

### 6.4 Outputs

The Outputs tab of the harness editor enables the user to add and remove items to the set of outputs. A value is recorded for each output at the end of each simulation step as tests are constructed. By default all pass-by-reference parameters of the entry function are considered outputs, but they may be removed from the list and global variables may be added. When the Outputs tab is displayed, entries from the Edit menu may be used to add or remove items to/from the list.

![Figure 6.6: The Outputs pane of the harness editor.](image)

The Outputs tab contains six columns for each input value. These are numbered 1 through 9 in Figure 6.6. The purpose of each column is as follows:

1. **#**  The number of the output (1...N).
2. **Name**  The name of an output is usually, but not always, the same as the name of the variable which holds the output value. See Section 6.7 for details.
3. **Variable**  The name of the variable which holds the output value.
4. File  The name of source file in which the variable which holds the output value is defined.

5. Properties  A code which describes how the output is stored in the program under test and passed through the harness. See Section 6.8 for details.

6. Type  The type of the output. Double clicking opens a type editor dialog from which the type can be changed.

7. Tolerance  The tolerance method to use when comparing an actual output value with the value in a test suite (see Section 6.4.1).

8. Relative  The relative tolerance value. If the method does not use relative tolerance, this will be blank.

9. Absolute  The absolute tolerance value. If the method does not use absolute tolerance, this will be blank.

### 6.4.1 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 Reactis Harness Editor).
6.5 The Tolerance Editor Dialog

Figure 6.7: 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.

Tolerances are specified via the tolerance editor dialog, which is shown in Figure 6.7. The labeled items in Figure 6.7 are explained as follows:

1. This pull-down menu lets you choose the tolerance method. The possible choices are Inherit, Relative, Absolute, Min, or Max. These are explained in Section 6.4.1.

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.

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.

6.6 Configuration Variables

Configuration variable directives allow certain scalar global variables to be treated as “configuration variables” whose values can only change between tests (but not during a test).

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 Reactis Harness Editor window that comes up. Reactis will display a list of all candidate configuration
variables which users may select from. The types associated with these configuration variables in .rsh files are the same as those that may be assigned to inputs. 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.

As shown in 6.8, the Configuration Variables tab has five columns, whose meaning is as follows:

1. **#** The number of the configuration variables (1...N).

2. **Name** A user-defined name which is usually, but not necessarily then as the name of variable. See Section 6.7 for details.

3. **Variable** The name of the configuration variable in the source code.

4. **File** The name of source file in which the configuration variable is defined.

5. **Type** This field specifies the set of values that can be assigned to the configuration variable. See Section 6.9 for details.

---

### 6.7 Names

The Inputs, Outputs and Configuration Variables tabs of the Reactis Harness Editor have columns titled **Name** and **Variable**. Both of these columns contain names. The name displayed in the **Variable** field is the actual name of the variable in the source code. This name cannot be changed and does not have to be unique. The name displayed in the **Name** field, on the other hand, can be changed and must be unique.
6.8 Properties

The Inputs and Outputs tabs of the Reactis Harness Editor have a column labeled Properties, which contains a code describing how each input and output is passed through the harness and stored in the program under test. There are four property codes, P, P*, R and G, whose meaning is given by the following table:

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>Pass-by-value parameter to entry function.</td>
</tr>
<tr>
<td>P*</td>
<td>Pass-by-reference parameter to entry function.</td>
</tr>
<tr>
<td>R</td>
<td>Return value of entry function.</td>
</tr>
<tr>
<td>G</td>
<td>Global variable.</td>
</tr>
</tbody>
</table>

6.9 Types

The Inputs, Outputs and Configuration Variables tabs of the Reactis Harness Editor have a column labeled Types. The Types column contains type specifications. For inputs and configuration variables, the type specification determines what values may be assigned to the variable during simulation, test generation and validation. The initial type specification for each variable is the equivalent of its C type. This indicates that the full range of values which the variable can hold should be generated during testing. In many cases, the set of desired input values is actually smaller than the full range of the C type. For example, an input may have a type of char, but may actually be used as a Boolean value whose allowed values are 0 or 1. If this is the case, then editing the type specification will significantly improve coverage results during testing.

Note that for outputs, the type specification for an input is permanently fixed to the equivalent to its C type.

Type specifications can be changed from the Inputs and Configuration Variables tabs of the harness editor. A type specification starts with a base type, which is selected from the list: int8, int16, int32, uint8, uint16, uint32, double, single.

A base type can then be extended by notation to specify (1) ranges and subsets within a base type, (2) rate-of-change (or delta) constraints, (3) resolution constraints and (4) array types. The full set of type specifications is defined by the following grammar (in which rsiType is the starting symbol).
6.9. TYPES

intType : int8, int16, int32, uint8, uint16, uint32

floatType : double, single

fixedPointType : sfixbitsSlopeBias | ufixbitsSlopeBias

numberType : intType | floatType | fixedPointType

baseType : numberType | boolean

rsiType : baseType | baseType { val1, val2, ..., valn } | numberType [ min, max ] | numberType delta [ min, max ] | numberType [ min, max ] delta [ min, max ] | floatType [ min : step : max ] | floatType [ min : step : max ] delta [ min, max ] | <\* rsiType1 * rsiType2 * ... * rsiType_n |

| vectorType | structType

vectorType : <\* rsiType1 * rsiType2 * ... * rsiType_n |

structType : { fieldName1 : rsiType1 , ..., fieldName_m : rsiType_m } | structName -> { fieldName1 : rsiType1 , ..., fieldName_m : rsiType_m }

val : value | value : weight

The variables in this grammar are defined as follows. min, max, step, and value are elements of the given base type, weight is an integer greater than zero and less than one thousand. For a vector type <\* rsiType1 * rsiType2 * ... * rsiType_n >, the base types of all rsiType_i must be identical. For a structure type, the base types of the different field types rsiType_i may be different.

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 \( \text{slope} = \text{f}p\text{Decimal} \)
- as \_FfpDecimal \_EfpInt such that \( \text{slope} = \text{f}p\text{Decimal} \times 2^{\text{f}p\text{Int}} \)

**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>&lt;</td>
<td>double { 1.0,2.0,3.0 }</td>
</tr>
<tr>
<td>double {10.0}</td>
<td>&gt;</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 }</td>
</tr>
<tr>
<td>double {10.0}</td>
<td>&gt;</td>
</tr>
<tr>
<td>{ x : uint8 { 1, 2 }, \</td>
<td>{ x = 1, y = 3 },</td>
</tr>
<tr>
<td>y : unit8 { 3, 4} }</td>
<td>{ x = 1, y = 4 },</td>
</tr>
<tr>
<td></td>
<td>{ x = 2, y = 3 },</td>
</tr>
<tr>
<td></td>
<td>{ 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 enumeration types (i.e. those of form \( t\{u_1, \ldots, u_n\} \)) the probability weights in the \( u_i \) enable the user to influence the probabilities of the different values in the type being selected during simulation, test generation, and C code validation. For example, an input 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 input \( x \) 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 input \( x \) being assigned the value \( v_i \) is \( w_i / W \).

Using **delta types** (i.e. those of form \( t \delta [l,u] \)) users may constrain how the values fed into inputs change from one simulation step to the next. For example, an input representing a sensor measuring temperature might have the type double delta [-1.0,1.0] to specify
that temperature can either increase or decrease by at most 1 degree between any consecutive simulation steps. In general, if input $x$ 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.

### 6.10 The Type Editor Dialog

The Type Editor Dialog, shown in Figure 6.9, helps users construct types for inputs and configuration variables. In the following discussion, we use TUC to denote the type under construction, i.e. the type currently being defined with the dialog.

The Type Editor may be invoked from either the Inputs or Configuration Variables tabs and selecting an input or configuration variable row with a single click, and then right-clicking in the body of the Reactis Harness Editor and selecting Edit Type, or selecting menu item Edit $\rightarrow$ Edit....

#### 6.10.1 Using the Type Editor

A type may be constructed with the dialog by performing the following steps. Recall that TUC denotes type under construction, i.e. the type currently being defined with the dialog. The process for editing a type works as follows:

1. Do you wish to use a type in the test harness which is different from the type in the C source code?
   - No. Leave the harness base type unchanged (it will initially match the internal base type) and proceed to step 2.
   - Yes. Use the pull-down menu (item 1 in Figure 6.9) to select the desired type and proceed to step 2.

2. Should the values chosen during simulation 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 (item 3 in Figure 6.9) and proceed to step 3.
   - Only a subrange of the base type. Select the Subrange of base type radio button (item 4 in Figure 6.9) and specify a lower and upper bound for the range using window items 5 and 6 respectively. If the input 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 (item 7 in Figure 6.9) and enter a resolution value in the text box to the right (item 8 in Figure 6.9). When a resolution is specified, a variable of the TUC will only assume values that differ from the lower bound (item 5 in Figure 6.9) by some multiple of the resolution value (item 8 in Figure 6.9). Proceed to step 3.
Figure 6.9: The dialog for specifying type information for inputs or configuration variables. As shown here a type of `double` is specified for input `x` which will be converted to fixed-point before being stored in the memory of the program under test. Furthermore, the input values for `x` have been constrained to the set \{-10000000.0, 0.0, 10000000.0\}. During random simulation, `x` will be assigned one of these three values with equal probability.

**A specific set of values in the base type.** Select the `Set of specific values` radio button (item 9 in Figure 6.9) and use the `Add`, `Edit`, and `Remove` buttons (items 12, 13, and 14 in Figure 6.9) 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 6.3 for a description of probability weights. Proceed to step 3.
3. Do you wish to constrain how the values fed into the input parameter may change from one step to the next during simulation?

**No.** Select the Complete radio button (item 16 in Figure 6.9) and proceed to step 4.

**Yes.** Select the Limits radio button (item 17 in Figure 6.9) and enter values in the Minimum and Maximum text entry boxes (items 18 and 19 in Figure 6.9). If \( \text{min} \) and \( \text{max} \) are the values specified here, then values fed into the entry function parameter with the TUC will be constrained as follows. If \( v_1 \) and \( v_2 \) are values for such a port in consecutive simulation steps, then \( v_1 + \text{min} \leq v_2 \leq v_1 + \text{max} \). Proceed to step 4.

4. Does the C code expect a fixed-point value as input?

**No.** Leave the internal type unchanged and proceed to step 5.

**Yes.** Select Fixpoint from the internal base type pull-down menu (item 20 in Figure 6.9), then select a scaling mode (item 21 in Figure 6.9), and finally the scaling parameters (item 24 in Figure 6.9). Proceed to step 5.

5. Does the conversion from the harness type to the internal type require rounding or can it overflow?

**No.** Proceed to step 6.

**Yes.** Select a rounding mode (item 25 in Figure 6.9) and/or overflow behavior (item 26 in Figure 6.9). Proceed to step 6.

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

7. Select File \(\rightarrow\) Save to save your changes to the .rsh file.

### 6.10.2 Labeled Window Items

This section describes the labeled items in Figure 6.9. Recall that TUC denotes *type under construction*, i.e. the type currently being defined with the dialog.

1. This pull-down menu changes the base type of the harness variable.

2. The radio buttons in this group (window items 3, 4, and 9) enable the user to specify the values from the base type to be include in the TUC.

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

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

5. When the Subrange of base type radio button is selected, this entry box enables the user to specify a lower bound for the subrange.
6. When the Subrange of base type radio button is selected, this entry box enables the user to specify an upper bound for the subrange.

7. If the Subrange of base type radio button is selected for a base type of double or single, then this check box may be checked in order to specify a resolution for the TUC.

8. When the Resolution check box (window item 7) is checked, a resolution value must 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.

9. Selecting this radio button indicates that only a specified set of values from the base type should be included in the TUC. Window items 10-14 are used to specify the set of values.

