EUnit: A Unit Testing Framework for Model Management Tasks

Antonio García-Domínguez[1], Dimitrios S. Kolovos[2], Louis M. Rose[2], Richard F. Paige[2], and Inmaculada Medina-Bulo[1]

[1] University of Cádiz, Department of Computer Languages and Systems, C/Chile 1, 11002, Cádiz, Spain
{antonio.garciadominguez,inmaculada.medina}@uca.es
[2] University of York, Department of Computer Science, Deramore Lane, YO10 5GH, York, United Kingdom
{dkolovos,louis,paige}@cs.york.ac.uk

Abstract. Validating and transforming models are essential steps in model-driven engineering. These tasks are often implemented as operations in general purpose programming languages or task-specific model management languages. Just like other software artefacts, these tasks must be tested to reduce the risk of defects. Testing model management tasks requires testers to select and manage the relevant combinations of input models, tasks and expected outputs. This is complicated by the fact that many technologies may be used in the same system, each with their own integration challenges. In addition, advanced test oracles are required: tests may need to compare entire models or directory trees.

To tackle these issues, we propose creating an integrated unit testing framework for model management operations. We have developed the EUnit unit testing framework to validate our approach. EUnit tests specify how models and tasks are to be combined, while staying decoupled from the specific technologies used.

Keywords: Software testing, unit testing, model management, test frameworks, model validation, model transformation.

1 Introduction

Model-driven approaches are being adopted in a wide range of demanding environments, such as finance, health care or telecommunications [10]. In this context, validation and verification is identified as one of the many challenges of model-driven software engineering (MDSE) [21].

MDSE in practice involves creating models, and thereafter *managing* them, via various tasks, such as model transformation, validation and merging. The validation and verification of each type of model management task has its own specific challenges. Kolovos et al. list testing concerns for model-to-model (M2M) and model-to-text (M2T) transformations, model validations, model comparisons and model compositions in [13]. Baudry et al. identify three main issues when testing model transformations [2]: the complexity of the input and output models, the immaturity of the model management environments and the large number of different transformation languages and techniques.

While each type of model management task does have specific complexity, some of the concerns raised by Baudry can be generalized to apply to all model management tasks:

- There is usually a large number of models to be handled. Some may be created by hand, some may be generated using hand-written programs, and some may be generated automatically following certain coverage criteria.
- A single model or set of models may be used in several tasks. For instance, a model may be validated before performing an in-place transformation to assist the user, and later on it may be transformed to another model or merged with a different model. This requires having at least one test for each valid combination of models and sets of tasks.
- Test oracles are more complex than in traditional unit testing [17]: instead of checking scalar values or simple lists, we may need to compare entire graphs of model objects or file trees. In some cases, we might only want to check specific properties in the generated artifacts.
- Models and model management tasks may use a wide range of technologies. Models may be based on Ecore [20], XML files or Java object graphs, among many others. At the same time, tasks may use technologies from different platforms, such as Epsilon [15], oAW [11] or AMMA [6]. Many of these technologies offer high-level tools for running and debugging the different tasks using several models. However, users wishing to do automated unit testing need to learn low-level implementation details about their modelling and model management technologies. This increases the initial cost of testing these tasks and hampers the adoption of new technologies.
- Existing testing tools tend to focus on the testing technique itself, and lack integration with external systems. Some tools provide graphical user interfaces, but most do not generate reports which can be consumed by a continuous integration server, for instance.

In this work, we propose addressing these issues through an integrated test framework for model management tasks. We illustrate this approach with an improved version of the EUnit framework initially presented in [13]. EUnit has been extended with a richer data model, implicit test setup and improved facilities for testing model transformations and validations, among other new features.

The rest of this work is structured as follows. Section 2 illustrates our previous points with a JUnit test case for a model-to-model transformation. Section 3 describes how EUnit test suites are organized, and Section 4 shows how they are written, with an example. Section 5 outlines how EUnit can be extended to accommodate other technologies. Section 6 shows how we used EUnit to test a model-driven workflow to generate GMF editors. Finally, Section 7 presents related works and Section 8 lists the conclusions for this paper and our future lines of work.

2 Testing a Model Transformation with JUnit

In this section we will illustrate the abstract issues listed in Section 1 using a unit testing framework for a general-purpose programming language to test a model management task. We will test a simple model-to-model transformation in the Epsilon Transformation Language (ETL) using JUnit 4 [3]. The input and output models are based on the Eclipse Modeling Framework (EMF) [20].

