

Emerging Patterns for Testing Model Management Tools

Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, Tara Gilliam and
Fiona A.C. Polack

Department of Computer Science, University of York, UK.
[louis,dkolovos,paige,tg,fiona]@cs.york.ac.uk

Abstract. The quality of model management tools, such as model transformation and validation tools, is of paramount importance to the adoption and success of Model-Driven Engineering (MDE). To build confidence in the operational correctness of such model management tools and avoid regressions, establishing a rigorous and automated testing process is crucial. In general, model management tools consume models, produce models or both. As such, to test such a tool in a black-box manner, one needs to be able to provide test input models, and check that the operation produces the expected results. This paper contributes to the current state of practice by presenting and comparing a set of novel approaches for performing these two tasks based on our experience of testing the languages and tools of the Epsilon platform.

1 Introduction

In the past, studies [3, 12] have found that the evolution of software can account for as much as 90% of a development budget; there is no reason to believe that the situation is different now. Sjøberg [19] identifies reasons for software evolution, which include addressing changing requirements, adapting to new technologies, and architectural restructuring.

Software evolution can cause a system to become resistant to further evolution: “Over time the code will be modified, and the integrity of the system, its integrity according to [its] design, gradually fades. The code slowly sinks from engineering to hacking.” [5, pg xvi]. A system that is resistant to change, such as the one described above, is termed *brittle* rather than *malleable*.

Automated testing is a key to developing malleable software. The ability to regularly run tests affords developers the confidence to improve the structure of existing code [5]. Moreover, when correcting defects a test case can be used to recreate undesired behaviour, and then to guard against regression. Test-driven development can be used to engineer new features alongside tests, and to produce well-structured code [1].

Model-Driven Engineering (MDE) introduces additional challenges for controlling and managing evolution [11] and testing, particularly because of the application of automated model management operations that manipulate and change models. This paper contributes to improved understanding of testing automated operations in the context of MDE.

1.1 Model Management Tools

Model management is the discipline of managing artefacts used in MDE. Model management is typically implemented as a set of model management *operations*, implemented using languages and tools such as ATL [7], OpenArchitectureWare [16], and Epsilon [9]. Typical model management operations (such as model-to-model transformation, model validation, model merging and code generation) consume models, produce models or both. These operations are designed to be automated and reusable: thus, establishing a rigorous and automated testing process to build confidence on their operational correctness and avoid regressions is critical.

When testing model management operations and tools, we require structures and processes for *constructing* models and for *checking* models. In the course of implementing many operations and tools in the Epsilon platform, we have found that existing software testing frameworks do not provide built-in mechanisms for constructing and checking models and, consequently, for testing these tools in a systematic and automated way. This paper contributes a synthesised analysis of our experience of attempting to test model management tools, and in particular presents a set of emerging *patterns* that we have applied for this testing process. The patterns that we have applied have been developed so as to help to produce model management tests that are malleable and readable, and enhance automation of the testing process.

In Section 2, we introduce relevant terminology and briefly present an existing software testing framework, JUnit. In Section 3, we present model management testing patterns for constructing and for checking models and provide an example implementation for each pattern. Finally, we discuss related work and conclude in Sections 4 and 5.

2 Background and Terminology

A *test* checks the behaviour (according to some specification) of, or *exercises*, some aspect of the system-under-test. A *test* comprises many *test cases*. Test cases can be specified using *assertions* on the system-under-test. A test can also include a *test fixture*, which specifies the context of the test. A test fixture is constructed before test cases are executed and can be shared between test cases.

In this paper, we concentrate on unit testing. *Units* are the smallest testable parts of the system-under-test. A *unit test* exercises a single unit.

xUnit is the name given to a family of unit-testing frameworks. Implementations exist for several programming languages, such as cUnit for C and dotUnit for .NET. In our work, we use JUnit for Java.

JUnit tests are Java classes, containing methods annotated to identify them as either test cases, fixture set up methods or fixture tear down methods. JUnit provides several methods for specifying assertions, such as `assertEquals(Object expected, Object actual)` and `assertTrue(boolean actual)`.

