Model-Based Test Driven Development of the Tefkat Model-Transformation Engine

Jim Steel[1] and Michael Lawley[2]

1: INRIA/Irisa
University of Rennes 1, France
jsteel@irisa.fr

2: Distributed Systems Technology Centre (DSTC)
University of Queensland, Australia
lawley@dstc.edu.au

Abstract

Tefkat is an implementation of a rule- and pattern-based engine for the transformation of models defined using the Object Management Group's (OMG) Model-Driven Architecture (MDA). The process for the development of the engine included the concurrent development of a unit test suite for the engine. The test suite is constructed as a number of models, whose elements comprise the test cases, and which are passed to a test harness for processing. The paper discusses the difficulties and opportunities encountered in the process, and draws implications for the broader problem of testing in a model-driven environment, and of using models for testing.

1 Introduction

The Object Management Group's Model-Driven Architecture [12] describes a scenario for the development of applications where the concept of models is central, and where these models are connected by transformations. This approach builds on and leverages the existing OMG modelling specifications such as the Unified Modelling Language (UML) [17], the XML-Based Model Interchange (XMI) [18], and in particular the Meta-Object Facility (MOF) [15].

As the central concept for the connection and interrelationship of models in a model-driven engineering scenario, it is vital that model transformations be very reliable, and that this reliability can be validated by thorough testing. Such testing must be done at two levels. Firstly, the language/s used to describe and define the transformations must be validated, and secondly, the transformations themselves must be validated.

There are generally two approaches to the implementation of a language for model transformations. In the first, each transformation is used as the basis for the generation of a component that will transform the models. In the second, a single "transformation engine" is used to ensure that the models provided obey a transformation, and to alter one or more of them until they do.

In this paper, we present the architecture used for development and implementation of a test suite and harness for a transformation engine for a specific model-transformation language, XMorph. This testing process presented a number of unique challenges, because of the challenges of testing in an MDA environment, and because of the role played by a transformation engine within this environment.

Testing in MDA represents a new challenge compared to testing in traditional systems [3] or object-oriented systems [4]. This is because the data involved are models, which have significantly more complex structures than simple data types or traditional objects. In fact, the testing problems in MDA more closely resembles those of testing interpreters or virtual machines [19], and are also related to testing grammars [13] or compilers [12].

This paper is divided as follows. In section 2, we give a background to the MDA, and in particular to the concepts of models and model transformations, and the various approaches to transformation. Following this, in section 3 we give a more detailed description of the specific model transformation language used, XMorph. In section 4 we give a broad overview of the general architecture of the Tefkat engine, which implements XMorph. Following this, in section 5 we detail the architectures and processes involved in the testing of the engine's two major components: the pattern matcher and the pattern resolver.

Finally, in sections 6 and 7, we draw conclusions on the problems encountered, and offer insights into the broader problems of testing in an MDA setting.

2 Background to MDA

The central idea of MDA is that a system's architecture can be expressed as a number of models, related by model transformations. The idea is most commonly applied to the process of system development, whereby a system is specified using one or several high-level "platform-independent" models, which are successively refined into increasingly "platform-specific" models, which are eventually realised as source code for the running system.

The central concepts of MDA are those of metamodel, model and model transformation. These are described briefly below.

2.1 Models and Metamodels

At its most abstract level, a model is a series of objects that represent the elements of some system. These models are described by a modelling language, which is defined in MDA by a metamodel. A metamodel is also a model, but more specifically it is a model that describes the structure of a language. In the MDA, metamodels are defined using the Meta-Object Facility (MOF) [15], which is thus a meta-metamodel. The problem of successive metamodels is solved by reflection: the MOF is self-describing.

The relationship between these can be seen in the following example. If one takes the case of a banking database, at the lowest level one has the data: John Smith, Savings Account number 12345, and Commonwealth Bank Ballarat Branch. These are described by the database schema, a model which might include: Accounts Table, Person Column, AccountNumber Column. The metamodel for database schemas, in turn, contains concepts like Table and Column, and that Tables Contains Columns. This in turn is described by MOF classes and associations.

2.2 Model Transformations

In 2002, the OMG issued a request for proposals [16] of a language for describing the relationships between two or more models as described by their respective metamodels. A number of languages [6] have been submitted, and the process of merging and selecting a single language for adoption as a standard is still underway. Generally, the languages can be separated into three groups.

The first group proposes languages which are imperative in their control structures. These languages include submissions based on UML Action Semantics, and those bearing more similarity to languages like Java.

