`UnitTesting`: A Library for Modelica Unit Testing

Dr. Michael M. Tiller[1]
Emmeskay, Inc.
Plymouth, MI USA
mtiller@emmeskay.com

Burit Kittirungsi
Emmeskay, Inc.
Plymouth, MI USA
burit@emmeskay.com

Abstract

One of the challenges when attempting to deploy model based development processes in industry is the "distrust" that many engineers have of models. It is often the case that engineers favor experiments and data over models and simulation results. The typical reason given for this distrust is that engineers often feel they can "trust" information collected from actual hardware during an experiment more than they can trust simulation results.

There is some validity in this notion. In many ways, modeling is as much an "art" as it is a science. The process of developing models involves creativity and many subjective decisions. In order for modeling to be thought of as a "science", it needs to be viewed as more objective and analytical. The process of developing models must mature from an ad hoc process into a process that includes formal validation, verification and quality assurance processes just like any other "product".

This paper describes the `UnitTesting` library which is a commercial Modelica library offered by Emmeskay, Inc. The `UnitTesting` library supports automated testing of Modelica models to ensure, among other things, the ongoing validity and accuracy of the models.

Keywords: testing, quality, coverage

1 Background

Although not widely recognized or appreciated, model development closely parallels software development. This is particularly true for model development in Modelica because, in addition to the fundamental mathematical constructs required in a modeling language, Modelica has incorporated many software engineering principles into the language design (*e.g.* encapsulation, interface compatibility, *etc.*).

For this reason, many software development methodologies can be applied to model development as well. A recent trend in software development has been the adoption of testing methodologies to improve quality. These methodologies involve codifying requirements and automatically testing those requirements. One example of this is the JUnit library[1] developed by Gamma and Beck which can be used to automatically test Java classes. Most previous efforts at applying testing to model based control system development have focused on embedded code[2] and perhaps a few invariant physical or mathematical properties[3]. In this paper we propose a generic approach that handles testing for all types of Modelica models whether their behavior is continuous, discrete or a mixture of the two.

The purpose of unit testing is to identify problems at their source. This is done by using carefully designed test cases to immediately identify and isolate model errors. One convenient way of identifying errors as they are introduced into existing models is to compare results using the new model versions with baseline results from previous versions.

In order for this kind of testing to be effective, it needs to be automated and focused. For example, lacking the appropriate tools to automate the comparison process some developers may have to rely on visual inspection. In such cases, the testing process becomes tedious and error prone. Furthermore, test cases themselves must be carefully chosen so that failures are meaningful and help to isolate errors. For example, if only a few complex models are used as test cases, it will be difficult to trace the source of the problem. Without this degree of automation and focus, identifying errors is like trying to identify a poorly tuned instrument in a large orchestra. Assuming you are astute enough to notice the problem, simply detecting the problem is not sufficient to identify the source of the problem. If, on the other hand, you were to analyze each individual instrument (or even small clusters of them) you would be able to isolate

1 All correspondence should be directed to the first author.

the source of the problem much faster. The `UnitTesting` library provides not only a framework for creating test cases but also includes automation and reporting tools.

2 Use Case

In this paper, the use case we consider is that of a library developer who wishes to test a specific library (and only that library). Under these assumptions all models fall in to one of three categories.

The first category is *library components*. These are the components of the library being tested (*i.e.* the components that the user of the library would drag and drop into their models). The next category are *non-library components*. These are component models from libraries other than the library being tested[2]. For example, components from the Modelica standard library would generally be non-library components (unless, of course, you were a Modelica Standard Library developer testing the Modelica Standard Library). The last type of model is a *test case*. The test cases are models that instantiate library and non-library components and exist only to test library components (*i.e.* they are not considered library components).

Before proceeding, we need to describe a few more details about these test case models. The principle behind a test case is that it is a model that is instrumented to compare a simulated result with some *expected result*. How these expectations are represented will be discussed in greater detail later. The important point is that the test case model itself includes information about its expected result. When the test case is simulated it can report whether the expected result was achieved or not. In the next section, we will commonly refer to the case where a test case *fails*. This is what happens when the test case is simulated and fails to match *all* the expected results.