ETL is one of the languages implemented in the Epsilon platform [15], which provides an infrastructure for implementing uniform, integrated and interoperable model management languages that can be used to manage models of diverse metamodels and technologies. Like all Epsilon languages, ETL is based on the Epsilon Object Language (EOL). EOL is a reworking and extension of OCL that includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard I/O streams.

Definition of the test suite. For the sake of brevity, we will only outline the contents of the JUnit test suite. It is a Java class with three public methods:

1. The test setup method (marked with the `@Before` JUnit annotation) loads the required models by creating and configuring instances of EmfModel. After that, it prepares the transformation by creating and configuring an instance of EtlModule, adding the models to its model repository.
2. The test case itself (marked with `@Test`) runs the ETL transformation and uses the generic comparison algorithm implemented by EMF Compare to perform the model comparison.
3. The test teardown method (marked with `@After`) disposes of the models.

Issues. We can identify several issues in each part of the test suite. First, test setup is tightly bound to the technologies used: it depends on the API of the EmfModel and EtlModule classes, which are both part of Epsilon. Later refactorings in these classes may break existing tests.

The test case can only be used for a single combination of input and output models. Testing several combinations requires either repeating the same code and therefore making the suite less maintainable, or using parametric testing, which may be wasteful if not all tests need the same combinations of models.

Model comparison requires the user to manually select a model comparison engine and integrate it with the test. For comparing EMF models, EMF Compare is easy to use and readily available. However, generic model comparison engines may not be readily available for some modelling technologies.

Finally, instead of comparing the obtained and expected models, we could have checked several properties in the obtained model. However, querying models through Java code can be quite verbose.

Possible solutions. We could follow several approaches to address these issues. Our first instinct would be to extend JUnit and reuse all the tooling available for it. A custom test runner would simplify setup and teardown, and modelling platforms would integrate their technologies into it. Since Java is very verbose when querying models, the custom runner should run tests in a higher-level language, such as EOL. However, JUnit is very tightly coupled to Java, and this would impose limits on the level of integration we could obtain. For instance, errors in the model management tasks or the EOL tests could not be reported from their original source, but rather from the Java code which invoked them. Another problem with this approach is that new integration code would need to be written for each of the existing platforms.

Alternatively, we could add a new language exclusively dedicated to testing to the Epsilon family [15]. Being based on EOL, model querying would be very concise, and with a test runner written from scratch, test execution would be very flexible. However, this would still require all platforms to write new code to integrate with it, and this code would be tightly coupled to Epsilon.

As a middle ground, we could decorate EOL to guide its execution through a new test runner, while reusing the Apache Ant [1] tasks already provided by several of the existing platforms, such as AMMA or Epsilon. Like Make, Ant is a tool focused on automating the execution of processes such as program builds. Unlike Make, Ant defines processes using XML *buildfiles* with sets of interrelated *targets*. Each target contains in turn a sequence of *tasks*. Many Ant tasks and Ant-based tools already exist, and it is easy to create a new Ant task.

Among these three approaches, EUnit follows the last one. Ant tasks take care of model setup and management, and tests are written in EOL and executed by a new test runner, written from the ground up.

3 Test Organization

In the previous section, we listed some of the issues when testing M2M transformations with a general-purpose framework. In this section, we will describe how the internal structure of EUnit test suites and test cases helps flexibly combine models, tasks and tests.

3.1 Test Suites

EUnit test suites are organized as trees: inner nodes group related test cases and define *data* bindings. Leaf nodes define *model* bindings and run the test cases.

Data bindings repeat all test cases with different values in one or more variables. They can implement parametric testing, as in JUnit 4. EUnit can nest several data bindings, running all test cases once for each combination. Model bindings are specific to EUnit: they allow developers to repeat a single test case with different subsets of models. Data and model bindings can be combined.

Figure 1 shows an example of an EUnit test tree: nodes with data bindings are marked with `data`, and nodes with model bindings are marked with `model`. EUnit will perform a preorder traversal of this tree, running the following tests: A with $x = 1$ and model X, A with $x = 1$ and model Y, B with $x = 1$ and both models, A with $x = 2$ and model X, A with $x = 2$ and model Y and finally, B with $x = 2$ and both models.