The languages of the second group provide pattern matching for selection of source-model elements, and then imperative constructs for the subsequent creation of target-model objects. These languages bear structural similarity to languages such as XSLT and awk.

The third group uses pattern matching for the selection of source-model elements, but also uses patterns for the subsequent population of target models. The language under consideration here, XMorph, belongs to this third group, and is described in more detail in the following section.

2.3 Testing Transformations

The experiences of testing a model transformation engine discussed in this paper are a special case of the more general problem of testing a model transformation.

To understand this relationship, let

- t be a transformation engine,
- T be a transformation specification,
- M_S be a source model,
- M_T be a target model, and
- O be an oracle, either partial, or preferably total.

Then, a test case is a tuple, $<T, M_S, M_T, O>$.

As described in this paper, to test the transformation engine, t, we take T and M_S then compare the result of $t(T, M_S)$ with M_T based on O. This determines whether t is behaving either correctly or incorrectly.

To test a transformation specification, T, we would do exactly the same thing, but by assuming that t behaves correctly, we must conclude that it is the specification, T, that is correct or incorrect.

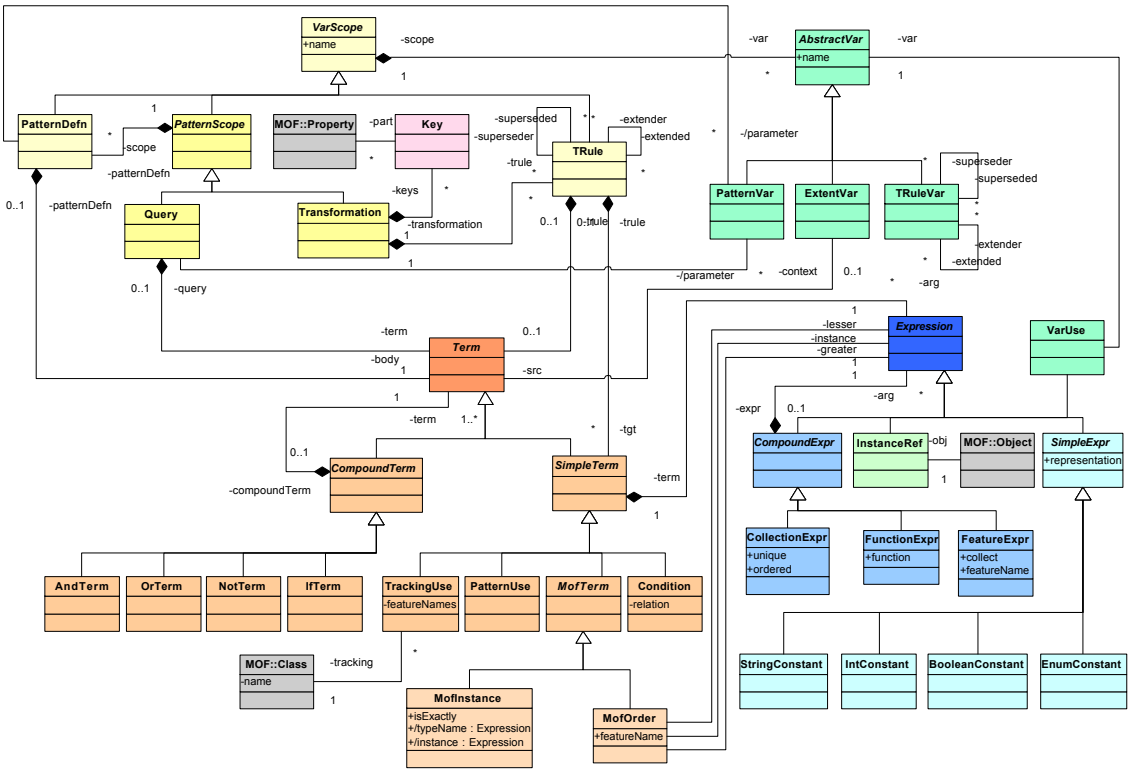

Figure 1 Abstract Syntax of the XMorph Language

3 The XMorph Transformation Language

The XMorph language, described more completely in [8], and motivated by [9] defines transformations as a set of rules. Each rule consists of a pair of constraints: one for selecting a pattern from the source model/s, and another for populating a pattern in the target model/s. These patterns are expressed as parameterised logical constraints, where the parameters provide the link between the source-model pattern and the target-model pattern. Furthermore, the language allows patterns to be defined outside of rules, and to be referred to by rules.