10. The entries in this column enumerate the values in the TUC.

11. The entries in this column associate a probability weight with each value in the TUC. See Section 6.3 for a description of probability weights.

12. Add a new value (a new entry in each of window items 10 and 11) to the TUC.

13. Edit the probability weight for the currently selected value.

14. Remove the currently selected value.

15. The radio buttons in this group (window items 16 and 17) enable the user to specify whether or not the type has a delta constraint. For a parameter 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 a parameter with the TUC may change by no less than the value specified here. If $v_1$ and $v_2$ are the values of the parameter in consecutive simulation steps then $v_2 - v_1 \geq min$.

19. The value for a parameter with the TUC may change by no more than the value specified here. $v_2 - v_1 \leq max$.

20. This pull-down menu changes the internal type from a fixed-point type. It can only be used when the type in the source code is an integer type.

21. This pull-down menu determines the scaling used when the internal base type is a fixed-point type. Available choices are Binary Point or Slope and Bias.

22. The signedness of the fixed-point type is displayed here. It matches the signedness of the underlying integer type and cannot be changed.
23. The width (in bits) of the fixed-point type is displayed here. It matches the width of the underlying integer type and cannot be changed.

24. When binary point scaling is selected, the fraction width (in bits) of the fixed-point type is entered here. If slope and bias scaling is chosen, the fraction width will be replaced with entries for the slope and bias.

25. This pull-down menu determines the rounding mode to use when converting values from the harness type to the internal type. Possible choices are Zero, Ceiling, Floor, or Nearest.

26. This pull-down menu determines the overflow mode to use when converting values from the harness type to the internal type. Possible choices are Saturate or Wrap over.

27. View help on the type editor dialog.

28. Update the TUC and dismiss the dialog. The revised TUC will appear in the appropriate row of the Reactis Harness Editor and “[modified]” will appear in the title bar. Selecting File → Save will write the .rsh file to disk and cause the “[modified]” tag in the title bar to disappear.

29. Discard any changes to the TUC and dismiss the dialog.

30. Update the TUC without dismissing the dialog.

6.11 Error Checking

The Error Checking pane of the Reactis Harness Editor, shown in Figure 6.10, lets you specify what action to take when anomalous conditions occur during program execution. The settings controlled via this pane are numbered 1-12 in Figure 6.10:

1. The action to take when a type delta exceeds the range of the type. Possible actions are Clip Delta, Clip delta and produce warning, or Produce error.

2. The default tolerance to use when comparing test suite outputs to program outputs.

3. The action to take when a signed integer overflow occurs. Possible values are Wrap over, Produce warning, or Produce error.

4. The action to take when an unsigned integer overflow occurs. Possible values are Wrap over, Produce warning, or Produce error.

5. The action to take when an overflow occurs while converting an unsigned integer value to signed integer. Possible values are Wrap over, Produce warning, Produce error, or Inherit from "On signed integer overflow".

6. The action to take when an overflow occurs while converting an floating point value to (signed or unsigned) integer. Possible values are Wrap over, Produce warning, Produce error, or Inherit from "On signed integer overflow".
7. The action to take when a NaN (not-a-number) or infinite value is detected. Possible values are No action, Produce warning, Produce error.

8. The action to take when an invalid pointer is detected. Possible values are Do nothing, Produce warning, or Produce error.

9. The action to take when an empty structure or union type is defined. Possible values are Do nothing, Produce warning, or Produce error.

10. The action to take when an undefined is called. Possible values are Return zero, or Produce error.

11. The action to take when an extern variable is never defined. Possible values are Initialize to zero, Produce warning and initialize to zero, or Produce error.

12. The simulation time limit (in seconds) for a simulation step to finish. A value of 0 makes the step time unlimited.

### 6.12 Coverage Metrics

The Coverage Metrics pane of the Reactis Harness Editor is used to specify which metrics should be tracked and how they should be tracked. An annotated screenshot of the Coverage Metrics pane appears in Figure 6.11. The following items are labeled Figure 6.11:

1. This setting enables or disables the C Statement metric. See section 7.1 for details.
6.12. COVERAGE METRICS

2. This setting enabled or disabled the C Function metric. See section 7.2 for details.

3. This setting enabled or disabled the C Function Call metric. See Section 7.3 for details.

4. This setting controls which expressions are considered to be decisions. Possible values are Expressions which control if/while/for statements, Expressions which control if/while/for statements or the ? operator, or All non-trivial Boolean expressions. See Section 7.4 for details on the effect of this setting.

5. This setting enables or disables the Decision metric. See Section 7.4 for details.

6. This setting enables or disables the Condition metric. See Section 7.5 for details.

7. This setting enables or disables the Modified Condition/Decision Coverage (MC/DC) metric. See Section 7.6 for details.

8. This setting enables or disables the Multiple Condition Coverage (MCC) metric. See Section 7.7 for details.

9. This setting controls the maximum number of conditions per decision for which MCC targets will be created. No MCC targets will be created for decisions which are composed from more conditions than the given limit. Note that $2^N$ MCC targets are created for a decision composed from $N$ conditions, so use caution when changing this setting. See Section 7.7 for details on the MCC metric.

Figure 6.11: The Coverage Metrics pane of the Reactis Harness Editor.
10. This setting enables or disables the Boundary Value metric. See Section 7.8 for details.

11. This setting enables or disables the User-Defined Target metric. See Section 10.1.2 for details.

12. This setting enables or disables the Assertion metric. See Section 10.1.1 for details.

13. This specifies if assertion targets should be created for each `assert` statement in a program.

### 6.13 Excluded Coverage Targets

![Excluded Coverage Targets](image)

Figure 6.12: The Excluded Coverage Targets pane of the Reactis Harness Editor.

The Excluded Coverage Targets pane is shown in Figure 6.12. Each row of this pane contains one target excluded from coverage. There are five data items per target:

1. **Metric.** The metric measured by the target.

2. **Item.** A description of the target (a location and source code fragment).

3. **Target.** A description of the action which causes the target to be covered. For example, if a target’s metric is `Condition`, then the target will be either `Condition true` or `Condition false`.

4. **Status.** The target status, which is either `excluded`, or `excluded with assertion`.

5. **System.** The source file and (possibly) library where the target is located.
Chapter 7

Reactis Coverage Metrics

Reactis for C uses a number of different coverage metrics to measure how thoroughly a test or set of tests exercises a program. In general, coverage metrics record how many of a given class of syntactic constructs, or coverage targets, that appear in a program have been executed at least once. The metrics discussed in this chapter may be visualized using Simulator and are central to test generation and C code validation using Tester and Validator.

7.1 Statement Coverage

Statement coverage records for each statement in a C program, whether or not the statement has been executed at least once.

7.2 Function Coverage

The Function metric records whether or not each function in a program has been called at least once. One target per function is created and a target is covered the first time the function is called.

7.3 Function Call Coverage

The Function Call metric records whether or not each particular call to a function has been executed at least once. One target per call to any function is created (i.e., if a function is called from multiple places, then a separate target is created for each call). A target is covered the first time the function is called from the location associated with the target. Targets are not created for calls via a pointer to a function.

Note that a single statement or condition may contain calls to more than one function, so function call coverage is not a subset of statement coverage.
7.4 Decision Coverage

*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 Reactis for C Global Settings dialog. 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.

---

7.5 Condition Coverage

*Condition coverage* tracks whether each condition in a program has evaluated to both true and false. A condition is a Boolean-valued subexpression of a decision which does not contain any Boolean operators. For example, if \( x < 100 \land \neg(y > 100) \) is a decision, then \( x < 100 \) and \( y > 100 \) are conditions.

---

7.6 MC/DC Coverage

The MC/DC coverage metric 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 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:

- \( C \) evaluates to a different truth value (true or false) in step \( X \) than in step \( Y \); and
7.7. MCC COVERAGE

- 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.

7.7 MCC Coverage

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 condition $D = \text{C1 && C2 && C3}$ 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>

However, in virtually all programming languages (including C), Boolean expressions are always evaluated using short-circuiting for reasons of efficiency. 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. 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 $D = \text{C1 && C2 && C3}$ 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>
Table 7.1: Boundary values associated with each type.

<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>double</td>
<td>0.0</td>
</tr>
<tr>
<td>single</td>
<td>0.0</td>
</tr>
</tbody>
</table>

7.8 Boundary Value Coverage

Boundary value coverage tracks whether a data item assumes values of interest for a particular variable. The boundary values tracked for an input or configuration variable are determined by its associated type \(^1\) as shown in Table 7.1. If an input or configuration variable has a type not shown in the table, then it has no boundary value targets.

7.9 Assertion Coverage

See Section 10.1.1.

\(^1\)Recall that types are associated with inputs using the Inputs tab of the Reactis Harness Editor (described in Chapter 6).
7.10 Boundary Value Coverage

See Section 10.1.2.

7.11 Coverage Highlighting

When Coverage → Show Details in Simulator is selected, uncovered coverage targets in C code are reported visually as shown in Figure 7.1. Any C statement which has not been executed is rendered in red. If a decision has not evaluated to true it has a thin red underline. 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. The MC/DC-related and MCC coverage details associated with a decision may be displayed by right clicking on the decision and selecting View Coverage Details.

To aid the user in finding uncovered targets, a thin red bar is drawn just to the right of the line number for any line that contains at least one uncovered coverage target.

Figure 7.1: Coverage highlighting in C code.

7.12 Excluding Coverage Targets

Reactis for C supports two different ways to disable coverage tracking for a subset of targets. When a target is excluded using either of these mechanisms, Reactis Tester will not try to generate a test to exercise the target and Reactis Simulator will not report the target as uncovered.
7.12.1 Disabling a Coverage Metric

In some cases, tracking a certain metric may not be desirable. For example, 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 Harness 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 7.2: Disabling all statement coverage targets via the Reactis Harness Editor (Edit → Coverage Metrics)

7.12.2 Excluding Individual Targets

Excluding individual targets from coverage tracking allows fine-grained control if the code under test contains 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 code under test 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 for C except for boundary targets can be excluded. The method to exclude a target differs slightly by target type.

The process for excluding Decision, Condition, MC/DC, or MCC targets from coverage requires three steps (labeled 1-3 in Figure 7.3):

1. Right-click on the code region containing the target and select View Coverage Details.
2. In the resulting dialog, right-click on the test/step information of the specific target.
3. Select the Track Coverage menu item.

For all other targets right-click on a statement, name of a function at the point where the function is defined, or name of a function within a function-call expression and select Track Coverage and select the appropriate target from the submenu (see Item 1 of Figure 7.4). A check-mark in the submenu listing coverage targets indicates that coverage is currently being tracked for that target.

Selecting Track Coverage brings up the dialog shown in Item 2 of Figure 7.4. 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:

- The color of the target in the source code will change to blue (instead of black or 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 Reactis Harness 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 target count in the coverage summary.
• The status of the excluded target 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 region of code containing the target is highlighted in yellow in the main panel.
• In the coverage report, the list of excluded targets, and when hovering in the main panel, 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.
Chapter 8

Reactis Simulator

Simulator provides an array of features — including single- and multi-step forward and backward execution, breakpoints, simulations driven by Tester-generated test suites, and interactive simulation — for simulating your source code. The tool also allows visual tracking of coverage data and the values data items assume during simulation.

![Figure 8.1: The Reactis Simulator toolbar.](image)

Figure 8.1 contains an annotated screen shot of the the Simulator toolbar. Some of the buttons and 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 8.1, while the section following discusses the menu entries related to Simulator. The subsequent sections discuss the different modes for generating inputs during simulation, the ways to track data values, how to monitor code coverage, the importing and exporting of test suites, and the different code highlighting styles used by Simulator.