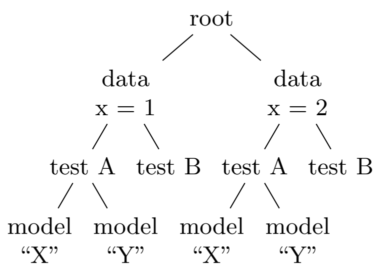


Fig. 1. Example of an EUnit test tree

3.2 Test Cases

The execution of a test case is divided into the following steps:

1. Apply the data bindings of its ancestors.
2. Run the model setup sections defined by the user.
3. Apply the model bindings of this node.
4. Run the regular setup sections defined by the user.
5. Run the test case itself.
6. Run the teardown sections defined by the user.
7. Tear down the data bindings and models for this test.

An important difference between JUnit and EUnit is that setup is split into two parts: model setup and regular setup. This split allows users to add code before and after model bindings are applied. Normally, the model setup sections will load all the models needed by the test suite, and the regular setup sections will further prepare the models selected by the model binding. Explicit teardown sections are usually not needed, as models are disposed automatically by EUnit. EUnit includes them for consistency with the xUnit frameworks.

Due to its focus on model management, model setup in EUnit is very flexible. Developers can combine several ways to set up models, such as model references, individual Apache Ant [1] tasks, Apache Ant targets or Human-Usable Text Notation (HUTN) [18] fragments. This is detailed in Section 4.

A test case may produce one among several results. `SUCCESS` is obtained if all assertions passed and no exceptions were thrown. `FAILURE` is obtained if an assertion failed. `ERROR` is obtained if an unexpected exception was thrown while running the test. Finally, tests may be `SKIPPED` by the user.

4 Test Specification

In the previous section, we described how test suites and test cases are organized. In this section, we will show how to write them.

As discussed in Section 2, after evaluating several approaches, we decided to combine the expressive power of EOL and the extensibility of Apache Ant. For

Listing 1. Example invocation of the EUnit Ant task

```
<epsilon.eunit src="..." failOnErrors="...">
  <model     ref="OldName" as="NewName"/>
  <uses      ref="x" as="y" />
  <exports   ref="z" as="w" />
  <parameter name="myparam" value="myvalue" />
  <modelTasks><!-- Zero or more Ant tasks --></modelTasks>
</epsilon.eunit>
```

this reason, EUnit test suites are split into two files: an Ant buildfile and an EOL script with some special-purpose annotations. The next subsections describe the contents of these two files and revisit the example in Section 2 with EUnit.

4.1 Ant Buildfile

EUnit uses standard Ant buildfiles: running EUnit is as simple as using its Ant task. Users may run EUnit more than once in a single Ant launch: the graphical user interface will automatically aggregate the results of all test suites.

EUnit Invocations. An example invocation of the EUnit Ant task using the most common features is shown in Listing 1. Users will normally only use some of these features at a time, though.

The EUnit Ant task is based on the Epsilon workflow tasks, inheriting some useful features. The attribute *src* points to the path of the EOL file, and the optional attribute *failOnErrors* can be set to `false` to prevent EUnit from aborting the Ant launch if a test case fails. EUnit also inherits support for importing and exporting global variables through the <*uses*> and <*exports*> elements: the original name is set in *ref*, and the optional *as* attribute allows for using a different name. For receiving parameters as name-value pairs, the <*parameter*> element can be used.

Model references (using the <*model*> nested element) are also inherited from the regular Epsilon workflow tasks. These allow model management tasks to refer by name to models previously loaded in the Ant buildfile. However, EUnit implicitly reloads the models after each test case. This ensures that test cases are isolated from each other.

The EUnit Ant task adds several new features to customize the test result reports and perform more advanced model setup. EUnit generates reports in the XML format of the Ant <*junit*> task. This format is also used by many other tools, such as the TestNG unit testing framework [4], the Jenkins continuous integration server [12] or the JUnit Eclipse plug-ins.

The optional <*modelTasks*> nested element contains a sequence of Ant tasks which will be run after reloading the model references and before running the model setup sections in the EOL file. This allows users to run workflows more advanced than simply reloading model references, such as the one in Listing 4.

Listing 2. Example of a 2-level data binding

```
@data x
operation firstLevel()  { return 1.to(2); }

@data y
operation secondLevel() { return 1.to(2); }

@setup
operation generateModel() { -* generate model using x and y *- }

@test
operation mytest() { -* test with the generated model *- }
```

Helper Targets. Ant buildfiles for EUnit may include *helper targets*. These targets can be invoked using `runTarget("targetName")` from anywhere in the EOL script. Helper targets are quite versatile: called from an EOL model setup section, they allow for reusing model loading fragments between different EUnit test suites. They can also be used to invoke the model management tasks under test.

