Verification and Validation of Simulink Generated Source Code with VectorCAST

Introduction
The usage of model driven development is becoming more and more popular. There are many leading tools in this market space all catering for different types of application development. The MathWorks® Simulink® product is one such product. Simulink is an environment for multi-domain simulation and Model-Based Design for dynamic and embedded systems. It provides an interactive graphical environment and a customizable set of block libraries that let you design, simulate, implement, and test a variety of time-varying systems, including communications, controls, signal processing, video processing, and image processing. The Simulink product also allows the user to auto-generate the source code (typically in programming language ‘C’) from these defined models.

Simulink has very strong capability in proving model correctness through model simulation, it can also plugin auto generated source code components into the model simulation, through a process called Software In the Loop (SIL) where the code is executed natively, or progress a step further and allow this code to be executed as part of the model on the actual embedded target device through a process known as Processor In the Loop (PIL).

Where this auto generated code is going to be used in a safety critical system such as Automotive (ISO 26262), Avionics (DO-178B), Industrial Automation (IEC 61508), Medical Device (FDA MED or IEC 62304) or Railway (EN 50128), it must also be proven that every line, condition and sub-condition of code have been executed. This is normally done through Code Coverage.

The VectorCAST family of products automates testing activities across the software development lifecycle. VectorCAST/Cover specializes in enabling developers to measure code coverage from an application that has been executed either natively or on the target embedded device (system Testing).

VectorCAST/C++ automates the process for construction of test frameworks around modules of code in isolation of the rest of the system (Unit Testing) to develop scenarios to prove the correctness of these modules. The process of Unit Testing would also measure code coverage, with the key goal to combine coverage from System and Unit Testing to prove the code coverage requirement.
In this whitepaper we discuss how the Simulink product and VectorCAST products can be used together to produce the required verification and validation artifacts for developing a safety critical application.

**Software-in-the-loop Testing**
The Software-in-the-loop simulation captures the behavior of the generated C code in the simulator environment of the target controller. The simulated results, when compared with the simulated Model-in-the-Loop simulation results, should be identical. If found otherwise the results can be used to evaluate the cause of deviation.

As discussed previously, when developing software for a safety critical standard, one of the artifacts that is required is structural code coverage. The VectorCAST/Cover integration with Simulink allows the user to measure code coverage from the source code that has been executed when conducting Software-in-the-loop testing. Figure 1 demonstrates this workflow. VectorCAST/Cover allows you to gauge the effectiveness of system testing by determining which areas of an application have been exercised (or covered). You can perform coverage analysis on a portion of an application or on the entire application.

![Figure 1 - Software-in-the-loop Workflow](image)
Here are the steps to integrate Simulink® with VectorCAST/Cover:

The first thing that needs to be done is configure compiler integration with Simulink for SIL (Software-in-the-loop) execution. Inside Matlab run command “mex('-setup’)” in command window and follow prompts and select the correct compiler. You should only need to do this once for your Matlab installation.

Figure 2 - Configure compiler for Matlab
Next we need to create a model. This can be done by using the Menu in Matlab File→New→Model. To use some common library components go to View→Library Browser. You need inputs and outputs. These are normally in “commonly used blocks”. Figure 3 shows the model we will be using for this example. In this example we have created a model that takes 2 inputs (In1 and In2) which are summed together, they are then sent through a gain (10 times multiplier) after which the result of this is multiplied by a constant to produce the final result (Out1).
Now let’s configure our model for use. We need to launch the Configuration Parameters dialog. This can be done from the Model view menu → Simulation → Configuration Parameters. First select in the left hand column ‘Solver’ and set Type to Fixed Step as shown in Figure 4.

Figure 4 - Configure Solver
We then need to configure the Hardware Implementation. This can be done by selecting ‘Hardware Implementation’ in the left column and the setting ‘Device Vendor = Intel’ and ‘Device Type = x86/Pentium’. This is shown in Figure 5.