8.1 Labeled Window Items

1. Disable Reactis Simulator.
2. Enable Reactis Simulator.
3. Clicking this button resets the simulation; the code execution is returned to the start state, and coverage information is appropriately reset.
4. Clicking this button causes the simulation to take \( n \) steps back, where \( n \) is specified by window item 15. Coverage information is updated appropriately upon completion of the last backward step.

5. Clicking this button causes the simulation to take one step back. Coverage information is updated appropriately.

6. Step backward to the point just before the current function was called.

7. Step backward one statement. Any function calls performed by the statement are stepped over.

8. Step backward one statement. If the statement performed any function calls, stop at the end of the last function which was called.

9. Clicking this button executes a single C statement, stepping into a function when at a function call. The button is disabled during fast simulation and reverse simulation.

10. When paused at a function call, clicking this button steps over the function (executes the function and pauses at the following statement).

11. Step out of the currently executing function.

12. Clicking this button causes the simulation step to advance forward by one full step; that is, values are read on the harness inputs, the program’s response is computed and values are written to the harness outputs. If a step has been partially computed, 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 harness outputs.

13. When paused, clicking this button causes \( n \) forward simulation steps to be taken, where \( n \) is specified by window item 15. The diagram in the main panel is updated during simulation to reflect the currently executing code. When Coverage \( \rightarrow \) Show Details is selected, coverage targets will change from red to black as they are covered during the simulation run. If the end of the current test or test suite is reached or you click the Run/Pause button again (window item 13), then simulation stops at the end of the current simulation step.

14. Clicking this button causes \( n \) simulation steps to be executed, where \( n \) is specified by window item 15. The diagram in the main panel 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 and the end of the currently executing step.

15. This window item determines how many steps are taken when buttons corresponding to window items 4, 13, or 14 are clicked. When the Source-of-Inputs Dialog (window item 16) 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.
16. The Source-of-Inputs Dialog determines how input values are computed during simulation. See Section 8.4 for details.

17. Clicking this button causes a new, empty test suite to be created. The name of the .rst file containing the suite is initially “unnamed.rst” and is displayed in the title bar of the Reactis for C window.

18. Clicking this button displays 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 16).

19. Clicking this button causes the current test suite to be saved.


21. The hierarchy panel (not shown explicitly) supports the navigation of the project, as described in Section 4.1. It shows the root build file (.rsm file) at the top, and the C files and libraries below.

22. The main panel displays the contents of the C or .rsm file currently selected in the hierarchy panel. You may interact with the panel in a number of different ways using the mouse. These include hovering over items in the code (e.g. variables, function names, macros) or right-clicking in various parts of the panel. The following mouse operations are available when Simulator is enabled:

**Hovering...**

- over a variable name (global or local) or function parameter displays\(^1\):
  - the current value of the variable or “[out of scope]” if the variable is not currently in scope.
  - where the variable is defined.
  - where the variable was last modified.
- over a function name displays the return type and types of its parameters along with the location of the function definition.
- over a macro name in a macro invocation displays the expansion of the macro.
- over a typedef shows the typedef declaration.
- over the argument of a #include directive displays the pathname of the file which was included.
- over any Reactis 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. For boundary value coverage, you may hover over the definition of inputs or configuration variables. In the case of inputs, this is either the definition of a parameter of the entry function or a

\(^1\)Depending on the variable and its type, some of the hover information may not be available.
global variable declaration. In the case of configuration variables, this is a
global variable declaration.
For more details on querying coverage information see Section 8.6, Chap-
ter 12, and Chapter 7.

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
summary of the menu items available when Simulator is enabled follows. For
descriptions of the menu entries available when Simulator is disabled, see Sec-
tion 4.1.

<table>
<thead>
<tr>
<th>Right-Click Location</th>
<th>Menu Entries (when Simulator is enabled)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global variable or static</td>
<td>Add To Watched  Add item to watched variables list (see section 8.5.1).</td>
</tr>
<tr>
<td>local variable</td>
<td>Open Scope  Display item in scope (see section 8.5.2).</td>
</tr>
<tr>
<td></td>
<td>Open Distribution Scope  Display item in distribution scope (see section 8.5.3).</td>
</tr>
<tr>
<td></td>
<td>Add To Scope  Add item to previously opened scope. This item only appears when other scopes are open.</td>
</tr>
</tbody>
</table>
| Decision coverage target   | View Coverage Details  Display dialog containing detailed coverage information for the decision (i.e. stat-
|                            |   us of decision, condition, and MC/DC targets).                                                        |
| Harness output             | Open Difference Scope  This menu item is enabled when a test suite is loaded. It opens a scope which    |
|                            |   displays the differences between the output value computed by the program under test and the output   |
|                            |   value stored in the test suite. See Section 8.5.4 for details.                                        |
| In the line number bar     | Toggle Breakpoint  Enable or disable breakpoint for the line.                                             |
| to the left of the main    |                                                                                                         |
| panel.                    |                                                                                                         |

Double Clicking...

- in the line number bar to the left of the main panel toggles a breakpoint for
  the line.
- on any other item opens a separate window which contains the same in-
  formation displayed when hovering on that item. This is useful when the
  information in in the hover window is clipped due to excessive length.

2Depending on a number of factors including the type of a right-clicked data item, some menu items may
not be available. For example, scopes are not available for arrays
Except for the documented exceptions related to editing .rsh 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.
- **Clear Watched Variables.** Remove all items from the watch list.
- **Open Difference Scopes...** Open a difference scope on one or more harness outputs. This item is only enabled when the input source is a test suite. See Section 8.5.4 for details.
- **Close All Scopes.** Close all open scopes.
- **Open Scope (Signal Group)** Open a scope on a signal group. A signal group is created by clicking the save button 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 a 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 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.

**Simulate menu.** The following entries are available when Simulator is enabled.

- **Simulator on/off.** Enable or disable Simulator. When disabled, Simulator behaves as a source code viewer; that is, the code can be viewed but simulation capabilities are disabled.
- **Fast Run with Report...** Execute a fast simulation simulation and produce a report. See Section 8.3 for details.
- **Fast Run.** Same as window item 14.
- **Run.** Same as window item 13.
- **Step.** Same as window item 12.
- **Step Into.** Same as window item 9.
- **Step Over.** Same as window item 10.

---

3 Any operation which modifies the .rsh file is disabled when Simulator is enabled. Several entries from the Edit menu open the Reactis Harness Editor in read-only mode.
CHAPTER 8. REACTIS SIMULATOR

Step Out Of. Same as window item 11.
Stop. Stop a fast or slow simulation run.
Reverse Step Into. Same as window item 8.
Reverse Step Over. Same as window item 7.
Reverse Step Out Of. Same as window item 6.
Back. Same as window item 5.
Fast Back. Same as window item 4.
Reset. Same as window item 3.
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 statements.
Update Configuration Variable... Initiates a dialog for changing values of configuration variables, which are global or static local 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 \(\mathcal{R}\), 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 17.
Open. Same as window item 18.
Save. Same as window item 19.
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 8.7.2.
Export... Export the current .rst file in different formats. Exporting is described in more detail in Section 8.7.1.
Create... Launch Reactis Tester. See Chapter 9 for details.
Update... Create a new test suite by simulating the current code using inputs from the current test suite, but recording values for outputs generated by the code. This feature is described in Section 8.8.
Browse... Open a file selection dialog, and then launch the Test-Suite Browser on the selected file. See Chapter 11 for details.
8.2. MENUS

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 16). Note that the new test will not be written to an .rst file until the current test suite has been saved using window item 15 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 19 or the Test Suite → Save menu item.

Compare Outputs. Specify whether or not Simulator should compare the simulation results against the results contained in the test suite being executed. When enabled if a difference is detected then the difference between the computed value and the value stored in test suite is reported in a warning.

Coverage menu. The Coverage menu contains the following entries. Details about the different coverage objectives may be found in Chapter 7. 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. All targets covered by the portion of the current test executed so far, plus those targets exercised in previous tests are listed as covered.

Show Summary. Open the coverage summary dialog shown in Figure 8.12.

Show Details. Toggle the reporting of coverage information by coloring, underlining, or over-lining code elements in the main panel.


Highlight Decisions, Conditions, Decisions, MC/DC, MCC, Boundaries, User-Defined Targets, Assertion Violations, C Statements. Each of these menu entries corresponds to one of the coverage metrics tracked by Reactis and described in Chapter 7. When a menu entry is selected and Show Details is selected, any uncovered target in the corresponding coverage criterion 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 that the target will never be covered prior to simulating the code. The analysis used is conservative: marked items are always unreachable, but some unmarked items may also be unreachable.
8.3 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 output values stored in the test suite and those computed by the program under test.

![Reactis Test Execution Report Options](image)

Figure 8.2: The Reactis Test Execution Report Options dialog is used to select the items which appear in a test execution report.

When Simulate → Fast Run with Report... is selected, the dialog shown in Figure 8.2 will appear. This dialog is used to select the items which will appear in the report. The following items are labeled in Figure 8.2:

1. The Report Options panel is used to select optional report items, such as the date, pathnames of input files, etc.

2. When Include coverage report is selected in the Report Options panel, the Coverage
Metrics panel can be 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 lets you choose between HTML and XML output formats. There is also an option to preview the results before saving the report.

4. The **Difference limit** prevents reports for test runs with many output differences from becoming excessively long. Once the limit is reached, output differences are still counted but no details are included in the report.

5. The **Output** panel determines where the report will be saved. When the output format is HTML, you can choose to save the results for each test in a separate file instead of generating a single file which contains the entire report.

6. Clicking on this button opens the help dialog.

7. When you are satisfied with the selected report options, clicking on this button will close the dialog and start the simulation run.

8. 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 8.3. 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 **Openall.** See section 8.3.1 for details.
Figure 8.3: A test execution report can be generated by loading a test suite in Simulator and selecting Simulate → Fast Run with Report...

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 12.4).

8.3.1 Test Data Plots

Figure 8.4 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 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.

---

### 8.4 Specifying the Simulation Input Mode

Reactis Simulator performs simulations in a step-by-step manner: at each simulation step values are generated for each input, and resultant output values are reported. You control how Simulator computes input values using the Source-of-Inputs Dialog (window item 16 in Figure 8.1) shown in Figure 8.5. This dialog always includes the Random Simulation and User Guided Simulation entries; if a test suite has been loaded, then the dialog 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 input, Reactis for C randomly selects a value from the set of allowed values for that variable, using type and probability information contained in the associated .rsh file. See Chapter 6 for a description of how to enter this information using the Reactis Harness Editor.

**User Guided Simulation.** You determine the value for each input using the Next Input Value dialog, which appears when the User Guided Simulation entry is selected. See Section 8.4.1 below for more information on this mode.

---

<sup>4</sup>only the first three are shown for inputs
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 on 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. 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 code execution is reset to its starting configuration, although coverage information is not reset, thereby allowing users 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 code execution is reset to its starting configuration, although coverage information is not reset, thereby allowing you to view cumulative coverage information for the entire test suite. Section 8.4.2 contains more information on this 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

Figure 8.5: The Source-of-Inputs Dialog enables you to specify how Simulator computes input values.
8.4. SPECIFYING THE SIMULATION INPUT MODE

the Steps column. Clicking again on that header will sort by number of steps in descending order.

You may also use the Source-of-Inputs 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.

8.4.1 User Input Mode

When the User Guided Simulation mode is selected from the Source-of-Inputs dialog, you provide values for inputs at each execution step. This section describes how this is done.

![Figure 8.6: The Next Input Values dialog lets you control the simulation by specifying the next value for inputs (item 4) and clicking the stepping buttons (item 10).](image)

To enter the user-guided mode of operation, select User Guided Simulation from the Source-of-Inputs dialog (window item 16 in Figure 8.1). Upon selecting user-guided mode, a Next Input Values dialog appears, as shown in Figure 8.6, that allows you to specify the input values for the next simulation step. Initially, each top-level input 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 8.6). The toolbar for the dialog contains items 7-13. The elements of the dialog work as follows:

1. The name of an item (input, output, test point, or configuration variable).

2. This check box 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 input is specified:
**Random** Randomly select the next value for the input from the type given for the input in the .rsh file.