4.2 EOL Script

The Epsilon Object Language script is the second half of the EUnit test suite. EOL annotations are used to tag some of the operations as data binding definitions (`@data`), additional model setup sections (`@model`), test setup and teardown sections (`@setup` and `@teardown`) and test cases (`@test`).

Data bindings. Data bindings repeat all test cases with different values in some variables. To define a data binding, users must define an operation which returns a sequence of elements and is marked with `@data variable`. All test cases will be repeated once for each element of the returned sequence, setting the specified variable to the corresponding element. Listing 2 shows two nested data bindings and a test case which will be run four times: with *x=1* and *y=1*, *x=1* and *y=2*, *x=2* and *y=1* and finally *x=2* and *y=2*. The example shows how x and y could be used by the setup section to generate an input model for the test. This can be useful if the intent of the test is ensuring that a certain property holds in a class of models, rather than a single model.

Model bindings. Model bindings repeat a test case with different subsets of models. They can be defined by annotating a test case with `$with Map {elements}`, where `elements` is a list of key-value pairs. For each key-value pair *dst* $\rightarrow$ *src*, EUnit will rename the model named *src* to *dst*. Listing 3 shows a test which will be run twice: the first time, model “A” will be the default model and model “B” will be the “Other” model, and the second time, model “B” will be the default model and model “A” will be the “Other” model.

Additional variables and built-in operations. EUnit provides several variables and operations which are useful for testing. For example, supporting Ant

Listing 3. Example of a model binding

```
$with Map {"" = "A", "Other" = "B"}
$with Map {"" = "B", "Other" = "A"}
@test
operation mytest() { -* use the default and Other models *- }
```

targets can be invoked with `runTarget("targetName")`. Models written in HUTN [18] can be loaded with `loadHutn("modelName", "hutnSource")`. Ant tasks can be set up from the EOL script using `antProject`, a new global variable which refers to the Ant PROJECT object being executed.

Assertions. In addition to the usual assertions available in most unit testing frameworks, EUnit implements several assertions which are useful for testing model transformations: `assertEqualModels` and `assertNotEqualModels` compare entire models, `assertEqualFiles` and `assertNotEqualFiles` compare files, and file trees can be compared with `assertEqualDirectories` and `assertNotEqualDirectories`.

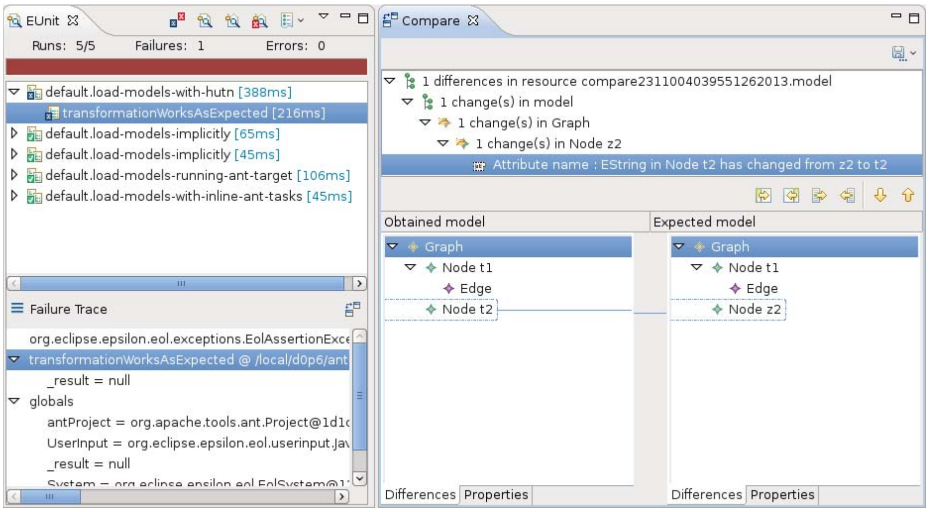
Model comparison is not implemented by the assertions themselves. We extended the Epsilon Model Connectivity abstraction layer [15] to provide model comparison as an optional service of its model drivers, in order to decouple tests from the model comparison engine in use. Additionally, model, file and directory comparisons take a snapshot of their operands before comparing them, so EUnit can show the differences right at the moment when the comparison was performed. This is especially important when some of the models are generated on the fly by the EUnit test suite, or when a test case for code generation may overwrite the results of the previous one.