![Figure 5 - Configure Hardware Implementation](image)

Finally we need to configure ‘Code Generation’. This is done by selecting ‘Code Generation’ in the left column and then setting ‘System Target file = ert.tlc’ and in the sub-node Report check boxes “create code generation report” and “model-to-code” in the right hand column.

Now run the model and confirm it runs without errors. Once you have done this, we now need to copy across some utility files that enable the integration between Simulink and VectorCAST/Cover. These files can be found in your VectorCAST installation directory in the subdirectory DATA\simulink. If you are unable to find these files with your version of VectorCAST, please contact your local sales office for assistance in finding these files.

The following four files should be there:

1. CodeCoverage.m
2. CodeCoverageConfigGui.fig
3. CodeCoverageConfigGui.m
4. ert_make_rtw_hook.m
Copy these files to the same directory as where the model is located. Now save the model again. In the Matlab Simulink command window type 'CodeCoverage'. This will launch a dialog where you can set coverage level. This is shown in Figure 6.

![Figure 6 - Code Coverage Dialog](image)

Now build the model and code generation. Select the model dialog and press “Ctrl-D”; this updates the model with all changes we have just done.
Now select the model dialog and press “Ctrl-B”; this builds the auto generated code and executable. This can take some time to start then you will see output on the command window. The critical selection of the output that shows you the VectorCAST/Cover instrumented source files have been used are shown in Figure 7.

```plaintext
### TLC code generation complete.
### Creating HTML report file model_aggregator_codegen_rpt.html
### Initializing VectorCAST code coverage
### Creating VectorCAST code coverage environment 'vcast'
at
C:\servers\dataserver\marketing\Partners\Mathworks\demos\simulink_full_workflow\model\model_aggregator_ert_rtw\vcast.vcp
### Instrumenting model source for VectorCAST code coverage
### Inserting VectorCAST IO code (and fclose) into
model_aggregator.c...
```

Figure 7 - Matlab Output

Now we need to tell the model to run in Software-in-the-loop mode. To do this in the Model dialog, go to Simulation→SIL (this tells it to use generated code as opposed to model). You can then click the run button again, and now you should see more output in the command window.

The resulting execution should produce a 'TESTINSS.DAT' file in the same directory as where the model is located. This file contains the raw coverage data for code coverage as a result of the execution.

To view the code coverage, navigate to the subdirectory `<model_name>_ert_rtw`. Inside here you will see the VectorCAST/Cover project. Launch VectorCAST and open this project. From the VectorCAST/Cover toolbar select the Results button and point it the ‘TESTINSS.DAT’ file.
When VectorCAST/Cover imports this file it will ask you for a label for it, any label that helps you identify the coverage data is suitable. In our example we will call it 'Simple run'.

The VectorCAST/Cover tool can now be used as normal. The resulting code coverage can be seen in Figure 8.

Figure 8 - VectorCAST/Cover - Code Coverage from SIL Execution

Usually it is not possible to test an entire application during system testing. For this reason, a combination of unit, integration, and system test is necessary to achieve 100% coverage.
Ensuring 100% Coverage

Figure 9 and Figure 10 illustrate two methods of using VectorCAST/Cover and VectorCAST/C++ (VectorCAST/Ada) to ensure 100% code coverage.

Method 1 (see Figure 9) typically applies to the traditional approach to testing, in which the testing progression is from individual components to aggregates, to the full system.

Method 2 (see Figure 10) typically applies to requirements-based testing, in which high-level system requirements are typically tested first, followed by low-level unit or integration tests. Achieving 100% coverage is the goal of both methods.

Under the first method, you would use VectorCAST/C++ to conduct unit and integration testing on components and aggregates. In conjunction with this testing, you would use the built-in coverage utility to determine and report on source coverage. At functional/system test, you would use the VectorCAST/Cover tool; along with coverage data from VectorCAST/C++ to ensure 100% coverage.
Under the second method, in conjunction with performing system testing, you would use VectorCAST/Cover to identify any ‘holes’ in source coverage. You would then use VectorCAST/C++ to augment the coverage by running test cases on the files of source code containing ‘holes’. In addition, you would use VectorCAST/C++ coverage-analysis utility; along with coverage data from VectorCAST/Cover to ensure 100% coverage.