3 Metrics

When it comes to model quality and testing, we cannot convey the state of a model library as a single number. Instead, we depend on a variety of metrics to assess different desirable properties. In this section, we will introduce several different metrics and the issues that they address.

2 The same model could be a library component in one circumstance and a non-library component in other (depending on what library was being tested).

For all of the metrics presented in this section, it is possible to do at least some basic calculation of the metric using only the Modelica source code (*i.e.* without the need to run any simulations). However, in many cases the basic metrics and generated reports are enhanced (sometimes considerably) by access to simulation results. If available, the `UnitTesting` tools will use such results to conduct a more thorough analysis.

3.1 Component Coverage

Component coverage measures what percentage of the library being tested is actually "covered" by the test cases for that library. As a consequence of this analysis, library components that are never tested can be identified. The premise behind this metric is that if a component never appears in a test case it will be impossible to identify errors in that component.

The value of this metric is computed as follows. First, we consider each library component in turn and determine whether it appears in any of the test cases. We say that the library component appears in a test case if it is instantiated (in the Modelica sense) as part of the instantiation of that test case.

To evaluate the coverage metric for a library we simply divide the number of library components that appear in a test case by the total number of library components. A result of 100% indicates that the library has 100% component coverage. Said another way, every component in the library appears in at least one test case.

This metric is the most basic metric. Getting 100% component coverage is simply the first step in establishing a robust testing process. In principle, 100% component coverage implies that a mistake in a single component cannot "slip through" but will instead be detected via the failure of at least one test case. However, as we shall shortly see this is a very optimistic assumption.

3.2 State Coverage

Earlier, we introduced the notion of an expected result. When building a test case model, many variables are involved. It is generally not sufficient to validate a model based only on the value of one variable in that test case.

So the question then becomes "what variables should be checked?". State coverage is one way to address this issue. It is quite common in Modelica to have *alias variables*. These are variables that have a

unique name in the instance hierarchy but are mathematically identical to other variables. In fact, due to the semantics of connections the vast majority of variables in a large Modelica model are typically alias variables. Obviously, there is no sense in confirming the value of all these redundant variables. Instead, we want to focus on the unique signals. But even these are often computed from each other. For example, if we have the equations "x=y+5", we don't necessarily need to check both x and y.

To minimize the set of signals to check while maximizing the chance of catching an error in the model we should focus on the state variables. For our purposes, we define the state variables in the model as those variables that uniquely define the state of the model (and from which all unique continuous and discrete signals must therefore be calculated).

Precisely determining the state variables in a Modelica model can be quite tricky. In fact, state selection is typically part of the compilation process of a Modelica model and this process is non-trivial. Fortunately, for our purposes we can accept "false positive" results. So we employ a few basic rules that help us to identify all the *potential* states. It may turn out that these states are later eliminated via constraint equations (*i.e.* the states are not, in fact, independent) but that is not so important for our purposes and only results in a slight amount of redundancy in our checking.

We consider a variable a state if it is either assigned a value in a when clause (*i.e.* it is a discrete variable) or the variable appears within the der(...) operator. As mentioned previously, such variables are not necessarily actual states but this substantially narrows the set of variables to consider and ensures that we will at least catch all potential states. Said another way, to achieve state coverage it is sufficient to identify the potential state variables but not necessary to identify the actual state variables.

The state coverage of a test case is determined by taking the the number of states for which expected results are provided and dividing that by the total number of states in the model. Since the set of states can be automatically determined *a priori* it is possible to analyze a test case and automatically instrument it with all the necessary expected results in order to achieve 100% state coverage. Note that we take a very conservative position here that the expected results must refer directly to these states (and not to aliases) in order to be considered "covered"[3].

It should be noted that it might also be useful to apply "input coverage" and "output coverage" to a test case as well. That is to say that expected "results" are supplied for all top level input and output signals and all parameters in the model. The combination of input, output and state coverage helps ensure that any issues that arise during testing are truly due to behavioral changes in the library components and not due to changes in the values of test case parameters or simulation conditions. However, the `UnitTesting` library does not currently address input and output coverage.