Figure 2 shows a screenshot of the EUnit graphical user interface. On the left, an Eclipse view shows the results of several EUnit test suites. We can see that the `load-models-with-hutn` suite failed. Users can press the Compare button to the right of “Failure Trace” to show the differences between the expected and obtained models, as shown on the right. EUnit implements a pluggable architecture where *difference viewers* are automatically selected based on the types of the operands. There are difference viewers for EMF models and file trees and a fallback viewer which converts both operands to strings.

4.3 Example: Testing a Model Transformation with EUnit

After describing the basic syntax, we will show how to use EUnit to test the transformation in Section 2.

The Ant buildfile is shown in Listing 4. It has two targets: *run-tests* (lines 2–16) invokes the EUnit suite, and *tree2graph* (lines 17–22) is a helper target which transforms model “Tree” into model “Graph” using ETL. The <*modelTasks*> nested element is used to load the input, expected output and output EMF models. “Graph” is loaded with *read* set to `false`: the model will be initially empty, and will be populated by the ETL transformation.



```
Runs: 5/5    Failures: 1    Errors: 0
default.load-models-with-hutn [388ms]
    transformationWorksAsExpected [216ms]
default.load-models-implicitly [65ms]
default.load-models-implicitly [45ms]
default.load-models-running-ant-target [106ms]
default.load-models-with-inline-ant-tasks [45ms]

org.eclipse.epsilon.eol.exceptions.EolAssertionExce
transformationWorksAsExpected @ /local/d0p6/ant
    _result = null
globals
    antProject = org.apache.tools.ant.Project@1d1c
    UserInput = org.eclipse.epsilon.eol.userinput.Jav
    _result = null
    System = org.eclipse.epsilon.eol.EolSystem@1
```

```
1 differences in resource compare2311004039551262013.model
  1 change(s) in model
    1 change(s) in Graph
      1 change(s) in Node z2
        Attribute name : EString in Node t2 has changed from z2 to t2

Obtained model          Expected model
Graph                   Graph
  Node t1                 Node t1
    Edge                    Edge
  Node t2                 Node z2
```

Fig. 2. Screenshot of the EUnit graphical user interface

The EOL script is shown in Listing 5: it invokes the helper task (line 3) and checks that the obtained model is equal to the expected model (line 4). Internally, EMC will perform the comparison using EMF Compare.

5 Extending EUnit

EUnit is based on the Epsilon platform, but it is designed to accommodate other technologies. In this section we will explain several strategies to add support for these technologies to EUnit.

5.1 Adding Modelling Technologies

EUnit uses the Epsilon Model Connectivity abstraction layer [15] to handle different modelling technologies. EMC has support for EMF models, Java object graphs and plain XML files. Drivers for MDR and Z models are also available.

Adding support for a different modelling technology only requires implementing another driver for EMC. Depending on the modelling technology, the driver can provide optional services such as model comparison, caching or reflection.

5.2 Adding Model Management Tasks

As mentioned in Section 3, EUnit uses Ant as a workflow language. Therefore, the basic requirement to test any model management task with EUnit is that it is exposed through an Ant task. It is highly encouraged, however, that the Ant task is aware of the EMC model repository linked to the Ant project. Otherwise,

Listing 4. Ant buildfile for EUnit with <*modelTasks*> and a helper target

```
<project>
 <target name="run-tests">
  <epsilon.eunit src="test-external.eunit">
   <modelTasks>
    <epsilon.emf.loadModel name="Tree" modelfile="tree.model"
       metamodelfile="tree.ecore" read="true" store="false"/>
    <epsilon.emf.loadModel name="GraphExpected" modelfile="graph.model"
       metamodelfile="graph.ecore" read="true" store="false"/>
    <epsilon.emf.loadModel name="Graph" modelfile="transformed.model"
       metamodelfile="graph.ecore" read="false" store="false"/>
   </modelTasks>
  </epsilon.eunit>
 </target>
 <target name="tree2graph">
  <epsilon.etl src="${basedir}/resources/Tree2Graph.etl">
   <model ref="Tree"/>
   <model ref="Graph"/>
  </epsilon.etl>
 </target>
</project>
```

Listing 5. EOL script using `runTarget` to run ETL

```
@test
operation transformationWorksAsExpected() {
 runTarget("tree2graph");
 assertEqualModels("GraphExpected", "Graph");
}
```

users will have to shuffle the models out from and back into the repository between model management tasks. As an example, a helper target for an ATL [6] transformation with the existing Ant tasks would need to:

1. Save the input model in the EMC model repository to a file, by invoking the <*epsilon.storeModel*> task.
2. Load the metamodels and the input model with <*atl.loadModel*>.
3. Run the ATL transformation with <*atl.launch*>.
4. Save the result of the ATL transformation with <*atl.saveModel*>.
5. Load it into the EMC model repository with <*epsilon.emf.loadModel*>.