Listing 1.1 shows a JUnit test case (annotated with `@Test`), `singletonShouldContainOneItem`, which uses an assertion to test the state of the collection

object under test (line 9). A test fixture is used to constructs the collection object under test (lines 3–4). Here, we use the `@BeforeClass` annotation to define a test fixture that can be shared between test cases.

```

1 @BeforeClass
2 public static void setUp() {
3     singleton = new ArrayList<String>();
4     singleton.add("itemA");
5 }
6
7 @Test
8 public void singletonShouldContainOneItem() {
9     assertEquals(1, singleton.size());
10 }

```

Listing 1.1. An excerpt of a unit test specified in JUnit.

3 Patterns

In this section, we present the emerging patterns that we have applied while testing model management tools. Each emerging pattern simplifies the construction or checking of models, areas in which we have found existing software testing frameworks to be lacking. For each pattern, we present an example use, along with a brief discussion of motivation and intent. We describe the patterns as emerging because, at this time, their definition is less formal than other patterns (e.g. [6]). A more rigorous definition will emerge through repeated application of these patterns.

For consistency, we will use the families metamodel shown in Figure 1 throughout this section.

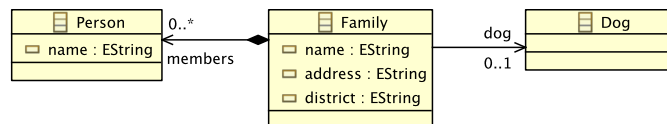


Fig. 1. Exemplar families metamodel.

3.1 Constructing Models

Models for a test fixture, like all other models, can be constructed using the tools provided by a modelling framework, such as the Eclipse Modelling Framework

(EMF) [2]. EMF, for example, can be used to serialise models to XMI and XML documents, which can be stored alongside, or embedded in, unit tests.

Use a metamodel-independent syntax The format in which a modelling framework stores its models may or may not be suitable for specifying test fixtures. For example, XML is optimised for consumption by machines, not humans, and consequently, specifying test fixtures with XML detracts from readability.

Furthermore, the format in which a modelling framework stores its models may use a metamodel-specific concrete syntax. However, in this case information from the metamodel is required to load and store models. Should the metamodel change, the modelling framework may be unable to load existing models [20]. For example, EMF stores all attribute values as strings without type information. Thus, as a result of metamodel evolution, data can be lost when values are inadvertently cast to the wrong type.

To address these issues, we have used a human-readable, metamodel-independent syntax to construct models in test fixtures.

Example OMG's Human-Usable Textual Notation (HUTN) [14] defines a metamodel-independent syntax, which aims to conform to human-usability criteria. Below, we use the HUTN implementation described in [17].

```
1 Family {
2   name: "The Smiths"
3   members: Person { name: "Jill" }
4 }
```

Listing 1.2. HUTN for an exemplar family.

```
1 <?xml version="1.0" encoding="ASCII"?>
2 <families:Family xmlns:families="families" name="The Smiths">
3   <members name="Jill"/>
4 </families:Family>
```

Listing 1.3. EMF XMI for the same family (XMI metadata omitted for brevity).

As Listings 1.2 and 1.3 clearly demonstrate, when specified in HUTN rather than in XMI, models become easier to read and maintain manually by humans. Furthermore, HUTN provides several shorthand constructs that further enhance its readability and conciseness [14]. Also, Epsilon HUTN reports a greater range of model and metamodel inconsistencies than EMF.

Create a Factory class Metamodel-independent syntaxes such as XMI and HUTN are generally less concise than metamodel-specific syntaxes. For some categories of models, a metamodel-independent syntax can be cumbersome. For example, we have found HUTN to be excessively verbose when instantiating

metaclasses with few features, and when constructing deeply nested model structures. An alternative approach is to define a factory class [6], which comprises one or more factory methods for each metaclass.