3.3 Decision/Condition Coverage

Previously we mentioned the conditions under which 100% component coverage can be achieved. The problem with the component coverage metric is that it assumes that simply instantiating a component is sufficient to test all functions of that component. While this is true of very simple models, it is not true in general. A more complete metric is "condition/decision coverage" [4].

The goal of condition/decision coverage is to verify that logical aspects of the models are completely exercised. To achieve decision coverage we must verify that every decision being made in the code is "exercised" (*i.e.* for each decision both a "true" and "false" result is generated). Coming up with test cases that achieve complete decision coverage is challenging (and in some cases, simply not possible). It should be noted that the `UnitTesting` library only evaluates coverage *a posteriori* and does not provide functionality to identify how uncovered decisions can be covered.

Decisions can be compound expressions involving many individual boolean values. For example in the statement:

```
if a>10 and b<5 then
```

the *decision* is based on the value of the entire expression "`a>10 and b<5`" but this decision contains two individual *conditions*, e.g. "`a>10`" and "`b<5`". Even if both outcomes of the decision have been covered in a test case, this does not mean that every condition has been verified. For example, imagine `b` always had a value of 3. In that case all decisions would be covered as long as `a` had a value both higher and lower than 10. But this does not test whether the condition "b<5" was implemented correctly (*e.g.* "b<6" would work just as well).

3 An alternative approach would be to determine all alias variables of the potential states and see if they were being compared against expected results but we do not currently support that.

For this reason, condition coverage expands the notion of decision coverage to include all possible values of every condition. However, it is not necessary to exhaustively test all conditions. Instead, something called "modified condition/decision coverage" is sufficient but the definition of modified condition/decision coverage is beyond the scope of this paper.

In order to evaluate decision coverage it is first necessary to construct a decision graph. This shows all possible outcomes in a model. As an example, consider the very simple controller model shown in Figure 1. Essentially, this controller model provides a constant actuation "force" if the system it is controlling goes outside a specified set of bounds. It also includes the ability to "suppress" this output when needed.

```
model Controller
  import Modelica.Blocks.Interfaces.*;
  RealInput u;
  RealOutput y;
  BooleanInput suppress;
  parameter Real lowerLimit;
  parameter Real upperLimit;
  parameter Real level;
algorithm
  when u>=upperLimit then
    if not suppress then
      y := -level "Outcome 1";
    else
      y := 0 "Outcome 2";
    end if;
  end when;
  when u<=upperLimit then
    y := 0 "Outcome 3";
  end when;
  when u<=lowerLimit then
    if not suppress then
      y := level "Outcome 4";
    else
      y := 0 "Outcome 5";
    end if;
  end when;
  when u>=lowerLimit then
    y := 0 "Outcome 6";
  end when;
  when suppress then
    y := 0 "Outcome 7";
  end when;
end Controller;
```

Figure 1: A Simple Logical Controller Model

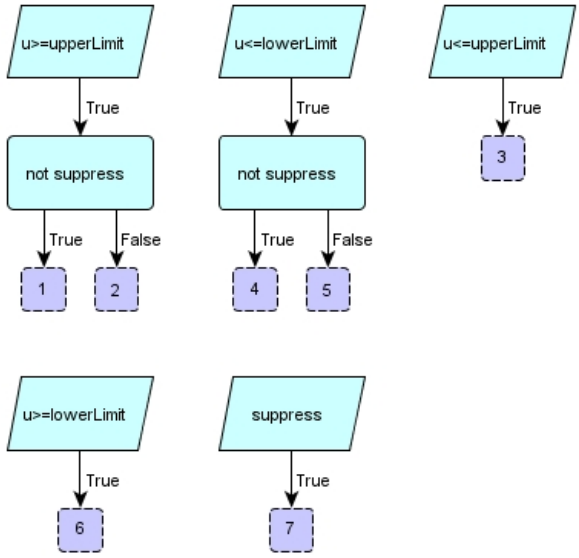

Figure 2: Controller Model Decision Tree

As previously mentioned, the `UnitTesting` library includes automation and reporting tools. As part of the standard reports, a decision tree is rendered for each model. Figure 2 shows the decision tree for the controller model shown in Figure 1. Each possible outcome is indicated by dark blue squares with dashed outlines. Decisions result from `if`, `when` and `while` statements as well as `if` expressions.