This does not prevent EUnit from testing ATL transformations, but it makes the helper task quite longer than the one in Listing 4. Ideally, Ant tasks should be adapted or wrapped to use models directly from the EMC model repository.

Another advantage in making model management tasks EMC-aware is that they can easily "export" their results as models, making them easier to test. To illustrate this point, we extended the Ant task for the Epsilon Validation Language for model validation with the attribute *exportAsModel*: when set, the task exports its validation results as an EMC Java object graph model. This way, EOL can query the results as any regular model (see Listing 6). This is simpler than transforming the validated model to a problem metamodel, as suggested in [5]. The example in Listing 6 checks that a single warning was produced due to the expected rule (`LabelsStartWithT`) and the expected model element.

Listing 6. Testing an EVL model validation with EUnit

```
@test
operation valid() {
  var tree := new Tree!Tree;
  tree.label := '1n';
  runTarget('validate-tree');
  var errors := EVL!EvlUnsatisfiedConstraint.allInstances;
  assertEquals(1, errors.size);
  var error := errors.first;
  assertEquals(tree, error.instance);
  assertEquals(false, error.constraint.isCritique);
  assertEquals('LabelsStartWithT', error.constraint.name);
}
```

5.3 Integrating Model Generators

By design, EUnit does not implement any model generation technique, as we consider that running the tests is orthogonal to generating them. Several model generation tools already exist, such as OMOGEN [7] or Cartier [19]. To EUnit, model generation is just another kind of model management task. There are basically two ways in which models can be generated: *batch* model generation generates all models before repeating every test through them, and *inline* model generation invokes the generator in every test, producing the required models.

Batch model generation can be implemented by calling the Ant task of the model generator before invoking EUnit, and then using a data binding to repeat the tests over every generated model. The Ant tasks required to load these models can be set up by EUnit on the fly in a `@model` operation, using the *antProject* built-in variable. Inline model generation uses data bindings to set the parameters for generating each model, and then invokes the Ant task of the model generation tool in a `@model` operation.

Listing 7 shows a simple example of inline model generation, using EOL code instead of invoking the Ant task of a model generation tool. Several Tree models are generated by combining data and model bindings. The data variable *nlevels* indicates the number of levels the generated binary tree should have. The `@model` operation loads an empty model and populates it as needed. All tests will be repeated 5 times, with complete binary trees of 0 to 4 levels.

6 Case Study: Regression Tests for Eugenia

In this section, we will show a more advanced case study for the EUnit test framework. This case study is a set of regression tests for the Eugenia [14] tool, which simplifies the creation of graphical model editors based on the Eclipse Graphical Modeling Framework (GMF) [8]. The transformations in Eugenia are non-trivial: some of them are implemented in ETL, and some of them are implemented in EOL. Before conducting this case study, testing Eugenia was an entirely manual process, as it was deemed too difficult to automate.

Listing 7. Inline model generation in EUnit

```
@data nlevels
operation levels() { return 0.to(4); }

@model
operation generate() {
  // Load an empty model and populate it
  loadHutn('Tree', '@Spec { Metamodel { nsUri: "Tree" }} Model {}');
  generateBinaryTree(new Tree!Node, nlevels);
}

operation generateBinaryTree(root, nlevels) {
  if (nlevels > 0) {
    for (n in Sequence { new Tree!Node, new Tree!Node }) {
      n.parent := root;
      generateBinaryTree(n, nlevels - 1);
    }
  }
}

/* ... tests ... */
```

After developing EUnit, we decided to use it to add regression tests for the Eugenia model transformations. We created a new Ant task for Eugenia, and defined the EUnit test suite as follows:

– The Ant buildfile contains a single target which prepares a test environment, runs Eugenia on the test environment and invokes EUnit.
– The EUnit test suite uses a data binding to repeat the tests over each of the six models produced by Eugenia: `.ecore`, `.genmodel`, `.gmfgraph`, `.gmftool`, `.gmfmap` and `.gmfgen`.
– Test setup creates, configures and runs <*epsilon.emf.loadModel*> Ant tasks to load the expected and obtained models.
– Test execution compares the expected and obtained models.