Example EMF can be used to generate, from a metamodel, Java code for constructing instances of that metamodel. We define factory classes as singletons [6] decorating the metamodel code generated by EMF. Each factory method can provide parameters for specifying the values of features. For specifying the values of many-typed features, we use a Java 5 vararg parameter (see lines 4 and 7 of Listing 1.4). Consequently, our test fixtures need not construct arrays or lists.

```
1 import java.util.Arrays; import families.*;
2
3 public abstract class FamiliesFixtureFactory {
4     public static Family createFamily(Dog d, Person... members) {
5         final Family f = FamiliesFactory.eINSTANCE.createFamily();
6         f.setDog(d);
7         f.getMembers().addAll(Arrays.asList(members));
8         return f;
9     }
10
11     // Omitted for brevity: createDog(), createPerson(String name)
12 }
13
14 // In the test fixture
15 createFamily(createDog(), createPerson("Jill"));
16
17 // Equivalent HUTN
18 Model {
19     contents: Family { dog: Dog "d"
20                       members: Person { name: "Jill" }
21     },
22     Dog "D" {}
23 }
```

Listing 1.4. A Factory for constructing test fixtures.

Again, test fixtures are more readable when models are specified using factory classes rather than XML. Factory classes will also highlight metamodel and model inconsistencies, because any breaking changes to the metamodel will produce compilation errors. We have found factory classes to be more concise than the equivalent HUTN when instantiating metaclasses with few features, and when constructing deeply nested model elements.

Anonymous subclass with an initialiser Factories become harder to use when they contain several methods with similar signatures. Furthermore, for each combination of features that are to be set, a new factory method must be

defined. In these cases, we usually use HUTN to define test fixtures. However, this is sometimes undesirable (for example, when constructing deeply nested model elements), and another approach is required.

We considered adding factory methods with a map (between feature names and values) parameter. Factory Girl [4] for Ruby uses exactly this approach. In Java, however, constructing map objects is less readable than in Ruby. Rather than use map objects, we create an anonymous subclass with a non-static initialiser, a technique described by Norvig in [13]. This rarely-used language feature is denoted simply by a pair of curly braces in the body of the class and can contain arbitrary code; the statements are executed on object creation after superclass initialisation, but before the subclass constructor.

Example: Below, we construct an anonymous subclass of `FamilyImpl`, a class generated by EMF from our metamodel. Lines 2 to 4 set the name, address and district of the family within a non-static initialiser. Values for any feature can be set, and in any order. As anonymous classes have no constructor, this is the last initialisation step and will override any previous values. Any non-private, non-final member variable or method is visible in the block.

```
1 new FamilyImpl() {
2   { setName("John");
3     setAddress("10 Main Street");
4     setDistrict("City of York"); }
5 };
```

Listing 1.5. An exemplar subclass with a non-static initialiser.

We have found this approach greatly reduces repetition across unit tests when used in conjunction with a default superclass. Below, we define a `DefaultFamily` class that sets the name attribute to Bill and the address attribute to 10 Main Street. In the test fixture (lines 10 to 12), we override the `DefaultFamily` with an anonymous subclass that sets the name attribute to John. The value for the address attribute is inherited from `DefaultFamily`.

```
1 // In the factory
2 public static class DefaultFamily extends FamilyImpl {
3   public DefaultFamily() {
4     setName("Bill");
5     setAddress("10 Main Street");
6   }
7 }
8
9 // In the test fixture
10 new FamiliesFixtureFactory.DefaultFamily() {
11   { setName("John"); } // 10 Main Street remains as address
12 };
```

Listing 1.6. Specifying default values in a factory.

Subclasses with non-static initialisers reduce the number of factory methods required to construct test fixtures. Together with default classes, the technique can be used to reduce repetition in test fixtures.

3.2 Checking Models

Often, modelling frameworks provide facilities for comparing models. Depending on the modelling framework, comparison may include superfluous data, such as unique identifiers used only by the modelling framework.