The structure language is specified using a MOF metamodel and is shown in Figure 1. In addition to the metamodel, the language has an associated concrete syntax which, for usability reasons, is based on the style of SQL. An extract from a transformation written in this concrete syntax is as follows.

```
TRANSFORMATION uml2rdbms(uml, rel)

IMPORT UMLRelationalTracking.ecore

RULE Class2Table(c, t)
    FORALL  UMLClass c
    WHERE   c.kind = "persistent"
    MAKE    Table t, Key k
    SETTING t.key = k;

PATTERN hasAttr(c, a)
    FORALL UMLClass c
    WHERE  (c.attributes = a
        OR (c.superclass = c2
            AND hasAttr(c2, a));
```

After specifying its name and its parameters, the transformation specifies its tracking metamodel. This model represents the structure of the relationship between, in this case, the UML and RDBMS models that are the parameters of the transformation. Since the Tefkat engine uses the Eclipse Modelling Framework [5] for its models, this reference points to an ECore XML file that contains the metamodel.

The transformation then goes on to specify a number of rules and patterns that constrain the population of this structure with actual relationship between, in the case of

the first rule, UML Classes and RDBMS tables. The rules may either define the constraints themselves, or reference constraints defined as patterns, as in the case of the hasAttribute pattern in the example. More complete versions of this transformation can be found either in [8] or on the Tefkat Demonstration web site [7].

4 The Tefkat Engine

The Tefkat engine is a Java-based implementation of the XMorph transformation language, which uses the Eclipse Modelling Framework (EMF) [5] as its modelling environment. The particular components of Tefkat whose testing environments are of interest here are the source pattern matcher, and the target pattern resolver. A number of other components, such as the rule orderer and the model reconciler, also form important parts of the system, but they are not discussed here.

The source-pattern matcher evaluates a parameterized pattern definition, and returns all of the parameter bindings in the source model/s which satisfy the provided pattern. A source pattern is described by a tree of Term and Expression instances contained by a root instance of a concrete sub-type of Term.

The target-pattern resolver takes these bindings and, for each one, ensures that the appropriately instantiated target pattern is resolved in the target model/s. A target pattern is described by a set of concrete sub-types of SimpleTerm and their contained Expression instances.

Both these components effectively implement a back-tracking tree-walker where the source-pattern matcher checks the existence of object instances and the values of their attributes while the target-pattern resolver may create objects or set attribute values to ensure a target pattern holds.

Naturally, the structure of these components is very similar. In each case, the component consists of separate handlers for each concrete subtype of Term in the XMorph metamodel (see Figure 1), e.g. for AndTerm, OrTerm, MofInstance, and Condition.

Each of these handlers deals with a number of possible cases for the values contained by the constituent model elements corresponding to the contained Expressions containing an unbound variable or being evaluable to one or more values.

The engine as a whole comprises approximately 4000 to 5000 lines of Java code. The components under discussion here comprise approximately 1000 lines of code apiece.

5 Testing the Engine

This section describes the general approach to testing the engine, and the test frameworks that were developed for two of the most significant components of the Tefkat engine; the source-pattern matcher, and the target-pattern resolver, as described in the previous section.

5.1 General Approach to Testing

The engine as a whole was developed using a bottom-up approach using a methodology including test-driven development and other agile-methods techniques. Being the most important, and most complex, components of the system, the pattern matching and resolving components were the main focus of testing.

5.1.1 Partitioning Into Units

There is some conjecture about the best technique for assigning units in object-oriented development. Many sources advise the treatment of each class as a unit, while others support alternative divisions such as by method.

In this case, the engine is implemented using a service-oriented paradigm, and its primary role is one of data-centric processing. As such, the components were divided into units based on the structure of the XMorph language. That is, each unit corresponds to the functionality that deals with a single concept in the language metamodel (as shown in Figure 1). In general, this corresponds to a single method in the implementation code.

5.1.2 Test Case Selection

The selection of test cases was performed using criteria on the XMorph language metamodel elements. For each metamodel element under test, both functional and robustness test cases were generated. The elements, such as MofInstance, requiring testing with the target resolver represent a subset of those requiring testing with the source matcher, but the different semantics dictate that the test cases be distinct. These test case structures are discussed in sections 5.2.1 and 5.3.1.

For functional tests, data was generated based on the coverage of interesting structural partitions of the appropriate metamodel element. For example, to test an AndTerm, it was required to test the conjunction of two Terms evaluating to “true”, two terms evaluating to “false”, a combination of “true” and “false”, and subsequent cases for conjunctions of more than two terms.