Using regular Ant tasks to integrate external tools has the added benefit that the same Ant tasks used for testing can also help end-users in automating their own workflows. If we had defined our own extension framework for EUnit, end-users would not be able to take advantage of these improvements.

EUnit has reduced the amount of code required to do the tests, by repeating tests implicitly through data bindings. The *antProject* variable supplied by EUnit helped simplify the Ant buildfile as well: instead of specifying everything in it, part of the required Ant tasks are created on the fly inside the EOL script.

Overall, our experience using EUnit in this case study has been positive. Still, we have identified several features which would be useful in EUnit. The EOL script could have run Eugenia by itself if EUnit had support for running a specific operation once before or after all test cases, like the `@BeforeClass` and `@AfterClass` annotations in JUnit. With this, the EOL script could do all the work, but users would still need to explicitly write and run an Ant buildfile. It would be convenient to have a launcher which generated and ran a minimal Ant buildfile on the fly, further reducing the learning curve required to use EUnit.

7 Related Work

Initial work on EUnit was presented in [13]. This paper presented the basic testing issues in several common types of model management tasks and showed how model-to-model and model-to-text transformations could be tested with an early prototype of EUnit. The present version of EUnit supports data and model bindings, implicitly reloads models, integrates Ant tasks for model setup and provides a graphical user interface for Eclipse, among other new features. Our current focus in EUnit is to allow users to test efficiently when confronted with the large number of combinations of models, tasks and technologies present in a typical system developed with Model-Driven Engineering.

Lin et al. presented a testing framework for model transformations in [16], identifying three main challenges: automatic comparison, visualization of differences and debugging transformation specifications. Their framework uses the C-SAW model transformation engine, which runs on top of the Generic Modeling Environment platform. Tests are written manually using a textual notation which binds transformations with input and output models. EUnit can be regarded as a more general framework, as it can be used for testing other categories of model management tasks, such as model validations or model-to-text transformations. EUnit delegates comparisons to external engines (such as EMF Compare) and visualizes model, file and directory differences through the Eclipse Compare component. As for the third challenge, interactive debugging was recently added to several Epsilon languages: it could be integrated into EUnit as well by extending the Ant tasks for those languages.

Most of the literature in validation of model management tasks focuses on specific techniques for model transformations, rather than on frameworks to organize them. Baudry et al. show in [7] the OMOGEN tool, which automatically generates input models based on a set of coverage criteria and manually defined *model fragments*. Sen et al. use the Cartier tool to generate models using partition-based testing [19]. Ehrig et al. generate models using graph grammars [9]. These techniques could be integrated in EUnit as model setup tasks.

Mottu et al. identify several test oracles for model transformations [17]: reference transformations, inverse transformations, expected output models, generic contracts, OCL assertions and model snippets. The first three can be implemented using the helper tasks and generic model and file comparison assertions in EUnit. Generic contracts can be checked by repeating a test with data and model bindings. OCL assertions can be emulated with EOL, which is inspired on it. EUnit does not have explicit support for checking if a model snippet is included in the output model, but it can be approximated using EOL.

8 Conclusions and Future Work

Testing any type of model management task involves dealing with several challenges. There are many input and output models, tasks and technologies involved. Models may need to be generated in different ways and tested against several tasks, and a single task may need to be tested against many models. Test oracles are harder to write. The technologies used present additional integration problems when performing automated testing. Existing testing tools do not integrate well with other systems, such as continuous integration servers.

To tackle these issues, we have proposed in this work creating an integrated unit testing framework for model management tasks. To illustrate our ideas, we have developed EUnit, an unit testing framework based on the Epsilon platform:

- EUnit can reuse the same test for many models with suite-wide parametric testing and test-specific model bindings. Parametric testing can integrate hand-written model generation programs into the test suite definition.
- Tests in EUnit are written in the Epsilon Object Language, a high-level imperative language inspired on OCL which is especially well suited for model management. EUnit integrates assertions for comparing models and file trees.
- Modelling technologies are unified by the Epsilon Model Connectivity layer, and model management tasks are wrapped in high-level Apache Ant tasks.
- Ant tasks can be extended to make model management tasks easier to test. For example, the Ant task for the Epsilon Validation Language can now provide EUnit with models of the validation results.
- EUnit provides a graphical user interface for the Eclipse integrated development environment, and generates test reports in the widely used XML format of the JUnit Ant task.