**Entry** Specify the next value with the text-entry box in column four of the panel.

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 \( \sin(t) \times 0.001 \). For the full description of the expression notation see Section 8.4.1.1 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 input 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 input. The arrows are available for inputs 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 contains 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 mode for all inputs at once to either “Random” or “Entry.”

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 configuration.

When “run fast simulation” (window item 14 in Figure 8.1) is selected, the input value specifications in the Next Inputs Values dialog are used for each step in the simulation run.
8.4.1.1 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 grammar shown in Figure 8.7.

8.4.2 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 toolbar to the right of Source-of-Inputs Dialog (window item 18 in Figure 8.1) 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 for C 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, after each simulation step, Simulator compares the values computed by the code against the values stored in the test suite for those items. Any difference is flagged if it exceeds the tolerance specified for that output. See Section 6.5 for more information on specifying tolerances for outputs.

8.5 Tracking Data-Item Values

Reactis Simulator includes several facilities for interactively displaying the values that data items 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 global or static local variables in order to display their values at the end of a simulation step plotted on a graph with time on the horizontal axis. Scopes let you easily see how a variable changes during a simulation run. Distribution scopes enable you
\[ NIV : \text{numericConstant} \mid \text{true} \mid \text{false} \]
\[ \mid \text{inputName} \]
\[ \mid u \]
\[ \mid \text{pre}(NIV) \]
\[ \mid \text{pre}(NIV,n) \]
\[ \mid t \]
\[ \mid NIV \text{ relOp } NIV \]
\[ \mid ! NIV \]
\[ \mid NIV \& \& NIV \]
\[ \mid NIV \mid \mid NIV \]
\[ \mid - NIV \]
\[ \mid NIV \text{ arithOp } NIV \]
\[ \mid \text{function} (NIVL) \]
\[ \mid \{\text{fieldL}\} \]
\[ \mid NIV \cdot \text{fieldName} \]
\[ \mid \text{if } NIV \text{ then } NIV \text{ else } NIV \]
\[ \mid (NIV) \]

\[ \text{relOp} : < \mid <= \mid == \mid != \mid >= \mid > \]

\[ \text{arithOp} : + \mid - \mid * \mid / \]

\[ \text{field} : \text{fieldName} = NIV \]

\[ \text{function} : \text{abs} \mid \text{fabs} \mid \text{sin} \mid \text{cos} \mid \text{tan} \mid \text{asin} \mid \text{acos} \mid \text{atan} \mid \text{atan2} \mid \text{sinh} \mid \text{cosh} \mid \text{floor} \mid \text{ceil} \mid \text{hypot}(a,b) \mid \text{ln} \mid \text{log} \mid \text{log10} \mid \text{pow} \mid \text{exp} \mid \text{rem} \mid \text{sgn} \mid \text{sqrt} \]

\[ NIVL : \text{list of } NIV \text{ delimited by }, \]

\[ rowL : \text{list of } NIVL \text{ delimited by } ; \]

\[ fieldL : \text{list of } field \text{ delimited by }, \]

Figure 8.7: The grammar of next input value expressions.
8.5. TRACKING DATA-ITEM VALUES

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 harness outputs when reading inputs from a test in order to plot the values computed by the program under test against the values stored in the test for the output.

You may add data items to the watch list, or attach scopes to them, by right-clicking on a data item in the Reactis main panel and selecting an entry from the resulting menu as described in Section 8.1.

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. You 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....**

### 8.5.1 The Watched-Variable List

The watch list is displayed in a panel at the bottom of the Simulator screen as shown in Figure 3.12. 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 8.2. 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 program under test 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.

### 8.5.2 Scopes

Scopes appear in separate windows, an example of which may be found in Figure 8.8. The toolbar of each scope window contains nine or more items.

#### 8.5.2.1 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 toolbar for each data item (window items 10 and 11 in Figure 8.8). 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.

### 8.5.3 Distribution Scopes

Distribution scopes also appear in separate windows, an example of which may be found in Figure 8.9. 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 scopes plot the values a data item has assumed during simulation.](image)

### 8.5.4 Difference Scopes

When executing tests from a test suite, a difference scope may be opened by right-clicking on a test harness output and selecting **Open Difference Scope**. The resulting scope plots the expected value (from the test) against the actual value (computed by the program under test), as shown in Figure 8.10. If the difference between the two values exceeds the tolerance specified for the output (see Section 6.4.1), 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 8.11. 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.

Figure 8.10: A difference scope may be opened by right-clicking on a test harness output and selecting Open Difference Scope. The scope plots the values stored in a test for an output and the values computed by the program for the output. Differences are flagged in red.

8.6 Tracking Code Coverage

Chapter 7 describes the coverage metrics that Reactis for C employs for measuring how many of a given class of syntactic constructs or coverage targets that appear in the code have been executed at least once. Simulator includes extensive support for viewing this coverage information about the parts of the code 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.

8.6.1 The Coverage Summary Dialog

The Coverage Summary Dialog shown in Figure 8.12 may be invoked at any time Simulator is enabled by selecting Coverage → Show Summary. The dialog reports summary statistics for each coverage criterion tracked by Reactis. Each row in the dialog corresponds to one of the criterion and includes five columns described below from left to right.

1. The name of the coverage criterion reported in the row.
2. The number of targets in the criterion that have been exercised at least once.

3. The number of targets in the criterion 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 criterion that have not been exercised.

5. The percentage of reachable targets in the criterion that have been exercised at least once.

### 8.6.2 Coverage Information in the Main Panel

Selecting Coverage → Show Details toggles the highlighting of uncovered targets in the main panel. Targets that have been covered are drawn in black, while red implies an uncovered target. Please refer to Chapter 7 for a detailed description of how the different coverage metrics are highlighted in the main panel.

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 run which has not yet been added to a test suite.

For items included in the decision, condition, MC/DC, and MCC metrics, detailed coverage information may be obtained by right-clicking on the item and selecting View Coverage Details. A dialog similar to the one shown in Figure 8.13 will appear with coverage information. This dialog has two tabs, one titled Decision which contains Decision, Condition, and MC/DC details, and a second named MCC which contains MCC details.

Item 1 in Figure 8.13 shows the coverage details dialog with the Decision tab selected. The table within this tab describes the coverage status of all decision, condition, and MC/DC targets for the following decision:

\[ g_{dsMode}=M_{INIT} \land (set \land !deactivate) \]

This decision contains three conditions: \( g_{dsMode}=M_{INIT} \), set, and \(!deactivate\). Conditions are the atomic Boolean expressions that are used in decisions (for more details on decisions and conditions, see Sections 7.4 and 7.5).
Each decision details table contains seven columns, numbered 1-7 in Figure 8.13. These are interpreted as follows:

1. The test/step during which the decision first evaluated to true.
2. The test/step during which the decision first evaluated to false.
3. The conditions from which the decision is composed.
4. The test/step during which the condition first evaluated to true.
5. The test/step during which the condition first evaluated to false.
6. The condition values and test/step during which an MC/DC target was covered and the decision evaluated to true.
7. The condition values and test/step during which an MC/DC target was covered and the decision evaluated to false.

Note that although all targets were covered in Figure 8.13, this will not necessarily be the every time you view the details of a decision, in which case a test/step value of \(-/-\) will indicate that a target has not yet been covered.

MC/DC requires that each condition independently affect the outcome of the decision in which it resides. (See Section 7.6 for additional details.) When a condition has met the MC/DC criterion in a set of tests, columns 6 and 7 of the table explain how. Each element of these two columns has the form $bbb: test/step$, 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 8.14 shows the coverage details dialog with the MCC tab selected. Each row of this table corresponds to a single MCC target. The table columns are interpreted as follows:

1. Each of these columns corresponds to a condition. Every condition within the decision is always represented by a single column in the table. 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.

2. The next-to-last column contains the decision outcomes for each MCC target.

3. The last column gives the test/step in which the MCC target was covered.

![Figure 8.15: Filtering MCC coverage information.](image)

MCC coverage details can be filtered by clicking on the column headers, as shown in Figure 8.15. A filtered column header is indicated by a prefix of \(T:\), \(F:\), or \(x:\), which correspond to the column values **True**, **False**, and **x**, respectively. In Figure 8.15, items 1 and 2 refer to columns with active filters:

1. Only MCC targets for which the condition \(g_{dsMode}==M\_INIT\) evaluated to true are being shown, as indicated by the \(T:\) prefix in the header for this column.

2. Only MCC targets for which the decision evaluated to false are being shown, as indicated by the \(F:\) prefix in the header for this column.

Clicking on a column header advances the filter setting for that column to the next feasible value, eventually cycling back to unfiltered. All columns can also be reset to the *unfiltered* state at any time by clicking on the **Clear Filter** button (item 3 in Figure 8.15). 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.
8.6.3 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 12.

8.7 Exporting and Importing Test Suites

8.7.1 Exporting Test Suites

The export feature of Reactis for C 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 8.16). 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, configuration variables) should be included in each test step. The following formats are currently supported:

- **.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, and configuration variables should have values recorded. If the Compress output check box is selected, then test steps will be omitted if no item that would be recorded in the step is different from the corresponding value in the previously recorded step. This is especially useful when exporting only input data for a test in which inputs are held constant for a number of steps.
The first line of an exported file will contain a comma separated list of the names of the harness inputs, outputs, 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.  

After the first line, 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. For array valued items, the elements of the array appear within double quotes (") as a comma-separated list.

- An empty line signaling the end of a test.

.\_txt\_ files: Suites may be saved in a more verbose plain ASCII format to facilitate inspection of the test data.

### 8.7.2 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 currently selected harness. A test suite matches a harness if it contains data for the same set of inputs, outputs, and configuration variables as the harness. If an externally visible data item (input, output, or configuration variable) is added to or removed from the harness, then previously constructed test suites will no longer match the new version of the harness. The import facility gives you a way to transform the old test suite so that it matches the new version of the harness. Such remapping is also available when importing .\_csv\_ files.

The Import Dialog, shown in Figure 8.17, 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, 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. 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 input variable X and is being imported with respect to a harness that has no input variable X, then X will appear at the bottom of the Input variables tab and be highlighted in yellow.

Each data item tab (e.g. Inputs) includes a column (e.g. Harness Input Name) listing the code elements in that category. The Suite column lists items from the file being imported that map to the corresponding harness 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 harness. If the harness 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 8.17 for input brake). If this setting is not changed, then upon import a random value
will be generated for the input at each test step. The value to be mapped to any harness 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.

![Import Test Suite Dialog]

Figure 8.17: The Import 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 harness. The import facility is also used to transform an .rst file to make it match a program under test.

---

### 8.8 Updating Test Suites

Test Suite → Update... creates a new test suite that retains the inputs from the current test suite, but updates the outputs with values computed by the execution of your code on the existing inputs. This provides a means to reuse previously constructed test suites when your code changes in a way that causes its externally visible behavior to change.
Chapter 9

Reactis Tester

Tester automatically generates test suites from C programs. Tests are formulated according to a test harness constructed by the user as described in Chapter 6. Among other items, a harness includes an initialization function, an entry function, as set of inputs, and a set of outputs. Each test in a suite consists of a sequence of steps formulated as follows:

1. If this is the first step, call the initialization function.
2. Select a value for each input.
3. Execute the entry function based on the inputs.
4. Record the values computed for the outputs.

The test suites are intended to maximize different coverage objectives while minimizing the total number of execution steps in the overall suite. The remainder of this chapter describes the workings of Tester.