Two robustness criteria were used. The first involved the generation of structures that were illegal according to the constraints on the language’s metamodel. For example, an AndTerm containing no Terms should be detected as illegal because it disobeys the rule of having one or more contained Terms. Secondly, robustness test cases were generated for models disobeying the static semantics of the language as expressed in its formal description in [6]. For example, a MofInstance whose typename does not resolve to a valid type should be reported as an error.

The application of these criteria to generate test cases was done by hand. This was necessitated in part by the lack of a formal (machine-readable) description of many well-formedness rules (beyond those in the metamodel) in the specification, and partly for reasons of expediency.

5.1.3 Oracle

The nature of the oracle is the most significant difference between the testing of the two components. Both are partial oracles; they only partially discriminate between results that indicate success or failure of a test. The oracle for the source matcher consists of encoded textual strings, but this proved insufficient for the target resolver. We describe this in more detail below.

5.2 Test Framework for the Source Pattern Matcher

The source pattern matcher consists of ~1000 lines of Java code, although it also makes use of a large amount of the system's structural library code. 34 separate test cases were developed for the initial phase of the source pattern matcher's development, which supported approximately 6 units of the system.

5.2.1 Test Data Structures and Generation

The inputs for the source-pattern matcher consist of a pattern with which to match, and a model to match against.

Since the body of a PatternDefn is equivalent to the src of a TRule (see Figure 1), PatternDefns were used as containers for test case patterns to allow separation of testing the source pattern-matcher from the target-pattern resolver. Each test case was defined in a PatternDefn, and all test case PatternDefns were contained within a single Transformation, as the test suite. To separate PatternDefns that should be considered as test cases from those included to test the PatternUse term, a "test" prefix to the name of the PatternDefn was added, followed by a nominal name of the test case, for diagnosis purposes.

For the models against which these patterns match, it was decided to use the test-case model itself. This is possible because transformations, and hence the test cases, are themselves defined as models. For example, a test case for MofInstance might consist of a pattern to match against any instances of the class called "PatternDefn". If it were the sole test case in the test suite, it would match once, against itself. It was expected that the large size of the test suite as a source model would allow for definition of sufficient test cases to cover the range of language features.

This decision was made to avoid the problem of having to define separate source models for each test case. This was a particularly unattractive proposition in the early stages of development, when the test-case input model instances were being explicitly constructed in the JUnit initialisation. However, the decision later lead to problems of test case interference. This is discussed in Section 6.2.

5.2.2 Oracle

The oracle for the test system was encoded explicitly in the name given to each test pattern. More specifically, the name was divided as follows:

1. the "test" prefix, to indicate that the pattern should be considered as a test case,

2. the name of the test case, for diagnosis purposes, and

3. one of "True<N>", "False", or "Exception" as a suffix, to indicate the expected outcome of the test case.

"False" indicated that no matches were expected for the rule. "True" indicated that the rule should match, and a subsequent integer N could, optionally, indicate the number of expected matches. "Exception" indicated that the test was intended as a robustness test, and should result in an exception.

A full oracle for the source-pattern matcher would have included the provision of a set of bindings of all the variables in the test pattern to objects in the source model for the true case. However, for the sake of expediency, it was decided to omit the expected bindings and use only a partial oracle.

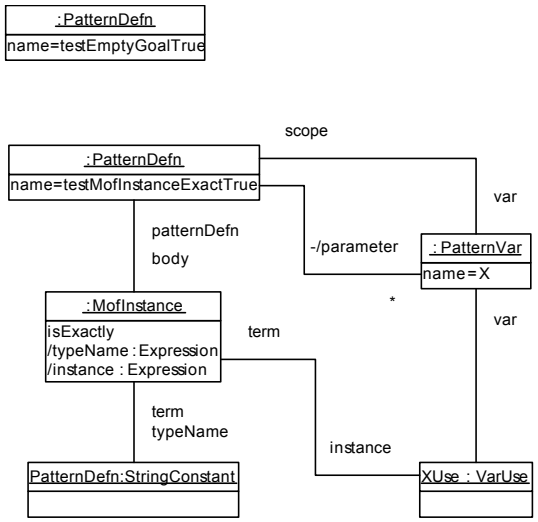

Figure 3 Two simple tests for the Source-Matcher

Figure 3 illustrates two simple functional test cases for the source-pattern matcher. The first, "testEmptyGoalTrue", is a simple test case of a pattern with no constraint as its body, which should match true. The second, "testMofInstanceExactTrue", is a test case for MofInstance, which tests for the existence of any object exactly of the type PatternDefn. This should succeed, since there is an object of that type in the test case itself, and another in the "testEmptyGoalTrue" test case. The textual version of these tests, in the XMorph concrete syntax, is as follows.

```
PATTERN testEmptyGoalTrue( )

PATTERN testMofInstanceExactTrue(x)
    FORALL  PatternDefn x
```