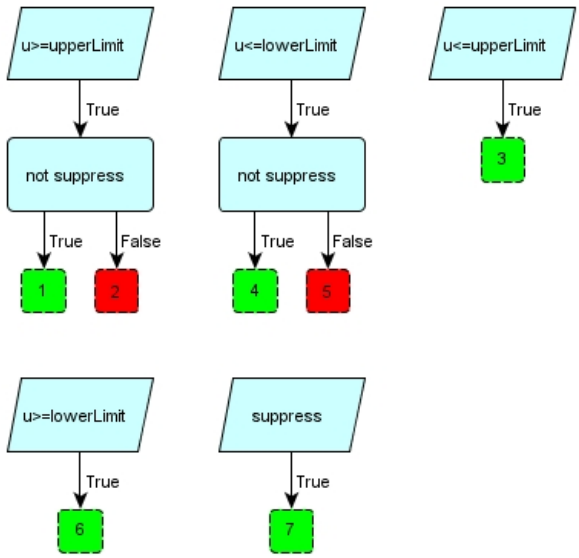

Figure 3: Decision Tree with Coverage Information

To visualization decision coverage it is necessary to use simulation results to assess whether each outcome has been reached. Given this information, we can color the figure, as shown in Figure 3, such that outcomes that have been achieved are green and outcomes that have not are red. To evaluate the deci-

sion coverage, we simply divide the number of green outcomes by the total number of possible outcomes. So in Figure 3, the decision coverage would be 71%.

Along with these decision figures, the `UnitTesting` reports include information about the individual conditional expressions associated with each decision. However, it does not currently visualize the coverage of these conditions (mostly for the sake of visual clarity).

3.4 Pinpoint Metric

The pinpoint metric is a very subtle concept. Having 100% component coverage implies that if a model suddenly stops behaving as expected, one or more test cases will fail. If each test case included only a single library component then it would be very easy to trace a particular failure to a particular model or even to a particular equation/statement.

However, it is not always practical to construct such precise test suites. For example, one library component may depend on the presence of another library component in order to function. It may not be possible to test them independently. The goal of the pinpoint metric is to determine how easy it will be to use test case failure information to identify potential sources for the failure. In other words, for a given set of test cases how easy will it be to identify the component (or set of components) that have errors based on which test cases passed and which failed.

To this end, we would like to compute the probability of an error in a specific component given the fact that exactly one test case has failed. Mathematically speaking, what we are trying to achieve can be expressed in terms of probabilities. For each library component, we would like to compute:

$$P(C_i|F_j \wedge \bar{F}_k \, \forall k \neq j)$$

where C_i represents an error in the i^{th} library component and F_j represents a failure in the j^{th} (and only the j^{th} test case). Given a means of computing such conditional probabilities, the pinpoint metric, π, is defined as:

$$\pi(C_i) = \max_j P(C_i|F_j \wedge \bar{F}_k \, \forall k \neq j)$$

The ideal value for the pinpoint metric is 1.0 because that indicates that a failure in one test case definitively points to a specific component as the source of the error. The equations used in computing the pinpoint metric are actually quite involved, especially for complex test suites, so we will not include them in the paper.

Once we have computed the pinpoint metric for each component, we can visualize them as follows. Consider the test suite shown in Table 1. An X in this figure indicates the presence of a particular component in a particular test case. For example, a failure of all test cases points to component C_2 as the likely source of the error since it is present in all test cases. For simple test suites, this kind of deduction is not difficult but for larger test cases it is nice to automate this analysis. Furthermore, the issue is complicated by the fact that a component might fail in one test case but not in others. For example, it is possible for an error in C_2 to cause test cases 1 and 3 to fail but not test case 2. Such a result might give the misleading impression that C_1 must be the source of the error. This is the main reasons that the derivation of the pinpoint metric is based on probabilities.