9.1 The Tester Launch Dialog

Figure 9.1 contains an annotated screenshot of the Tester launch dialog, which may be invoked from the Reactis for C 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 suites to be preloaded are given, new test data will be generated from scratch.
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
9.1. THE TESTER LAUNCH DIALOG

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. 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 for C to run out of memory. The upper bound on the number of steps possible depends on the size of the program under test, but in general much more time should be spent in the targeted phase which is more optimized for memory usage. |

9. 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 program under test 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.

10. 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 programs under test to exercise. The metrics supported by Tester are described in Chapter 7.

11. 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.

- `-C n` directs Reactis for C to use n cores during test generation. Currently supported values for n are 1 and 2. Leveraging multi-core architectures speeds up test-generation in many cases.

- `-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.
12. The name of the .rst file to be generated.

13. Clicking this button opens a file-selection dialog for specifying the name of the .rst file to be generated.

14. Clicking this button displays Tester help.

15. Clicking this button opens a file selection dialog to specify an .rtp file from which to load Tester launch parameters. Reactis for C may be configured from the Settings dialog to generate an .rtp file for each Tester or Validator run.

16. Reset the Tester launch parameters to the default values Reactis for C ships with.

17. Scroll backward in the parameter history.

18. Scroll forward in the parameter history.

19. This button is visible only if you went back to this screen by clicking the Back button in the Tester progress dialog. Clicking it will bring back the progress dialog with its previous results.

20. Clicking this button starts test-suite generation.

21. Clicking this button closes the Tester launch dialog without starting test-generation.

---

9.2 The Progress Dialog

The Tester progress dialog, shown in Figure 9.2, is displayed while Tester is running. What follows describes the labeled items in the figure. When Tester completes, buttons 5–12 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 each enabled metric. Coverage metrics are described in Chapter 7.

5. When warnings occur, this button may be clicked to view descriptions of the warnings.

6. View help for the progress dialog.
7. Open the Reactis Test Suite Browser and load the newly generated test suite.

8. Enable Reactis Simulator and load the newly generated test suite.

9. Enable Reactis Simulator, load the newly generated test suite, and close the Tester progress dialog.

10. View the program under test, highlighted with coverage results from the newly generated test suite.

11. Go back to the Tester launch dialog.

12. View coverage results for the newly generated test suite in the Coverage-Report Browser.

13. Interrupt test generation, save tests generated so far, and close the Tester progress dialog.
Reactis Validator searches for defects and inconsistencies in your program. The tool enables you to formulate a property that program behavior should have as an assertion 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 offers a mechanism called *user-defined targets* to specify specific test scenarios you want to create. User-defined targets may be seen as user-defined extensions to the built-in coverage metrics (statement, decision, condition, MC/DC, etc.) supported by Reactis: when generating test data, Tester will attempt to generate tests that satisfy the indicated scenarios.

Validator is particularly useful in requirements validation. Given a list of requirements on program behavior, you can formulate assertions to check whether the requirements are being satisfied and user-defined targets to define test scenarios intended to “stress” the requirements.

You use Validator as follows. First, you insert assertions and user-defined targets into your source code using the `reactis_assert` and `reactis_target` macros. We refer to assertions and user-defined targets as *Validator objectives*. Note that when using the instrumentation code mechanism described in Section 10.3, you can insert Validator objectives from within the Reactis GUI without modifying your application source code. The objectives and their location within the source code is managed by Reactis and stored in the `.rsm` file. Reactis Tester is then invoked on the instrumented code to search for assertion violations and paths leading to the specified coverage targets. The output is a test suite that includes tests leading to objectives found during the analysis.

The `reactis_assert` and `reactis_target` macros are defined in `reactis_validator.h` so this header file must be included in the C files that use the Validator macros.

As an alternative to `reactis_assert`, you can use the `assert` macro that is part of the ANSI C standard. To make Reactis treat `assert` macros as if they were `reactis_assert`, simply select the *Create coverage targets for assert() statements* setting in the C code tab of the Reactis Settings Dialog.

If you include `NDEBUG` as a Define in an `.rsm` file, then any assertion checking will be turned off for any `assert`, `reactis_assert`, and `reactis_target` macros in the C files and
libraries of the component.

## 10.1 Validator Objectives

A **Validator objective** is a target created by Validator. Coverage of Validator objectives is tracked in the standard manner, namely through highlighting in the main Reactis panel, the Coverage-Report Browser, and generated reports. Also, in Simulator, hovering over an objective shows its coverage status and, if covered, the test and step when it was first covered. There are two types of Validator objectives, **assertions** and **user-defined targets**.

### 10.1.1 Assertions

Assertions specify properties that should always be true for a program under test. If the argument to an `reactis_assert` macro evaluates to a non-zero value we say it holds; if the argument evaluates to zero we say it is violated or does not hold. For a given assertion, Validator searches for a simulation run that leads to a configuration in which the assertion does not hold. 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.

### 10.1.2 User-Defined Coverage Targets

User-defined coverage targets extend the Reactis built-in targets described in Chapter 7 (Statement, Decision, Condition, MC/DC, etc). Reactis will consider a user-defined target to be covered when the argument to a `reactis_target` macro evaluates to a non-zero value.

## 10.2 Use Cases of Validator Objectives

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 user-defined target:

```
reactis_target(on && activate && (speed < 30))
```

This objective would be inserted at a location of the code that executes during every execution of the entry function 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 test our requirement.
10.2. USE CASES OF VALIDATOR OBJECTIVES

We can test whether the application’s response to this test is correct with the following assertion:

```c
reactis_assert(!(active && (speed < 30)))
```

This assertion is inserted so that `active` monitors whether the cruise control is active. If this expression ever becomes false, then the requirement is violated.

The assertion and user-defined target that capture the simple requirement of the previous example depend only on the values of program variables at a given point in time. Note, however, that more complex properties involving timing can be constructed by creating Validator objectives that maintain state in between simulation steps. 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 an assertion that monitors the vehicle speed, whether the cruise control is active, and the speed at which the driver has set the cruise control. A convenient way to implement such an assertion is as a C function with three input parameters `active`, `speed`, and `desiredSpeed` and returning a zero value if the requirement is violated.

The C function that implements the assertion is shown in Figure 10.1. It implements a simple state machine in which the static local variable `state` records the current state. If state `SPDHECK_ACTIVE_ERROR` is ever reached, then the assertion is considered violated.
```c
int cruise_assert_spdCheck(int active, double speed, double dSpeed)
{
    static int state = SPOCHECK_NOTINIT, cnt;

    switch( state )
    {
    case SPOCHECK_NOTINIT:
        /* Stateflow chart starts out not initialized, first step is 
        taken up with initializing it */
        state = SPOCHECK_INACTIVE;
        break;

    case SPOCHECK_INACTIVE:
        if( active ) state = SPOCHECK_ACTIVE_OK;
        break;

    case SPOCHECK_ACTIVE_OK:
        if( !active )
            state = SPOCHECK_INACTIVE;
        else if( fabs(speed-dSpeed) > 1 )
        {
            state = SPOCHECK_ACTIVE_BIGDIFF;
            cnt = 0;
        }
        break;

    case SPOCHECK_ACTIVE_BIGDIFF:
        if( !active )
            state = SPOCHECK_INACTIVE;
        else if( cnt>3 )
            state = SPOCHECK_ACTIVE_ERROR;
        else if( fabs(speed-dSpeed) <= 1 )
            state = SPOCHECK_ACTIVE_OK;
        cnt++;
        break;

    case SPOCHECK_ACTIVE_ERROR:
        if( !active )
            state = SPOCHECK_INACTIVE;
        break;

    }

    return (state!=SPOCHECK_ACTIVE_ERROR);
}
```

Figure 10.1: The implementation of the speed maintenance assertion.
There are two ways to add Validator objectives to a program under test. One way is to insert `reactis_target` and `reactis_assert` statements directly within the source file (typically enclosed with `#ifdef __REACTIS__` ... `#endif`). Another way is to add instrumentation code from within Reactis. A block of instrumentation code is just one or more lines of C code that is “virtually” inserted at some location within your application source code. Each block of instrumentation code and its location are stored in the build file and integrated with the source code at compile time. This allows you to add Validator objectives (and possibly other test code) to the program under test without modifying the source code.

### 10.3.1 Advantages and Disadvantages of Instrumentation Code

Instrumentation code has the following advantages over directly modifying the source code.

1. If you normally save the Validator objectives in the code under test, using instrumentation code will eliminate the risk that the Validator objective code will affect the code under test outside of Reactis, such as causing a compilation failure or changing runtime behavior.

2. The instrumentation code is highlighted within Reactis, making it easy to distinguish from the rest of the code.

3. If you normally add the same Validator objectives each time a new revision of the code is tested, Reactis will automatically insert the instrumentation code at the appropriate locations with each revision if possible.

The main disadvantage of instrumentation code is that if a source file is modified, there is a chance that Reactis won’t be able to automatically relocate instrumentation code within the file. In this case you will need to manually place the instrumentation code in the correct location; but, in such cases Reactis will flag instrumentation code with a warning symbol to alert you that it may require relocation.

### 10.3.2 Identifying Instrumentation Code

By default, instrumentation code is highlighted in the main panel with a colored background, as shown in Figure 10.2. Each line of instrumentation code is identified by the number of the line in the source file which immediately precedes the block of instrumentation code, followed by a plus sign (+) and then the number of the line within the block. For example, the lines for the instrumentation code in Figure 10.2 are identified as 106 + 1, 106 + 2, ... 106 + 5. This double line number is displayed when hovering on the line number of a non-empty line of instrumentation code, as shown in Figure 10.2.
10.3.3 Adding Instrumentation Code

When Simulator is disabled, instrumentation code can be added by right-clicking on the line number of a regular source code line (i.e., any line which does not contain instrumentation code), and selecting Add Instrumentation Code. This will open an editor window similar to the one shown in Figure 10.3.

10.3.4 Editing Instrumentation Code

Instrumentation code can be edited by right-clicking on the line number of any line of instrumentation code and selecting Edit/Move Instrumentation Code. This will open an instance of the C Code Editor for the block of instrumentation code which was clicked on, as shown in Figure 10.3. The following numbered items appear in Figure 10.3:

1. The text of the instrumentation code is edited within this panel.
2. This drop-down list is used to determine if the code will be placed before or after the line number shown to the right (item 4).

3. This button opens the help dialog for the C Code Editor.

4. This entry box contains the line number where the instrumentation code will be placed. The placement will be either before or after the line depending on the setting of item 2.

5. This button saves all changes and closes the editor.

6. This button discards all changes and closes the editor.

### 10.3.5 Cutting, Copying, and Pasting Instrumentation Code

Instrumentation can be cut, copied, or pasted directly within the main panel of the Reactis top-level window. Before a cut, copy, or paste operation can be performed, one or more lines of instrumentation code must be selected. To select a single line of instrumentation code, left-click on the line number. To select multiple lines of instrumentation code, left-click on a line number, then shift left-click on a second line number within the same block of instrumentation code. This will select all lines between (and including) the lines which were clicked on. Selected lines are highlighted with a dark blue background.

Once one or more lines of instrumentation code have been selected, right-clicking on any line number and selecting **Copy Instrumentation Code** will copy the selected lines to the clipboard. Right-clicking on any line number and selecting **Cut Instrumentation Code** will copy the selected lines to the clipboard and then delete them.

Any time the clipboard is not empty, right-clicking on a line and selecting **Paste Instrumentation Code** will insert the instrumentation code containing the contents of the clipboard immediately before the line which was clicked on.

### 10.3.6 Verifying the Location of Instrumentation Code

Since instrumentation code is not stored in a source file, it is possible that changes to a source file will cause the location of instrumentation code within the source file to change. Reactis stores information about the surrounding context of each block of instrumentation code and automatically adjusts the starting location each time the file is loaded. If the lines immediately surrounding the instrumentation code have not changed, then the relocation will typically succeed (there will be a warning message stating that the location has changed). The new location of the instrumentation code does not take full effect until you select **Edit → Save Instrumentation Code**.