**Unit testing an auto generated source code module from Simulink with VectorCAST/C++**

When we look at unit testing VectorCAST/C++ will work with Simulink’s auto-generated code. The code generated by Simulink can either be created from The MathWorks Real Time Workshop Embedded Coder, or a 3rd party plugin coder.
When we talk about Unit testing our goal is to test either a single source file or a selection of source files in isolation of the rest of the system. In order to test this way, the source files under test need to be called from some framework code (Test Driver) and the dependencies of the source files under test will need to be replaced with some framework code so that the tester can either verify the values passed into the dependency or simulate certain conditions by returning values from the calls to the dependencies.

The replacement code for the real dependencies of the source files under test, are called stubs. The following key terms are used in the remainder of this paper:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Under Test</td>
<td>The UUT is usually a .c or .cpp file, or an Ada package. The individual subprograms (functions, methods, or procedures) can be invoked with either parameter data or global data to execute specific source lines of code within a subprogram.</td>
</tr>
<tr>
<td>(UUT)</td>
<td></td>
</tr>
<tr>
<td>Test Driver</td>
<td>A test driver is a main program created specifically to invoke the various subprograms in the UUT.</td>
</tr>
<tr>
<td>Dependent</td>
<td>A dependent is a module whose subprograms would be called from the UUT.</td>
</tr>
<tr>
<td>Stub</td>
<td>A stub takes the place of a real dependent. A stub is generally used when you want to control the values that a dependent subprogram would return to a calling subprogram. This allows tests to be developed with specific data to drive the execution down a specific path in the code.</td>
</tr>
<tr>
<td>Test Environment</td>
<td>A test environment (also test harness, or test bed) is an executable program built specifically to perform unit or integration testing on one or more units (files) of source code. A test environment includes a test driver, one or more UTTs, any stubs, and any real dependents. A test environment is generally able to:</td>
</tr>
<tr>
<td></td>
<td>• Invocate all visible functions</td>
</tr>
<tr>
<td></td>
<td>• Manipulate UUT inputs</td>
</tr>
<tr>
<td></td>
<td>• Capture UUT outputs</td>
</tr>
<tr>
<td></td>
<td>• Generate test reports</td>
</tr>
<tr>
<td></td>
<td>• Capture stub input parameters</td>
</tr>
<tr>
<td></td>
<td>• Manipulate stub output data</td>
</tr>
<tr>
<td></td>
<td>• Test exceptions</td>
</tr>
<tr>
<td></td>
<td>• Compare regression data</td>
</tr>
</tbody>
</table>
VectorCAST/C++ automates the key activities associated with unit and integration testing. Normally these activities are time-consuming, expensive, and for these reasons often short-changed.

As illustrated in Figure 12, the executable harness generated by VectorCAST/C++ consists of a test driver, the source file(s) under test, any user-designated stubs for dependent functions, and all un-stubbed dependents. The test harness is data-driven, meaning test data is read by the harness during execution. This approach precludes any need of having to compile and link a new executable harness each time the same tests are performed with different test cases.

Figure 12 - Automated generation of executable test harness

The following are basic steps for importing a Simulink model into VectorCAST.

1. Build a Simulink Model
2. Auto-generate code using Real Time Embedded Workshop
3. Create a VectorCAST Test Environment for the generated code
4. Export the Test Case Data from Simulink in a CSV Format
5. Create a mapping in VectorCAST of the CSV columns onto parameters or global data objects
6. Create test cases from the CSV mapping, each row of data will create a separate VectorCAST test case
7. Run the tests
8. Examine the test results and the code coverage in VectorCAST
Let’s return to the Simulink Model we created earlier in this paper. This model can be seen in Figure 3.