At the same time, there are many ways in which EUnit could be improved. In the near future, we intend to study how EUnit can help test model comparisons and model compositions, while staying decoupled from specific technologies. We also plan to integrate with EUnit the interactive debugging facilities which were recently added to most of the Epsilon languages.

Custom comparison rules could be integrated into the model comparison assertions provided by EUnit. Ant tasks for simplified integration with other platforms (such as AMMA or oAW) could be developed. It would be very interesting to expose model generation tools (such as Cartier) as Ant tasks and use them for model setup. Test specification and organization in EUnit could also be improved with support for running code before and after all test cases and with test groups, theories and assumptions. A test launcher which generated and ran minimal Ant buildfiles on the fly could be useful.

Acknowledgments. This work was partly funded by the research scholarship PU-EPIF-FPI-C 2010-065 of the University of Cádiz. It was also partly supported by the European Commission through the INESS project (Contract #218575).

References

1. Apache Foundation: Ant 1.8.2 (December 2010), `http://ant.apache.org/`
2. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.: Barriers to systematic model transformation testing. Communications of the ACM 53, 139–143 (2010)

3. Beck, K.: JUnit.org (April 2011), `http://www.junit.org/`
4. Beust, C.: TestNG (March 2011), `http://testng.org/`
5. Bézivin, J., Jouault, F.: Using ATL for checking models. Electronic Notes in Theoretical Computer Science 152, 69–81 (2006)
6. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: The AMMA platform support for modeling in the large and modeling in the small. Research Report 04.09, LINA, University of Nantes, Nantes, France (Feburary 2005)
7. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Le Traon, Y.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proc. of the 17th Int. Symposium on Software Reliability Engineering, pp. 85–94. IEEE Computer Society, Los Alamitos (2006)
8. Eclipse Foundation: Graphical Modeling Project (2011), `http://eclipse.org/modeling/gmf/`
9. Ehrig, K., Küster, J.M., Taentzer, G.: Generating instance models from meta models. Software & Systems Modeling 8(4), 479–500 (2008)
10. Guttman, M., Parodi, J.: Real-Life MDA: Solving Business Problems with Model Driven Architecture, 1st edn. Morgan Kaufmann, San Francisco (2006)
11. Haase, A., Völter, M., Efftinge, S., Kolb, B.: Introduction to openArchitectureWare 4.1.2. In: Proc. of the MDD Tool Implementers Forum, TOOLS Europe 2007 (2007)
12. Kawaguchi, K.: Jenkins CI (April 2011), `http://jenkins-ci.org/`
13. Kolovos, D.S., Paige, R.F., Rose, L.M., Polack, F.A.: Unit testing model management operations. In: Proc. of the 2008 IEEE Int. Conf. on Software Testing Verification and Validation, Lillehammer, Norway, pp. 97–104 (April 2008)
14. Kolovos, D.S., Rose, L.M., Abid, S.B., Paige, R.F., Polack, F.A.C., Botterweck, G.: Taming EMF and GMF using model transformation. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010. LNCS, vol. 6394, pp. 211–225. Springer, Heidelberg (2010)
15. Kolovos, D.S., Rose, L.M., Paige, R.F.: The Epsilon Book (March 2011), `http://www.eclipse.org/gmt/epsilon`
16. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. In: Beydeda, S., Book, M., Gruhn, V. (eds.) Model-Driven Software Development, pp. 219–236. Springer, Berlin (2005)
17. Mottu, J., Baudry, B., Le Traon, Y.: Model transformation testing: oracle issue. In: Proc. of the 2008 IEEE Int. Conf. on Software Testing Verification and Validation, Lillehammer, Norway, pp. 105–112 (April 2008)
18. Object Management Group: Human-Usable Textual Notation (HUTN) 1.0 (August 2004), `http://www.omg.org/technology/documents/formal/hutn.htm`
19. Sen, S., Baudry, B., Mottu, J.-M.: Automatic model generation strategies for model transformation testing. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 148–164. Springer, Heidelberg (2009)
20. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional, Reading (2008)
21. Van Der Straeten, R., Mens, T., Van Baelen, S.: Challenges in model-driven software engineering. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 35–47. Springer, Heidelberg (2009)