5.3 Testing the Target-Pattern Resolver

This section describes the techniques used to test the target pattern resolver component of the engine. As with the source patter matcher, the target pattern resolver component is divided into units based on the handling of single metamodel elements, and comprises ~1000 lines of Java code.

The testing of this component presented the greatest challenge in terms of the architecture of the test structures and environment. Only 4 test cases were produced before it was realised that a more heavyweight approach was required to testing this component. The reasons for this realisation are discussed below, and further in section 6.4.

5.3.1 Test Data Structures and Generation

The inputs for the target-pattern resolver consist of a transformation rule with a source pattern for matching and a target pattern for resolving, and a model to match against. In this way, the testing of the target resolver also depended on the successful execution of the source matcher.

For the target-pattern resolver, each test case was defined as a transformation rule, and all test case rules were contained within a single transformation. Again, a "test" prefix to the name of the rule was added to signify that the rule represented a test case, and it was followed by a nominal name of the test case, for diagnosis purposes.

5.3.2 Oracle

A full oracle for the target-pattern resolver would include, for each test case, the expected result in the form of a model that could be compared with the actual output from the resolver. However, again for the sake of expediency, a partial oracle was preferred.

The same naming scheme was used for encoding the oracle, but this proved to be somewhat limited in expressive power since tests of rules that should succeed really need to check that the resulting target model actually contains the expected objects with the expected values for attributes and references. Despite this limitation in completeness, simply checking for an expected success, failure, or exception proved quite effective for the initial implementation of the target-pattern resolver.

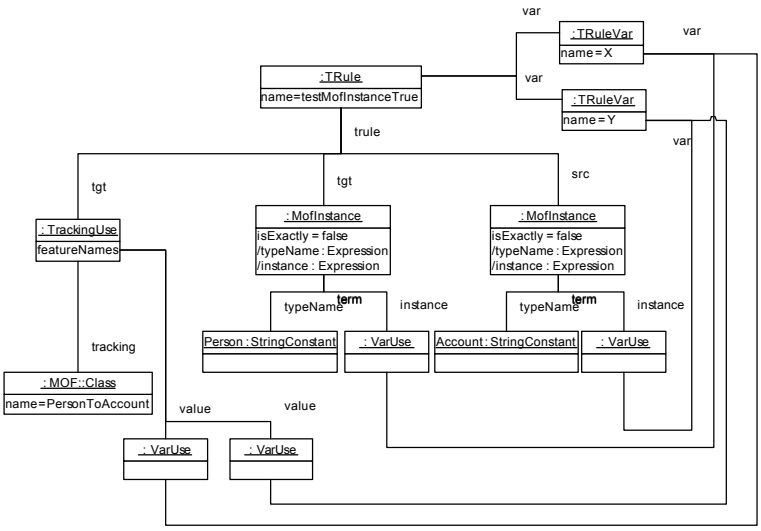

Figure 4 Simple test case for the Target-Resolver

Figure 4 shows an example of a test case for the target-pattern resolver, in which the existence of an instance in the source model of the Person class should result in an instance in the target model of the Account class, and a relationship between the two corresponding to the PersonToAccount tracking class. The textual version of this test case is as follows.

```
RULE testMofInstanceTrue(x, y)
  FORALL  Person x
  MAKE    Account y
  LINKING PersonToAccount
     WITH person = x, account = y;
```

6 Observations

During the development and application of these testing mechanisms, a number of observations were made of the problem in general. The most significant of these are discussed in the following sections.

6.1 Test Harness

The test harness was developed using JUnit [10]. Initially, the test cases were constructed using programmatic interfaces to EMF to create the necessary transformation language structures.

However, this quickly became repetitive and unwieldy, with test cases becoming unreasonably difficult to define and excess effort spent debugging the construction of the test cases themselves. As such, the switch was made to defining the test cases separately from the test harness as models defined in the XMI [18] notation, which could then be edited either directly by editing the XMI files or through the graphical interfaces provided by the EMF framework and Eclipse.

The JUnit harness was then modified to load in the test models and to run test cases based on their contents.