After some source file modifications, Reactis will be unable to accurately determine the new location of an instrumentation code block. In this case there will be a warning message stating that the location could not be determined and the current location is unreliable. Blocks of instrumentation code with unreliable location are shown with red text in the .rsm file Editor (see Section 5.6.1). In the main window, instrumentation code with an unreliable location is indicated by a yellow background and warning symbol in the line
number column. The location can be verified by either moving the instrumentation code (see Section 10.3.7), or right-clicking on a line number within the block and selecting Keep Instrumentation Code Here.

10.3.7 Moving Instrumentation Code

The easiest way to move instrumentation code to a new location is to right-click on any line number within the instrumentation code block and select Edit/Move Instrumentation Code. The location can then be changed within the C Code Editor (items 2 and 4 of Figure 10.3).

Alternatively, you can also move a block of instrumentation code by selecting the entire block and then cutting and pasting it to a new location.

10.3.8 Removing Instrumentation Code

The easiest way to remove a block of instrumentation code from a source file is to right-click on any line number within the instrumentation code block and select Remove Instrumentation Code.

Alternatively, you can also remove a block of instrumentation code by selecting the entire block and then cutting it.
Chapter 11

The Reactis Test-Suite Browser

The Test-Suite Browser allows you to view test suites. Figure 11.1 contains an annotated screen shot of the Test-Suite Browser, which displays the test selected in the test selector button (window item 12). The body of the window consists of three tabs:

Test Data Displays a matrix whose first column lists the inputs and outputs of the test harness and whose subsequent columns each represent a simulation step. Within each of these latter columns there is a value for each input and output.

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.

Figure 11.1: The Reactis Tester test-suite browser.
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 input or output assumes in the test suite, as shown in Figure 11.2. 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 input/output 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. If either of the Test History or Suite History tabs are selected, copy any of the selected log information to the clipboard.

4. Go back 10 steps in the current test.

5. Go back 5 steps in the current test.

6. Go back 1 step in the current test.

7. Specify a step in the current test to view.

8. Go forward 1 step in the current test.

9. Go forward 5 steps in the current test.

10. Go forward 10 steps in the current test.


12. 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.

13. Checking this box activates the filter. When a filter is active only steps for which the filter expression evaluates to true will be displayed.

14. 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 ‘brake == 1’ and press return.

The syntax for filter expressions is ANSI C. The variables in the expression may be the names of inputs or outputs from the test suite. The elements of an array 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:

<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 inp are positive</td>
</tr>
</tbody>
</table>

15. Clicking this button activates the filter. When a filter is active only steps for which the filter expression evaluates to true will be displayed.

16. Clicking this button opens the Filter Editor dialog shown in Figure 11.3. The editor lets you formulate more complicated filters.

17. This column lists the inputs and outputs of the test harness.

18. Values of inputs and outputs in step 1 of the current test.

19. Values of inputs and outputs in step 2 of the current test.

20. This column lists the name of each configuration variable.

21. This column lists the value of each configuration variable in the current test.

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 Item)...** Specify how many significant digits should be used when displaying the values flowing over the currently selected input/output. 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 inputs and outputs in the harness. 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 item, open a distribution scope to display (as shown in Figure 11.2) the values the item assumes when executing the test suite.

![Figure 11.2: The values assumed by input “drag”. You may zoom in by clicking in the body of the dialog. Clicking the button in the toolbar zooms to fit.](image)

- **Open Distribution Scope (Current Test).** For the currently selected item, open a distribution scope to display the values the item 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 item, open a distribution scope to display the values the item 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.
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.

**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 11.3 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.** Stop hiding the currently selected rows. Note, to select hidden rows you must first toggle the **Show Hidden Rows** menu entry.

**Unhide All Rows.** Cause all rows to be displayed.

**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.

*Test-Suite Browser.* Display Test-Suite Browser help.

### 11.3 Test Step Filter Editor

The **Test Step Filter Editor**, shown in Figure 11.3, lets you formulate filters to search for test steps in a suite that satisfy a given 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 inputs and outputs from the test suite. The elements of an array 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:

<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 inp are positive</td>
</tr>
</tbody>
</table>

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 an array 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 an array of length 3, then we can check for steps in which all elements of inp are positive with the filter expression:
(inp[0] > 0) && (inp[1] > 0) && (inp[2] > 0)

4. Each menu in this column includes an entry for each input and output and the sample time in the test suite. The item selected will be fed into the filter 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 (menu in the first column selected) to the input or output 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

The Reactis Coverage-Report Browser

The Coverage-Report Browser enables users to view detailed coverage information related to testing and to export the reports in HTML format for viewing and printing. Figure 12.1 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 100 of test 12 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.

12.1 Labeled Window Items

1. The hierarchy panel works much as the hierarchy panel does in the top-level Reactis window. Clicking the ▼ to the left a project component name displays the child components of the item. Clicking on a .c or .rsm file causes coverage information for the component to be displayed in the panels to the right. Double-clicking on a name causes the source code to be displayed in the main panel of the top-level window.

2. This panel displays summary coverage statistics for the item currently selected in the hierarchy panel.

3. This panel shows coverage information for each child of an .rsm file selected in the hierarchy panel. Since C files have no children this panel is blank when a C file is 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 metrics tracked by Reactis for C. Double-clicking on a row causes the child to become the current component 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 gives detailed information for each coverage target in the item currently
selected in the hierarchy panel. If an .rsm file is currently selected in the hierarchy panel, then this panel is empty, since an .rsm file does not directly contain coverage targets. If a C file is selected in the hierarchy panel, then this panel displays a list of coverage targets in the C file. Each row in the list displays whether or not a target has been covered. If it has been covered, the test and step in which it was first exercised are also displayed. Double clicking on a row causes the selected target to be highlighted in yellow in the main panel of the top-level window.

Note that decision, condition, MC/DC targets are grouped into a single row. The MCC targets for a decision are also grouped into a row (distinct from the decision, condition, MC/DC row). The status columns for these aggregate rows display covered when all targets in the associated group 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 8.13 will appear. See Section 8.6.2 for a description of this dialog.

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, the user simply clicks on the header of the column. Clicking once sorts in increasing order of the column entries, while
12.2 Menus

Report menu. The Report menu contains the following entries.

- **Export...** Invoke the dialog shown in Figure 12.2 that enables the user to export the coverage report in HTML format. See Section 12.3 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 in-tool documentation.
- **Index.** Go to the index in the in-tool documentation.

12.3 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 12.2 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 component (.rsm file or C file) of the project has a section in the report; a component’s section includes coverage information for the targets in that component.

The labeled items in the dialog are used as follows.

1. The radio button selected from this group specifies the portion of the project 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 of 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 .rsm file file and .rst file are included at the top of the report. 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. This section lets you configure which coverage metrics will be included in the generated report. Each metric has a row with radio buttons to select from three options for the metric:
12.4 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 12.3. For each .rsm and .c file, the report will include a summary section (coverage statistics for the file) and details section.
12.4. COVERAGE REPORT CONTENTS

(detailed information about the coverage of each target in the file). The Export to HTML dialog lets you specify which coverage metrics should be included in the summary and detail sections.

Figure 12.3: Example exported HTML report

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 12.4. 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 $S:excluded$ means the statement has been excluded from coverage tracking.
CHAPTER 12. THE REACTIS COVERAGE-REPORT BROWSER

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.
\( e \) 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.

Figure 12.4: Code coverage symbols in HTML report

The indicators in Figure 12.4 can be interpreted as follows:

**Line 46**  $D:---$ means at least one decision, one condition, and one MC/DC target remain uncovered.

**Line 50**  $D:+++$ means all decision, condition and MC/DC targets have been covered.

**Line 52**  $D:+--$ means both decision targets have been covered, but at least one condition and one MC/DC target remain uncovered.

Although Reactis for C supports a very large subset of C language features and standard runtime library functions, not every C program can be immediately used with Reactis for C. Before using Reactis for C on a C program or program fragment, you must first ensure that the portion of the program to be used with Reactis for C does not depend on any unsupported features.

The following is a quick summary of the requirements for using Reactis for C:

- **No binary code.** Because Reactis for C compiles code into its own internal virtual machine code, only the portion of a program for which source code is available can be used with Reactis for C.

- **No threads.** Currently, Reactis for C does not support multi-threaded code. It is of course possible to extract single-threaded components of multi-threaded programs for testing with Reactis for C.

- **Must use the Reactis for C standard libraries.** Like most C compilers, Reactis for C provides its own implementation of the C runtime library (`libc`) and math library (`libm`). The C program under test will be automatically linked with these libraries. While most of the standard library functions are implemented, there are some unsupported library functions. These are listed in Section 13.2.

- **No file I/O.** The Reactis for C paradigm requires that all data passed to/from the code under test pass through the harness, so that test suites can be generated. Hence, reading or writing data via the C standard I/O library is not allowed.

- **No Clocks.** Use of the clock functions provided by `time.h` will result in tests which are not reproducible. For this reason, they should not be used in Reactis for C.

### 13.1 Basic type sizes and alignments

The sizes and alignments of the common built-in C types under Reactis for C are listed in Table 13.1. 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).
13.2 Unsupported C Features

Reactis for C 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); };`).

13.3 Supported C library functions

The following C library functions are currently provided by Reactis for C.

```c
assert.h
assert

ctype.h
isalnum isalpha isascii iscntrl iscsym iscsymf isdigit isxdigit isprint isgraph ispunct islower isupper isspace isblank toint tolower toupper

ererro.h
strerror
```
13.3. SUPPORTED C LIBRARY FUNCTIONS

fcntl.h
   fileno

fenv.h
   fegetround fesetround

math.h
   acos acosf acosl asin asinf asinl atan atan2 atan2f atan2l cbrt cbrtf cbrtl ceil
cceilf ceilf copysign copysignf copysignl cos cosf coshf coshl cosh coshf
coshl exp exp2 exp2f exp2l expf expl fabs fabsf fabsl fdim fdimf fdiml floor floorffloorl fmax fmaxf fmaxl fmin fminf fminl fmod fmodf fmodl fpclassify
frexp frexpf frexpl isfinite isgreater isgreaterequal isninf
isless islessequal islessgreater isnan isnormal isunordered ldexp ldexpf
ldexpl lrint llrint llrintf llrintl ln lnf lnl log log2 log2f log2l log10
go 10f log10l logf logl lrint lrintf lrintl lround lroundf
lroundl modf modff modfl nannanl nannanl nextafter nextafterf nextafterl
nexttoward nexttowardf nexttowardl pow powf powlf powm powqf remquo remquof
remquol round roundf roundl scalbln scalblnf scalblnl scalbnf scalbnf
scalbnl scalbnl signbit sin sinh sinhf sinh1 sinl sqrt sqrtf sqrtfl tan
tanf tanhl tanhl tanh trunc truncf truncl

memory.h
   bcopy memccpy

setjmp.h
   longjmp setjmp

stdarg.h
   va_arg va_copy va_end va_start

stddef.h
   offsetof

stdint.h
   INT8_C INT16_C INT32_C INT64_C INTMAX_C UINT8_C UINT16_C UINT32_C
UINT64_C UINTMAX_C

stdio.h
   snprintfsprintf

stdlib.h
   _Exit abort abs atof atoi atol atoll bsearch calloc div exit free
genenv labs ldiv llabs lldiv malloc mblen mbstowcs mbtowc qsort rand realloc
rando rand srand strtod strtof strtol strtold strtolllstrtolwstrtohl
wctombs

string.h
   memchr memcmp memcpy memmove memset strcat strchr strcmp strcspn
strdup strlen strncat strncmp strndup strpbrk strrchr strspn strstr strtok
strxfrm

tgmath.h
   acosl asin atan atan2 cbrt cceil copysign cos cosf coshf coshl cexp cexpf
cexp2 cexp2f cexp2l fabs fdim floor fma fmax fmin fmod frexp
def log log2f log2l log10f log10l logf logl lrint lrintf lrintl lround
lroundf
lroundl modf modff modfl nearbyint nextafter nexttoward pow remainder
remquo rint round scalbln scalbnln sin sinh
13.4 Unsupported Library Functions