	C_1	**C_2**	**C_3**
T_1	X	X	
T_2		X	X
T_3	X	X	X

Table 1: Sample Test Suite

The main purpose of the pinpoint metric is to determine how well the test suite has been constructed such that component errors lead to outcomes that allow us to easily track down the source of the error. This is analogous to the concept of observability in plant models. The idea here is that given the outputs of the system (*i.e.* whether each test case passed or failed) we can gain some insight into the internal state of the system (*i.e.* whether a given component is working properly or not).

The `UnitTesting` library tools also have the ability to compute the conditional probability that a given component contains an error given not just a single test case failure (as previously discussed) but any arbitrary pattern of test case failures. With this information it is then possible to construct an accountability matrix like the one shown in Table 2.

	T_1	**T_2**	**T_3**
T_1	?	?	C_1
T_2	?	?	C_3
T_3	C_1	C_3	?

Table 2: Accountability Matrix

The accountability matrix allows us to assess, given a failure in one or two tests, which component is the most likely to have an error. Each row and column in the accountability matrix represents a test case that has failed. Each cell in the table represents the most likely component responsible (*i.e.* the one with the largest conditional probability). The accountability matrix in Table 2 was constructed based on the information in Table 1. A question mark in the table indicates cases where there is no clear choice for the responsible component.

	C_1	**C_2**	**C_3**
T_1	X		
T_2		X	
T_3	X	X	X

Table 3: Sample Test Suite

An ideal accountability matrix would show every component in at least one cell. As we can see from Table 2, failure in only two components can be reliably determined for the test suite in Table 1. On the other hand, if the test suite was constructed as shown in Table 3 we would get the accountability matrix shown in Table 4 where each component appears in at least one cell.

	T_1	**T_2**	**T_3**
T_1	C_1	?	C_1
T_2	?	C_2	C_2
T_3	C_1	C_2	C_3

Table 4: Accountability Matrix

The accountability matrix takes the same information as the pinpoint metric and renders it in a very practical form that allows the developer to use knowledge about test case failures to work backward to identify the most likely source of the error. As mentioned previously, the value of automation is in being able to construct such tables for complex test suites.

3.5 Conservation Checking

In some modeling applications, it is particularly important to make sure that all interactions between components are properly accounted for. Let us start with a very simple example. Consider the EMF model from the Modelica Standard Library shown in Figure 4. This model is a basic DC motor model that relates current and voltage to torque and speed.

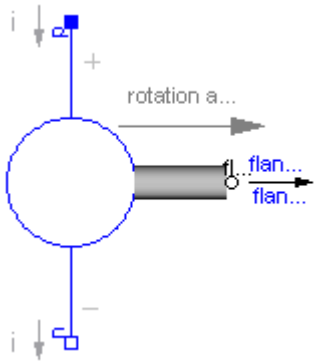

Figure 4: Standard Library EMF Model

To understand the conservation analysis, consider Figure 4 a "free body diagram" for the EMF model. Each connector represents a potential external influence on the EMF model. Generally speaking, the flow variables on the connector represent the flow of a conserved quantity into that component. In the EMF model, we see both electrical and mechanical interactions. If we run the `UnitTesting` conservation analysis on this component, the report includes a balance equation for all conserved quantities:

Quantity	**Modelica.Electrical.Analog.Basic.EMF**
Angular Momentum	**flange_b.tau = 0**
Charge	p.i+n.i = 0

Note that one particular conservation equation is rendered in a bold red font. This is to draw our attention to the fact that, based on structural analysis only, this component is unlikely to conserve angular momentum. This is because any exchange of momentum with another component would violate this equation. In some cases, such an equation is OK because the device includes some storage term. To avoid getting erroneous error messages, the `UnitTesting` tools will look for a variable inside the model that includes an annotation of the form:

```
annotation(msk(coverage(storageTerm(quantity="AngularMomentum"))));
```

Such an annotation indicates to the tools that this variable is a storage term and should be included in the balance equations. The ability to specify a storage term means that it should always be possible to avoid a "false positive" for conservation errors. Note that the EMF model does not include any storage term so the `UnitTesting` library is correct when it highlights this as an error in the model.