Use a comparison algorithm Some modelling frameworks allow developers to customise the way in which models are compared. When specifying unit tests for model management tools, we have found it necessary to define different comparison algorithms for different metamodels.

Example We have used the Epsilon Comparison Language (ECL) [8] to define comparison algorithms for our metamodels. ECL is a declarative language. Rules are used to specify comparisons between pairs of model elements. ECL supports sophisticated string matching techniques such as fuzzy and dictionary-based matching. Our test cases use an assertion method, `assertEclEqual(File ecl, Model expected, Model actual)`, which decorates the ECL execution engine.

In Listing 1.7, we specify two rules for matching expected and actual model elements. Two persons are equal when their names are equal (lines 8–12). Two families are equal when their names are equal and their members are equal (lines 1–6). We use ECL’s `matches` operation to compare two collections (line 5).

```
1 rule Families
2   match f : Expected!Family
3   with g : Actual!Family {
4
5     compare: f.name = g.name and f.members.matches(g.members)
6   }
7
8 rule Person
9   match p : Expected!Person
10  with q : Actual!Person {
11
12    compare: p.name = q.name
13  }
```

Listing 1.7. An ECL comparison algorithm for the families metamodel.

Only the model elements and values relevant to model comparison need be considered when using a comparison algorithm to check models. Irrelevant data, and data specific to the modelling framework, is ignored.

Use a model inspection language It is common to define related unit tests, each testing a different set of parameters (for example, when testing boundary conditions). The unit under test may produce very similar output for each of the related unit tests. Some test cases can be re-used across all related unit tests.

Unit tests exercise only one unit of the system-under-test. Our model management tools most often construct models by using several units together. Thus, we have rarely needed, when specifying a unit test, to use a comparison algorithm on complete models. Rather, we have needed to compare fragments of models.

When re-using test cases or comparing fragments of models, we have found using a language tailored to inspecting (navigating and querying) models, such as the Object Constraint Language (OCL) [15], to be effective. Assertions can be re-used across unit tests, and any number of, rather than all, model elements can be inspected and checked.

Example: We have used the Epsilon Object Language (EOL) [10] to check models in our unit tests. Like OCL, EOL provides model navigation constructs and, unlike OCL, constructs for updating models, statement sequencing, and simultaneous access to two or more models.

To simplify the use of EOL to perform unit testing, we have defined a class, `ModelWithEolAssertions`, that decorates the EOL execution engine and provides methods specific to unit-testing. In particular, `ModelWithEolAssertions` adds a constructor that allows fragments of EMF models to be loaded, utility methods that allow EOL variables and operations to be specified, and assertion methods that allow checks against the model to be expressed using EOL.

```
1 public class FamilyTest {
2     private static ModelWithEolAssertions model;
3
4     @BeforeClass
5     public static void setup() {
6         // exercise unit under test
7         model = new ModelWithEolAssertions(...);
8
9         model.setVariable("f", "Family.all.first()");
10    }
11
12    @Test
13    public void familyShouldContainMemberCalledJohn() {
14        model.assertTrue("f.members.exists(p | p.name = 'John')");
15    }
16 }
```

Listing 1.8. An exemplar unit test that uses `ModelWithEolAssertions`.

In Listing 1.8, the unit-under-test is exercised, and the resulting model fragment is used to construct a new instance of `ModelWithEolAssertions` (line 6). The `setVariable` method is used to name a `Family` in the resulting model fragment. A test case checks that the `Family` contains a member called `John` (lines 10-13). EOL provides declarative operations for collections, such as the `exists` for expressing existential quantification (line 12).

Using a model inspection language allows unit tests to check a subset of, rather than all, model elements and values. Re-use can be achieved between unit tests by adding a common superclass that includes re-usable assertion methods.