The following C runtime library functions are not supported:

**complex.h**
- cabs cabsf cabsl cacos cacosf cacosh cacoshf cacoshl cacosl carg cargf cargl casin casinh casinhl casinl catan catanf catanh catanhf catanhl catanl ccos ccosf ccosl ccosh ccoshf ccoshl ccosl cexp cexpf cexpl cimag cimagf cimagl clog clogf clogl conj conjf conjl cpow cpowf cpowl cproj cprojf cprojl creal crealf creall csin csinl csinh csinhf csinhl csinl csqrt csqrtf csqrtl ctan ctanf ctanh ctanhf ctanhl ctanl
dlfcn.h
- dlclose dlerror dlopen dlerror
erno.h
- perror
fcntl.h
- creat fcntl open close read write lseek unlink
locale.h
- localeconv setlocale
math.h
- acosh acoshf acoshl asinh asinhf asinhl atanh atanhf atanhl erf erf c erf cf erfc erfcf erfc erfl expm1 expm1f expm1l fma fmal fmaf hypot hypotf hypotl ilogb ilogbf ilogbl lgamma lgammaf lgammaf lgammal log1p log1pf log1pl rint rintf rintl tgamma tgammaf tgammaf
setjmp.h
- sigsetjmp siglongjmp
signal.h
- raise
stdio.h
- clears err fclose feof fflush fgetc fgetpos fgets fopen fprintf fputc fputs fread freopen fscanf fseek fsetpos ftell fwrite getc getchar gets printf putc putchar puts remove rename rewind scanf setbuf setvbuf tmpfile tmpname ungetc vscanf vscanf vsscanf
stdlib.h
- atexit bsearch execl execlp execv execvp qsort system
tgmath.h
- carg cimag creal conj cproj
unistd.h
- access alarm brk chdir chown chroot close confstr crypt ctermid cuserid dup dup2 encrypt fchdir fchown fdatasync fork fpathconf fsync fprintf fputc ftell ftellf ftell64 ftell64f ftruncate getcwdu getdtablesize getegid
13.4. **UNSUPPORTED LIBRARY FUNCTIONS**

geteuid getgid getgroups gethostid getlogin getlogin_r getpagesize getpass getpgid getpgrp getpid getppid getsid getuid getwd isatty lchown link lockf lseek nice pathconf pause pipe pread pthread_atfork pwrite read readlink rmdir sbk setgid setpgid setpgrp setregid setreuid setsid setuid sleep swab symlink sync sysconf tcgetpgrp tcsetpgrp truncate ttyname ttyname_r ualarm unlink usleep vfork write

**utime.h**

utime

**wchar.h**

fgetwc fgetws fputwc fputws fwide fwprintf fwscanf getwc getwchar putwc putwchar swprintf swscanf ungetwc vfwprintf vwscanf vswprintf vswscanf vwprintf vwscanf wcschr wcsncmp wcsncrel wcsncpy wcsnrlen wcsncsscanf wcsncstr wcsnlen wcsncrtim wcsncswscanf wcsncmp wcsncpy wcsncmp wcsncmp wcsncpy wcsncpy wcsncmp wcsncpy wcsncpy wcsncmp wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsncpy wcsnç

**time.h**

asctime clock ctime difftime gmtime localtime mktime strftime
Chapter 14

Errors Detected by Reactis for C

14.1 Integer Overflows

An integer overflow occurs when an operation produces an integer result which is too large or too small to be represented by the destination type of the operation. When an integer overflow occurs, the course of action taken by Reactis for C depends on the setting for the operation which overflowed: Wrap Over or Error.

14.1.1 Wrapping over

When overflow behavior is set to Wrap Over, Reactis for C emulates the behavior of typical hardware, which is to truncate the most significant bits of the value until the value is small enough to be represented. The effect of wrapping over is different for unsigned and signed integers.

In the case of unsigned integers, wrapping over a value $x$ is typically equal to $x \mod 2^n$, where $n$ is the width (in bits) of the container type.

When signed values are wrapped over, the sign bit may change, producing results whose sign is the opposite of its value prior to wrapping over. For example, in most 32-bit environments, adding the maximum signed int value ($0x7fffffff$) to itself produces the value $0xffffffff$, wrapped-over result $-2$.

14.1.2 Error diagnosis

When overflow behavior is set to Error, Reactis for C stops execution as close to the statement where the overflow occurred as possible and displays an error dialog. You can hover on variables at this point to see their values and step backward to inspect earlier values if desired.

14.1.3 Controlling overflow behavior

The action taken when integer overflows occur is controlled by the four parameters in the C Code tab of the Global Settings dialog. (See Section 4.3.2 for details on how to set these parameters.) The following operations are controlled by each parameter:
On signed integer overflow  Signed integer arithmetic, signed integer shift, conversion from floating point to signed integer, conversion from signed integer to a smaller signed integer type.

On unsigned integer overflow  Unsigned integer arithmetic, unsigned integer shift, conversion from signed to unsigned integer, conversion from unsigned integer to a smaller unsigned integer type.

On overflow during conversion from unsigned to signed integer  Conversion from unsigned to signed integer. (Conversions from other types to signed integer are controlled by On signed integer overflow.)

On overflow during conversion from floating-point to unsigned integer  Conversion from floating-point to unsigned integer. (Conversions from other types to unsigned integer are controlled by On unsigned integer overflow.)

Note that Bit-level operations, such as bitwise-and (&), bitwise-or (|), etc. never produce overflows.

You should also be aware that, according to the C language specification, overflows during unsigned computations are wrapped over. This is different than the specification for signed integer computations, which states that the results for an overflow are undefined. Many C programs are designed to exploit the wrapping over of unsigned values, and setting On unsigned integer overflow to Error for such programs will result in superfluous errors during testing.

14.1.4 Determining the type of an expression

Reactis for C uses C99 rules for determining the type (signed or unsigned) of integer 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 U or u suffix, as determined by the following rules:

1. Any integer constant with a U or u suffix is unsigned.
2. Any decimal constant without a U or u suffix is signed.
3. Any non-decimal constant X without a U or u suffix will be assigned the first type that can represent X from the following list: int, unsigned int, long long int, unsigned long long int. This means that 0xffffffff will be assigned type unsigned int and 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 4294967295U, it will be assigned type unsigned int.

The C99 rules for evaluating integer expressions state that values of any type smaller than int, including unsigned types, are first converted to type 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 `((int)-1) > ((unsigned int)1)` will evaluate to 0 (false).

---

### 14.2 Floating-Point Errors

In C and most other languages, floating-point calculations which overflow do not trigger an exception, but instead produce a distinct value which represents positive or negative infinity (+\text{inf} or -\text{inf}). Similarly, calculations whose result is indeterminate, such as 0.0/0.0, produce a special value called \textit{not-a-number} (\text{nan}).

Reactis for C detects indeterminate and infinite values and will either raise an error, generate a warning message, or ignore the value, depending on the setting of \textit{When calculation results in NaN (Not-a-Number)} or infinity (see Section 4.3.2). However, an error or warning is not immediately produced whenever a calculation generates a infinite or indeterminate result. Instead the error/warning is produced when a result of \text{nan} or \text{inf} is assigned to a storage location which does not already hold an infinite or \text{nan} value. This is done to minimize the number of redundant warning messages produced while generating or executing a test suite.

---

### 14.3 Memory Errors

Whenever a pointer is used to access memory, Reactis for C 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. \textit{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 \text{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 an temporally invalid referent result in a \textit{temporal memory error}. 
int sum(int A[], int n)
{
    int i = 0, x = 0;
    while(i <= n)
    {
        x += A[i++];
    }
    return x;
}

Figure 14.1: A function with a spatial memory error.

Note that it is possible that a program which appears to function correctly actually contains a memory error. For example, consider the function `sum` shown in Figure 14.3. This function sums the first \( n + 1 \) element of \( A \), which will include 1 element past the end of the array when \( n \) equals the number of elements in \( A \). If the element which follows the end of \( A \) happens to always be zero, \( \text{sum} \) will function “correctly” in the sense that its return value is equal to what the programmer intended. At some point the future, however, a change to the program may cause the invalid memory access to return a non-zero value, which will cause the value returned by \( \text{sum} \) to be incorrect. 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.

14.3.1 Uninitialized Memory

In C, variables which are allocated in static memory (which includes all variables declared with the `static` keyword plus all variables declared outside the body of a function) are initialized to zero when there is no initial value given in the source code. Variables declared inside the body of a function without the `static` keyword, on the other hand, have no default initializer, and if they are not initialized in the source code will receive a value which is undefined by the C standard (which in practice is whatever value happens to be stored in the memory location where the variable is allocated when the function is called). Reading the contents of such a variable prior to the first write is another difficult to diagnose error which afflicts programs written in C.

Similarly, memory which is dynamically-allocated by `malloc()` is uninitialized, and it is the responsibility of the caller to initialize the memory before reading from it.

Reactis for C keeps tracks of which memory locations have been initialized and will raise an error when an attempt to read from uninitialized memory occurs.

14.3.2 Invalid Pointer Creation

Reactis for C 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:
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 handling of out-of-bounds pointers should be set to ignore or warning.

14.4 Other Runtime Errors

In addition to memory errors and overflows, Reactis for C 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.
Chapter 15

The Reactis for C API

The Reactis for C 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 http://www.reactive-systems.com/api.msp.

15.1 Using the Reactis for C 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\reactis4c 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 V2019\lib\api\MATLAB\reactis4c

or (if running a 32-bit Reactis within a 64-bit version of Windows):

C:\Program Files (x86)\Reactis V2019\lib\api\MATLAB\reactis4c

After adding the folder, you can get information about the Reactis API functions via the regular MATLAB help functionality. For example, typing help reactis4c on the MATLAB command line will list information about all Reactis API functions. Detailed information for each function can be accessed by typing help function or doc function.

Some examples for using the API can be found in the following folder within the Reactis for C installation directory: lib\api\MATLAB\reactis4c\examples

15.2 Using the Reactis for C API from C Programs

15.2.1 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 `rcOpen` to receive an `RcHandle` value which all other API functions require as a parameter. In the following, the term `RcHandle` will refer to the handle returned by this call.

2. Call `rcTester` passing the `RcHandle`, a model file name and other parameters according to the documentation of `rcTester`. This will create a test suite and return an `RcTestSuite` value (an abstract data structure representing the generated test suite).

3. If the `rcTester` call in the previous step fails, call `rcGetLastError` 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 `rcGetCoverageCriteria` functions to retrieve coverage of the test suite that was just created.

5. Call `rcSimOpen` passing the `RcHandle`, and a model file name. This will return a `RcSim` value which serves as a handle to the newly created Simulator session.

6. Call `rcSimExportSuite` passing the `RcSim` handle returned by `rcSimOpen`, the `RcTestSuite` value returned by `rcTester` 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 `rcSimClose` passing the `RcSim` handle returned by `rcSimOpen`. `rcSimClose` will free all memory allocated by the Simulator functions.

8. Call `rcClose`, passing the `RcHandle` as an argument. This will free all memory allocated by the Reactis for C API.

---

### 15.3 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 `libreactis4c.lib` to the linker arguments, so that the linker will be able to find the functions in `libreactis4c.dll`. `libreactis4c.lib` will work with a wide range of C compilers, including Microsoft compilers and any compiler which accepts Microsoft .lib format, such as the Gnu C
• Compile your application. The way to do this is highly dependent on your compiler and application code. For example, the following command compiles apitest.c using GCC:

\[
gcc -o apitest.exe apitest.c libreactis4c.lib
\]

• Add the Reactis lib\api folder (e.g., C:\Program Files\Reactis for C V2019\lib\api) to your Windows search path, so that your application can find the libreactis.dll library. Alternatively, you can copy libreactis.dll into the folder where your application’s executable file is located.