So far, we have only considered structural analysis but the conservation analysis also has the ability to use simulation results to determine if, in practice, the balance equation has been satisfied. In that case, it will highlight components that failed to properly balance these quantities while running the test cases.

The conservation analysis currently only addresses non-energy quantities. The reason energy is currently excluded is that computing the energy across the boundaries requires, in many cases, computing the derivatives of variables on the connectors. Unfortunately, the simulation may not need to compute (and therefore may not store) the derivatives required to perform this analysis (either because they are not necessary or because it computes the derivative of an alias variable instead). Ideally it will be possible in the future to conduct conservation analysis for energy as well. In that case, the complete conservation analysis process would be:

1. Check to make sure that the sum, over all connectors, of each conserved quantity besides energy (*e.g.* angular momentum, mass, current, *etc*) either equals zero or balances the internal storage of that quantity.

2. Check to make sure that the sum (computed either symbolically or numerically based on simulation results) of power flow over all connectors either equals zero or balances the internal storage of energy.

So far, we have focused on conservation within the components themselves. It may also seem necessary to analyze the connections between components to verify that they also conserve energy. However, for any physical connector were power flow can be computed using the variables on the connector (or their derivatives), energy conservation is automatic (*e.g.* all physical connectors in the Modelica Standard Library).

It may seem like such errors are easy to detect, but it is very easy to overlook important interactions. We already discussed the EMF model in the standard library. It may appear that the only possible error is one where only a single connector is included in the component. But another very common error is the one shown in Figure 5. Note that this model has two ports. So it does not suffer from the obvious error of including only a single port (and no storage term) that the `UnitTesting` library identified previously in the EMF model. The error in this model is more subtle than the one in the EMF model.

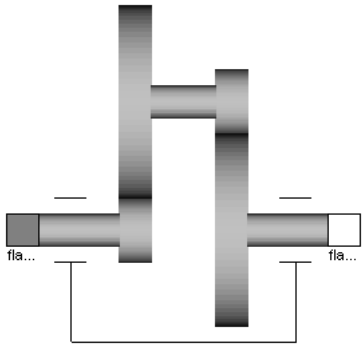

Figure 5: Non-Conserving Gear Model

To understand the nature of the error it is useful to recognize that *torque applied to a component represents a flow of angular momentum either into or out of the component*. This gear model, like the EMF model, does not include a storage term. Students are often taught to formulate the equations for a gear by first assuming a kinematic relationship between the element speeds, *i.e.* $\omega_a = R*\omega_b$ where R is the gear ratio. We then assume that the power into the component equals the power out, *i.e.* $\tau_a\omega_a+\tau_b\omega_b=0$. Using these two relationships it is trivial then to derive the relationship that $\tau_b=-R*\tau_a$.

Like a good magic trick, it appears that nothing suspicious has occurred here. Everything seems logical. But for this component the `UnitTesting` tools would generate the following report:

Quantity	ErroneousGear
Angular Momentum	flange_a.tau + flange_b.tau = 0

Based on structural information alone, the `UnitTesting` tools would not highlight this equation because it cannot determine *a priori* that this equation cannot be satisfied by the component[4]. But when it examines the simulation results it will see that something is not correct.

The source of the problem is that the constitutive equation, $\tau_b=-R*\tau_a$, cannot be correct. This is because the balance equation shown above would then be $\tau_a-R*\tau_a=0$. Assuming that the gear ratio is not 1, this equation implies that the amount of angular momentum going into the gear is not equal to the amount going out. Since this component has no storage term, angular momentum is not conserved by the component.

So where is the error? The error comes from not properly accounting for reaction torques. The model

4 Of course, with sophisticated symbolic processing this would be possible.

is implicitly grounded (*i.e.* it is assumed that the housing does not move). Ironically, this is exactly the same issue that the EMF model has. Torque cannot be generated out of nothing. We cannot forget Newton's Third Law, "For every action there is an **equal** and opposite reaction". For a gear to "multiply" torque something has to hold the gear housing (and absorb the reaction torque). This is exactly why the Modelica Standard Library gear looks like the one shown in Figure 6 (note the presence of the third port).