As a first step, what we would like to do is configure Simulink to generate the appropriate source code for us. To do this we configure Simulink to generate the code for us using Real-Time Workshop Embedded Coder.

This is done by going into the Model Explorer (Menu→View→Model Explorer or Ctrl-H). We want to configure the code generator, this can be done by selecting ‘Configuration’ in the first column, and ‘Code Generation’. In the right most column a set of tabs are now present. These can be used to configure our code generation.

In the first tab ‘General’, we want to set ‘System target file’ to use ert.tlc (Embedded Coder) as shown in Figure 13.

![Figure 13 - Selecting System target file to be ert.tlc](image-url)
Now, as we are planning to unit test our code, we need to ask the code generator to create the code just for our model, and not any of the framework code (e.g.; main()). This can be done by selecting the tab ‘Templates’ in the 3rd column and unselecting the check box ‘Generate an example main program’ as shown in Figure 14.

We now want to configure the solver to be ‘Fixed-Step’, as our model is. To do this, in the Model Explorer view, in the second column now select ‘Solver’, and in the third column under ‘Solver Options’ change ‘Type’ to ‘Fixed-Step’, and ‘Solver’ to ‘discrete (no continuous states)’. This is shown in Figure 15.
We now need to configure the code generation to create ‘C’ functions that take arguments, rather than using global variables. The tab for this can be found in the third column under the tab ‘Interface’, when ‘Code Generation’ has been selected in the second column inside the Model Explorer. This is shown in Figure 16. It is not essential that the functions take arguments, but from a testing point of view it simplifies things as shown later.

**Figure 16 - Generate ‘C’ style functions**

We are now ready to request Real-Time workshop to generate the code for us. This can be done by selecting the ‘General’ tab from the third column when the ‘Code Generation’ item is selected in the second column inside the Model Explorer. At the bottom of the ‘General’ tab is a button to generate the code.
Click this button as shown in Figure 17, and you should then see a report similar to that shown in Figure 18 once the code has been generated. It is important to ensure you set your working directory correctly, as Real Time Workshop will create the code in this directory.
Going back to the Model Explorer, by selecting the node ‘Code for <model_name> -> This Model’ in the first column, and then selecting '<model_name>.c' in the second column, in the third column you will now be able to see the auto generated code. This is shown in Figure 19.
At this point we can now use VectorCAST/C++ to build a Unit Test Environment around our code so that we can test it. To build an environment with VectorCAST on the auto generated code we follow the following steps. From the VectorCAST main menu, we select File→New→C/C++ Environment as shown in Figure 20.

**Figure 20 - Create a new C/C++ environment**

We now need to select the compiler we would like to work with. This can be any native or embedded compiler. The full list of supported compilers is available on the Vector Software website.

For this example we are using Visual Studio 2008 as shown in Figure 21.

**Figure 21 - VectorCAST compiler selection**
We then need to give the environment a name, this can be any name, but for the sake of simplicity, we will call it ‘SIMULINK_EXAMPLE’. This is shown in Figure 22.

![Figure 22 - Environment Name](image)

We now need to tell VectorCAST the testing mode we would like to use. For this example we will select ‘Traditional Unit Testing’. This is shown in Figure 23. However, the reader may want to investigate ‘Test Driven Development’, as this would allow test case definitions in parallel to model development.

![Figure 23 - Testing Mode](image)
We now need to select the type of coverage we would like, and if we want to do whitebox or black box testing. For this example Whitebox is more interesting, so we select this. For Code Coverage we select ‘Level A’ as this provides us ‘Statement’, ‘Branch’ & ‘MCDC’ Coverage. This is shown in Figure 24.

![Figure 24 – Build Options](image)

In the next step we want to tell VectorCAST where the source code for our auto generated code exists. This is shown in Figure 25.