### 15.4 Reactis API files

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:

- **reactis4c.h**  
  Header file containing declarations for the API functions.

- **libreactis4c.lib**  
  Library file suitable for Microsoft Visual C++ and GCC.

- **apitest4c.c**  
  Example program to illustrate how to use the API functions.

- **apitest4c.exe**  
  Compiled version of apitest4c.c.

- **libreactis.dll**  
  Dynamic library containing the API functions.
15.5 C Coverage Details

The/rcSimExportCCoverageDetails API call produces a primary output file containing target coverage information for C source code and an optional second file containing data for targets generated by macro expansion.

Each row of data in the primary output file contains the following columns:

1. The target id number (1 . . . n, where n is the number of targets).

2. A string describing the target kind:
   - "s"  Statement
   - "dt" Decision true
   - "df" Decision false
   - "ct" Condition true
   - "cf" Condition false
   - "m"  MC/DC

3. For condition targets, this field contains the id number of the decision target which contains the condition target. For MC/DC targets, this field contains the id number of the corresponding condition-true target. For decision and statement targets, this field will be zero.

4. The coverage status of the target (1 if covered, 0 if uncovered).

5. The test number in which the target was covered (-1 if uncovered or unknown).

6. The step number in which the target was covered (-1 if uncovered or unknown).

7. The path to the program component containing the target. This is useful when the same source file is used in more than one S-function.

8. The pathname of the C source file to which the target belongs.

9. The starting line number of the target.

10. The starting column number of the target.

11. The ending line number of the target.

12. The ending column of the target.

13. If the target is not defined within a macro expansion, this field will be zero. If the field is positive, it will contain an id number which can be used to find the macro expansion data in the macro data file.

14. The starting line within the macro expansion of the target.

15. The starting column within the macro expansion of the target.
15.5. **C COVERAGE DETAILS**

16. The ending line within the macro expansion of the target.
17. The ending column within the macro expansion of the target.

**Notes:**

- All strings are encoded as C literals.
- All numbers are encoded as decimal integers.
- Target ids, macro ids, lines and columns are numbered starting with 1 (not zero).
- For targets within macro expansions, the starting and ending source file locations will be the points where the macro expansion started and ended.
- If the target is not defined within a macro expansion, then columns 13-17 will contain zeros.
- If no macro data file is generated, then columns 13-17 will be omitted (the CSV data will contain only 12 columns).

### 15.5.1 Macro file format

Each row of CSV data in the secondary macro information file contains the following columns:

1. The id number of the macro expansion (1 \( \ldots n \), where \( n \) is the number of macro expansions).
2. The name of the macro which was expanded.
3. The file where the macro was defined.
4. The line number where the macro was defined.
5. The column number where the macro was defined (should always be 1).
6. The line number where the macro definition ended.
7. The column number where the macro definition ended.
8. The file in which the macro invocation occurred.
9. The line number at which the macro invocation started.
10. The column number at which the macro invocation started.
11. The line number at which the macro invocation ended.
12. The column number at which the macro invocation ended.
13. The text of the macro expansion, encoded as a C string literal.
Notes:

- All strings are encoded as C literals.
- All numbers are encoded in decimal.
- Macro ids, lines and columns are numbered starting with 1.
Appendix A

Frequently Asked Questions

Please see http://www.reactive-systems.com/faq.msp to view the Reactis frequently asked questions.
Appendix B

Revision History

Different versions of Reactis for C are labeled as shown in Figure B.1 and described below.

Figure B.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 for C is labeled by a “V” followed by the four-digit year (e.g., V2019). Each label for an intra-year release includes a suffix consisting of a decimal point followed by a *major release number*; for example V2019.1, V2019.2, etc. will label the releases during 2019 that follow V2019.

**Beta Releases.** RSI often makes beta releases available to customers interested in evaluating the newest features of Reactis for C. Beta releases do not undergo as much testing as major releases do. By convention, beta releases have odd numbered major release numbers. For example, V2019.1, V2019.3, ... 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:

- V2019.2.1 denotes the first patch release for V2019.2
- V2019.0.2 denotes the second patch release for V2019
B.1 V2019 (14 June 2019)

The V2019 release of Reactis for C includes all patches since the V2018 release, plus the following new features.

B.1.1 Instrumentation Code

Instrumentation code allows Validator objectives (assertions and user-defined targets) to be added to code under test without modifying any of the source files. The instrumentation code is stored separately and integrated with the code under test when the code is compiled.

B.1.2 Revamped User-Guided Simulation

The User-Guided Simulation mode of Reactis Simulator has been revamped. The new Next Input Values dialog now lets you either enter a concrete value (as it previously did) or an expression to be evaluated to compute the next value for an input. This offers an easy way to specify a ramp or sine wave for input signal. Additionally, the stepping buttons are now available in the Next Input Values dialog as well as the main Simulator window. Also, you can now control Configuration Variables (at the start of a test) and observe outputs. Scopes can easily be opened for a subset of signals from the Next Input Values dialog.

B.1.3 Watch Individual Structure/Array Elements

Individual structure members and array elements can now be watched. The value of the member is displayed on a separate line in the watched variables panel and is highlighted with a yellow background when it changes. If desired, you can also select the entire structure or array and watch it as a single aggregate value.

B.1.4 Open Scopes from Test Suite Browser

For easier visualization of test data, you can now open scopes directly within the Test Suite Browser for any signals (inputs, outputs, test points) contained in a test suite.

B.2 Previous Major Release Dates

Major releases of Reactis for C prior to V2019 have occurred on the following dates:

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>V2018</td>
<td>9 February 2018</td>
</tr>
<tr>
<td>V2016</td>
<td>21 October 2016</td>
</tr>
<tr>
<td>V2015</td>
<td>18 November 2015</td>
</tr>
<tr>
<td>V2014</td>
<td>10 April 2014</td>
</tr>
<tr>
<td>V2011</td>
<td>21 March 2011</td>
</tr>
</tbody>
</table>
Index

absolute tolerance, 61
assertion, 117, 118
boundary value, 78
breakpoint, 27
browsing
  coverage report, 23, 89, 133
  test suite, 27, 37, 88
closing
  scope, 87
condition, 76
configuration variable, 36, 62
  updating, 88
coverage
  excluding targets, 37
  highlighting, 79, 89
  selecting metrics, 89
  specifying metrics, 36
  summary, 89
  tracking, 20
  viewing details, 21, 89
coverage metric, 75
  assertion, 117
  boundary value, 78
  condition, 76
  decision, 76
  MC/DC, 22, 76
  MCC, 22, 77, 105
  UDT, 117
  user-defined target, 117
coverage report
  browsing, 23, 89, 133
  exporting, 135
decision, 76
delta types, 66
difference scope, 86, 87, 99, 101
distribution scope, 97, 126
division by zero, 149
division by zero error
  division by zero, 149
  floating-point, 147
  integer overflow, 145
  invalid shift, 149
  memory, 147
  spatial memory, 147
  specifying severity, 36
  temporal memory, 147
  uninitialized memory, 148
exporting
  coverage report, 135
  test suite, 88, 107, 108
FAQ, 157
fast simulation, 23
feedback
  report to RSI, 2
file
  .rsh, 12, 13, 53, 54, 63, 93
  .rsp, 87, 99
  .rst, 23, 85, 88, 97, 114
  .rtp, 35, 38, 114
floating-point error, 147
Frequently Asked Questions, 157
function
  entry, 10, 13, 14, 19, 27, 53, 55, 111
  initialization, 10, 14, 53, 55, 111
global settings, 36, 38
harness library
  default settings, 36, 43
help facility, 38
hierarchy panel, 85
history, 159

importing
test suite, 88, 108
initialization function, 10, 14, 53, 55, 111
input
  constraining, 36, 59
  from test suite, 23, 97
  from user, 25
  selecting, 85
input mode
  user, 95
input values
  from user, 95
  probabilities, 66
inputs
  plotting, 92
install
  silent, 6
installation, 3
instrumentation code, 121
  adding, 17, 122
  advantages, 121
  copying, 123
  cutting, 123
  disadvantages, 121
  editing, 37, 122
  identifying, 121
  moving, 124
  pasting, 123
  removing, 124
  verifying location, 123
integer overflow, 145
invalid shift, 149

launching Tester, 37, 88, 111
loading
  profile, 87
LUC, 45
MAC address, 42
main panel, 85
max tolerance, 61
MC/DC, 22, 76
MCC, 22, 77, 105
memory error, 147
min tolerance, 61

opening
  scope, 25
output
  comparing simulation vs. test, 89
  tolerance, 97
  updating, 88, 109
  viewing differences, 86, 87, 99, 101
output tolerances, 101
outputs
  plotting, 92
  tolerances, 101
panel
  hierarchy, 85
  main, 85
probabilities of input values, 66
profile
  loading, 87
  saving, 87, 99

Reactis
determining version, 38
Reactis for C
  installing, 3
relative tolerance, 61
report
test execution, 87, 90
resolution of a subrange, 15
resolution types, 67
revision history, 159
.rsh file, 12, 13, 53, 54, 63, 93
.rsp file, 87, 99
.rst file, 23, 85, 88, 97, 114
.rtp file, 35, 38, 114

saving
  profile, 87, 99
  test suite, 85, 88
scope
  closing, 87
  difference, 86, 87, 99, 101
distribution, 97, 126
  opening, 25
scopes
  difference, 101

searching
  test suite, 126

selecting
  input source, 85
  test suite, 85

setting
  animation delay, 88
  coverage metrics, 36
  decision metric, 76
  error checking, 36
  globals, 36, 38
  harness library defaults, 36, 43

signal group, 100

silent install, 6

simulation
  fast, 23
  user guided, 95
  user-guided, 25

simulation probabilities, 66

Simulator, 19, 83
  disabling, 33, 87
  enabling, 33, 87
  toolbar, 83

spatial memory error, 147

support
  report to RSI, 2

technical support
  report to RSI, 2

temporal memory error, 147

test execution report, 87, 90

test points
  plotting, 92

test suite
  adding a test, 27, 89
  browsing, 27, 37, 88
  creating empty, 85
  defragmenting, 88
  executing in Simulator, 23, 97
  exporting, 88, 107
    to CSV, 107
    to Text, 108
  filtering, 126

importing, 88
importing from CSV, 108
removing test, 89
saving, 85, 88
searching, 126
selecting, 85
updating outputs, 88, 109

Tester, 111
  display help for, 114
  launching, 37, 88, 111

parameters
  abstraction, 113
  halt condition, 111
  input change limit, 113
  loading from file, 114
  number of cores, 113
  number of random tests, 112
  number of targeted steps, 113
  output file, 114
  preload list, 111
  random seed, 113
  step limit, 112
  steps per random test, 113
  time limit, 112
  progress dialog, 114

tolerance
  absolute, 61
  editing, 62
  max, 61
  methods, 61
  min, 61
  output, 97
  relative, 61

tolerances
  output, 101

tracking coverage, 20

TUC, 67

type under construction, 67

UDT, 117

uninitialized memory error, 148

updating
  test suite outputs, 88, 109
user input mode, 95
user-defined target, 117, 118
user-guided simulation, 25

Validator, 117
Validator objective, 117, 118
  assertion, 118
  use case, 118
  user-defined target, 118
variable
  hovering on, 24, 85
  viewing distribution of values, 97
  viewing with scope, 24
  watching, 24, 87, 99
version, 159
  labeling, 159
  of Reactis, 38

watched variable, 24, 87, 99