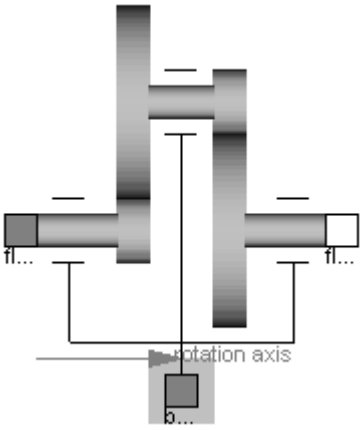

Figure 6: Conservative Gear

3.6 Style Guideline Compliance

A whole family of metrics can be constructed based on style guidelines. It is possible to formulate metrics around guideline adherence to help drive improvements in libraries. While this is beyond the scope of this paper (see [5] for a discussion of this topic), it should be noted that a comprehensive approach to library quality could benefit from attention to these kinds of details since they affect, in some cases significantly, user perception of library quality in subjective ways.

4 Using the Library

This section provides a basic overview of how the library is used in practice.

4.1 Making a Test

The first step in testing is to create a set of test cases. It helps to think of each test case as an experiment to test individual components or small groups of components. The fewer components in a test case, the better the pinpoint metric will be for the library components.

After an appropriate system model has been created, it can be turned into a test case by simply inheriting from the `UnitTesting.TestCase` model. This identifies the model as a test case. The TestCase model requires two parameters to be specified. The first is the `name` parameter which defines a unique name for identifying the test case[5]. The second parameter, `resultsDir`, indicates what directory any test result information should be stored in[6]. Typically, this results in a model that looks something like:

```
model TestComponentA
  import UnitTesting.TestCase;
  extends TestCase(name="TestCompo-
nentA", resultsDir=testResultsDir);
   ...
end TestComponentA;
```

In this case, `testResultsDir` is presumably some package level constant to define where all test results should be placed.

4.2 Adding Results

For the test case to be useful, it should include expected results to validate the model against. Although all kinds of expected results are possible, the most common type of result is one that compares the current simulation results to a previous baseline result. This kind of result is represented by the `ScalarResult` model. A typical declaration of the `ScalarResult` model might look like:

```
ScalarResult result1(name="iner-
tia.phi", y=inertia.phi);
```

where the `name` parameter should be given a unique value (at least within this test) to distinguish it from other results and the equation for `y` should set it equal to the variable being validated.

As mentioned previously, better state coverage will result in better error detection rates. However, instrumenting the model to check all possible states is tedious because it involves typing in many `ScalarResult` declarations. For this reason, the `UnitTesting` reports include a sample `ScalarResult` declaration (like the one above) for each state in the test case. Copying these declarations

5 If, at some point, the Modelica language is expanded to provide a function that returns the fully qualified name for any class then this parameter would no longer be necessary.

6 Again, better introspection support in the language could eliminate this parameter as well.

from the report into the test case ensures 100% state coverage.

4.3 Generating Expected Results

As mentioned previously, it is quite common to base the expected results on previously determined simulation results. Such results provide a baseline on which to assess the model over time. It would be quite tedious to manually specify all expected results (*e.g.* at time=4.32 the value of the signal should be 0.8203, and so on). For this reason, the `UnitTesting` library provides a means to extract expected values from existing simulation results.

For example, the previously mentioned `TestCase` model includes a special `generate` flag. When this flag is set, each `ScalarResult` component extracts the value of the variable it is monitoring and places it in a special file. When the `generate` flag is not set, these same components assume that results have already been generated and attempts to load them from the results directory (specified in the `TestCase` extends clause). The default value of the `generate` flag is false. So the first time the test case is run it is necessary to set the `generate` flag to generate the baseline results. After that, the `generate` flag should not be set and it will run properly as a check against the baseline results.

It is always a good idea to keep models under version control. In that case, the baseline results should also be version controlled. This is because it will be difficult (if not impossible) to regenerate these baseline results in the future. Note that it is not always necessary to generate baselines results. Some expected result models allow comparisons against analytical solutions. Furthermore, the `UnitTesting` library allows new kinds of expected result models to be created by simply extending from the `UnitTesting.TestResult` base class.