4 Related Work

The testing patterns presented in this paper provide a foundation for implementing others. For example, Schuh and Punke [18] describe the `ObjectMother` pattern for structuring test fixtures. An `ObjectMother` is a factory that returns named fixtures. For example, the “The Smiths,” a family comprising Jack, Jill and their two children, or “The Does,” a family comprising mother Jane and daughter Alice. `ObjectMother`’s named fixtures give rise to *personae*, which provide additional vocabulary for specification of tests. By extending the factory class pattern presented in Section 3.1, `ObjectMother` could be adapted for testing in the context of MDE.

Lin et al. [21] were the first to suggest using model comparison for testing model transformations. They outline a conceptual framework for structuring and executing model-to-model transformations, which uses model comparison to produce test results. However, [21] does not acknowledge the need for metamodel-specific comparison algorithms, and the drawbacks of using comparison for checking output models (discussed in Section 3.2).

5 Conclusion and Future Work

We have presented, compared and contrasted a set of patterns for constructing and checking models. We have discussed the way in which we have used the patterns to develop unit tests for model management tools.

Discovering and documenting further patterns is ongoing work. We will also make improvements to testing tools to better support the patterns presented here. In particular, the majority of the patterns presented would benefit from support for embedding domain-specific languages in JUnit. Tool support for embedding EOL, ECL and HUTN would allow errors in the specification of unit tests to be reported at compile-time, rather than at run-time.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

References

1. Kent Beck. *Test-Driven Development: by example*. Addison-Wesley, 2003.
2. Eclipse. Eclipse Modelling Framework project [online]. [Accessed 1 April 2009] Available at: <http://www.eclipse.org/modeling/emf/>, 2008.
3. Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
4. Joe Ferris. `factory_girl` at GitHub [online]. [Accessed 31 March 2009] Available at: http://github.com/thoughtbot/factory_girl/tree/master, 2008.
5. Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley, 1999.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
7. ATLAS Group. Atlas Transformation Language Project Website. <http://www.eclipse.org/m2m/atlas/>, 2009.
8. Dimitrios S. Kolovos. Establishing correspondences between models with the Epsilon Comparison Language. In *Proc. European Conference on Model Driven Architecture – Foundations and Application [to appear]*, 2009.
9. Dimitrios S. Kolovos. Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website. <http://www.eclipse.org/gmt/epsilon>, 2009.
10. Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. The Epsilon Object Language (EOL). In *Proc. European Conference on Model Driven Architecture – Foundations and Application*, volume 4066 of *LNCS*, pages 128–142. Springer, 2006.
11. Tom Mens and Serge Demeyer. *Software Evolution*. Springer-Verlag, 2007.
12. J Moad. Maintaining the competitive edge. *Datamation*, 36(4):61–66, 1990.
13. Peter Norvig. Java IAQ: Infrequently Answered Questions [online]. [Accessed 31 March 2009] Available at: <http://www.norvig.com/java-iaq.html>.
14. OMG. Human-Usable Textual Notation 1.0 Specification [online]. [Accessed 31 March 2009] Available at: <http://www.omg.org/technology/documents/formal/hutn.htm>, 2004.
15. OMG. Object Constraint Language 2.0 Specification [online]. [Accessed 1 April 2009] Available at: <http://www.omg.org/technology/documents/formal/ocl.htm>, 2006.
16. openArchitectureWare. openArchitectureWare Project Website. <http://www.eclipse.org/gmt/oaw/>, 2009.
17. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack. Constructing models with the Human-Usable Textual Notation. In *Proc. International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 249–263. Springer, 2008.
18. Peter Schuh and Stephanie Punke. ObjectMother: Easing test object creation. In *Proc. XP Universe*, pages 97–105, 2001.
19. Dag I.K. Sjøberg. Quantifying schema evolution. *Information & Software Technology*, 35(1):35–44, 1993.
20. Arie van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proc. Workshop on Model-Driven Software Evolution*, pages 41–49, 2007.
21. Jing Zhang Yuehua Lin and Jeff Gray. Model comparison: A key challenge for transformation testing and version control in Model Driven Software Development. In *Proc. Object Oriented Programming, Systems, Languages and Applications*, 2004.