![Figure 25 – Locate Source Files](image)
We are now ready to tell VectorCAST which units we want to test, and what we want to do with the dependencies. In this example we want to test the file called ‘<model_name>.c’, and we choose ‘custom’ or ‘all’ for the stubbing preferences for the dependencies. In this example we have chosen ‘custom’ and told VectorCAST to stub the dependency by implementation. This is shown in Figure 26.

![Figure 26 – Configure Environment](image)

We can now tell VectorCAST to ‘Build’ or ‘Update’ the environment for us.

Once VectorCAST has finished building the environment for us, we can see on the left hand side of the VectorCAST GUI a tree structure which shows us a list of the functions that are available to test. If we right-click on the function ‘<model_name>_custom’, or in this case ‘model_aggregator_custom’ and say insert test case, we are presented with a table view in the right hand side of the VectorCAST GUI. If we take a closer look we can see the parameters for the function, and switching back to Simulink, we can see they match the same parameters of the auto generated code. This can be seen in Figure 27.
At this point we are now ready to start defining test cases inside the VectorCAST GUI based on the requirements of the system. As an extension, some users may wish to import datasets that were used inside Simulink to verify the model, to independently verify the source code inside VectorCAST.

The following steps outline the process for extracting data from Simulink and importing it into VectorCAST/C++ as test case data. It is particularly valuable where a large data set is being used such as a Waveform or data stream, as it allows a batch export/import of the data set values between both tools.
To begin exporting dataset values out of Simulink, we need to configure Simulink to log the signal outputs. This can be done by first of all replacing the inputs with the required datasets, and then right clicking on each input node and selecting ‘Signal Properties...’. This is shown in Figure 28.

For each signal we need to make sure the ‘log signal data’ option is selected, and give the signal a name. This is shown in Figure 29.
We now need to set up the time series ‘object’ that will hold all of the logged data. If we go back to the Model Explorer view, and in the second column select ‘Data Import/Export’, in the 3rd column we can see some options to set the output buffer, the number of data points to capture and the object to store it in.

![Figure 30 - Data Import / Export](image)

We are now ready to run our Simulink model. This can be done by selecting the play button in the model viewer. We have also attached a scope to our output so that we can visualize the result. Once the model has finished executing, we should see some new objects created in Matlab. This is shown in Figure 31. In this case, we asked Simulink to store the values in the object ‘logsout’.

![Figure 31 - New Objects Created](image)
We are now ready to dump the data into an object that can be exported to CSV data. ‘csvwrite’ is a Matlab native utility which can assist us with this. Figure 32 shows us this.

Figure 32 - Write a CSV file
We are now ready to switch back to VectorCAST, where we will import the CSV data that has been exported from Matlab/Simulink. Inside the VectorCAST GUI, if we right click on the function `<model_name>_custom` and select ‘Generate CSV Map’, this will allow us to now import the CSV file we just created, and define the mapping between the columns in the CSV file and the parameters for our function under test. This is shown in Figure 33.

![Figure 33 - Generate CSV Map](image)

On the right hand side of the VectorCAST GUI we can now see a dialog window that allows us to specify the csv file we wish to import. Once we have specified the file, we can click the ‘Load’ button. This is shown in Figure 34.

![Figure 34 - Import CSV data configuration](image)
We now need to setup the mapping, and then tell VectorCAST to create test cases for the whole dataset. Once we have completed the mapping, we can right click on the CSV test case and ask VectorCAST to generate all of the csv testcases for us automatically.

The benefit of this process is that if the data we use to verify our model changes, we can just ask VectorCAST to regenerate the test cases for us again.

As we are working with floating points and the tolerances for the model inside Simulink® may vary to the code being executed, we can request VectorCAST to setup a standard tolerance for all Floating Point comparisons.
This can be done by going to the VectorCAST menu → Tools → Options. This should launch a dialog window. In here select the ‘Execute’ tab and then modify the ‘Floating point tolerance’ field. We will set this to 1.000% for this example.

We can now right click on the function '<model_name>_custom' and ask VectorCAST to execute all of the test cases. Figure 37 shows us a sample test case execution report from executing the test cases.