4.4 Running the Test

Once a test case has been created by extending from the `TestCase` base class and adding any expected result components, the test case is ready to be run. Running a test just means running a simulation of the test case model. The `TestCase` base class and expected result components contain all the "code" necessary to automatically determine whether the test case has passed or failed.

There are many things that can cause the failure of a test case. The obvious failure mode is that the computed value for a variable does not match the expected value (within some pre-specified tolerance). But there are other subtle ways a test can fail. For example, if the expected results occur over the time interval from 2 seconds to 7 seconds but the simulation runs from 0 seconds to 6 seconds, this is a failure. Recall our previous definition for a failure was that a test case did not match all expected results. In this case, the simulation failed to match the expected results between 6 seconds and 7 seconds because it did not simulate that period of time. This is a failure because the simulation ended too soon and it did not check all expected results. A similar failure occurs if the simulation starts too late.

If an expected result does not match, the simulation will terminate immediately and generate a message describing the reason for the failure. In some cases, in order to understand why a test is failing it is useful to run the test case to completion in spite of the failure. In this case, setting the `fatal` flag to `false` prevents the immediate termination of the simulation due to failure. This allows the full solution trajectory to be reviewed which may provide some insights into why the test failed.

4.5 Organizing Test Cases

The `UnitTesting` library identifies test cases based on whether they inherit from the `UnitTesting.TestCase` class. For this reason, it does not care how test cases are organized in the package hierarchy. Some developers may wish to keep the test cases in the same library as the library components while others may chose to maintain a separate library just for test cases. The `UnitTesting` library functions equally well in either scenario.

4.6 Automated Testing

Although it may occasionally be useful to run a single test, complete validation requires that all tests be run. For this reason, the `UnitTesting` library includes several built-in functions that, in conjunction with the "Commands" menu in Dymola, allow all tests to be run with the click of the mouse.

4.7 Reports

In the ideal situation, the test suite provides 100% component coverage, 100% condition/decision coverage and 100% state coverage and running all the tests yields no errors. But it takes some time to get to this point. In the meantime, library developers need assistance to understand what kinds of tests are required or what components are the most likely

source of the test failures that occur. This is where the `UnitTesting` reporting capabilities come in.

In addition to the metrics described previously, the `UnitTesting` reports generate additional information like dependency graphs, accountability matrices, decision trees, *etc*. They also include further analysis (beyond whether a test passed or failed) to highlight conservation issues and other information based on simulation results.

The reports are generated as HTML. This HTML is then embedded, using the `Documentation` annotation in Modelica, in special packages that are automatically inserted into the test case library. These packages include the `DocumentationClass` annotation so, like the Modelica Standard Library User's Guide, they can be immediately identified in the package hierarchy as reference documentation.

5 Conclusions

For library developers, testing can help ensure a quality product. Rather than counting on visual inspection of results from a few complex system models, the `UnitTesting` library and associated automation and reporting tools can be used to create a fine mesh of tests to detect errors or discrepancies at the lowest level and quickly identify the source of the problem.

This formal approach provides a rigorous validation and verification process for the underlying models. The presence of such a process may eliminate barriers preventing library components from playing a more significant role in the product development process.

References

[1] Husted, T., Massol, V., "JUnit in Action", Manning Publications, *ISBN 1930110995*.

[2] Busser, R. D., Blackburn, M. R., Nauman., A. M., "Automated Model Analysis and Test Generation for Flight Guidance Mode Logic", *Proceedings of AIAA/IEEE Digital Avionics Systems Conference*, v 2, 2001.

[3] Busser, R. D., Boden, L. M., "Extending Simulink Models with Natural Relations to Improve Automated Model-Based Testing", *Proceedings of 29th Annual IEEE/NASA Software Engineering Workshop, 2005.*

[4] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., Rierson, L. K., "A Pratical Tutorial on Modified Condition/Decision Coverage", NASA Center for AeroSpace Information, *NASA/TM-2001-210876*.

[5] Tiller, M., "Parsing and Semantic Analysis of Modelica Code for Non-Simulation Applications", *Proceedings of the 2003 Modelica Conference*.