6.2 Test case interference

The decision to use the test patterns as the source models for themselves was one made to reduce the initial setup cost of the test suite, in that it obviated the need to define a separate source model. However, from a maintenance standpoint it raised the problem that a change to the test suite, including the addition of a new test case, also meant a change to the source model for each test, and hence possible changes to the oracles of each of the existing test cases.

Creating a separate, but still common, source model partially solves this problem, however, in this case it is still often necessary to add elements to the source model as new tests are added to the test suite and this leads to the same maintenance problem. To avoid this we refactored the testing framework to use multiple pairs of transformation model and source model (i.e., test suites).

For testing transformation rules, the use of multiple transformation/source model pairs became necessary since the evaluation of a transformation rule usually has side-effects and we needed to ensure that groups of rules did not interact with each other.

6.3 Use of models as test data

The use of models as test data resulted in a dramatic increase in the ease of defining a test case pattern and its source model. This was due to the intrinsic advantage of using the EMF-generated metamodel-specific editor rather

than more general programmatic interfaces, and is a truism for the definition of most complex data structures.

From a methodology standpoint, this increased ease of test definition lead to developers creating many more tests. This in turn helped to accelerate the development of the engine, and to guide the direction of development effort.

However, the most conspicuous problem raised by this move was that of version maintenance. When the discovery of shortcomings in the transformation language led to changes in the language metamodel, the use of XML documents as models meant that even a small change in the DTD or schema generated from the metamodel resulted in the test data failing to be recognized by the metamodel-specific tools. At this point it became necessary to hand-edit the XMI of the models to roll forward the test data to the new version of the transformation language.

On one hand this should not be surprising; if the required semantics of a system change, then one would expect that the tests will need to change to reflect the new semantics. On the other hand, the inability to gracefully handle unexpected XML indicates a degree of brittleness in some of the EMF-supplied infrastructure on which we were building. This problem, characterised as Version Skew, is further discussed in section 7.2.

6.4 Limited Oracle

The naming-scheme based oracle proved very effective for the initial set of test cases since they focused on testing boundary conditions and simplistic categorizations of functionality of individual language elements.

However, when attempting to construct effective test cases for more complex combinations of language elements (e.g., OrTerm and IfThenTerm), the limited expressiveness of True<N>/False/Exception becomes a major stumbling block.

The natural solution to this problem, that of adding expected target and tracking models to the specification of a test suite is not entirely straightforward, since one needs a mechanism to perform an equivalence comparison of the actual output from a transformation with these sample models. In the context of EMF this is non-trivial, since the ECore metamodel has no general mechanism for specifying which, if any, features of a class are identifying (i.e., comprise a key)[1]. Thus, given any two Java Objects representing instances of one class, there is no way to determine whether or not they represent the same instance.

In the short term we plan to investigate the use of several XML-diff algorithms [21] applied to the XMI representations of the various models. This is complicated by the fact that, while the order of elements is significant in a generic XML document, it is sometimes but not always significant in an XMI document.

In the longer term, some research has also been done into the field of model difference and union [1], which may alleviate some of the problems encountered by the mapping of the model to XML.

6.5 Automated test model generation

An unexploited opportunity of using models to drive the testing was the possibility of semi-automatically generating a significant number of test cases.

Because the test models are instances of a known meta-model that includes such information as cardinality constraints on references, it is possible to automate the construction of sets of model instances that correspond to simple valid instances of the metamodel that cover the various corner cases. Such corner cases include cardinalities of 0, 1, or > 1 for multi-valued references, and references to instances of concrete sub-types of the reference's type. For example, three test cases could be generated for AndTerm corresponding to an AndTerm containing zero Terms, an AndTerm containing one Term, and an AndTerm containing more than one Term.

Having generated such test models, one is then faced with a number of problems:

- How values should be assigned to the attributes of the model elements, especially since the choice of values will determine the expected outcome of the test case.

  Indeed, many structurally equivalent test models may be needed with alternate attribute values in order to achieve appropriate coverage.

  It may be possible to further automate the process of assigning attribute values by providing the test model generation process with sample source metamodels, models, and tracking models.

- Many constraints on the model are not explicitly manifest in the metamodel, so some of the generated instances will violate these constraints and should therefore be tagged as robustness cases (Exceptions) in the oracle.

- Since CompoundTerms can recursively contain other CompoundTerms, the containment depth

[1] ECore does allow a single attribute to operate as a key, but does not support multi-valued keys, nor is there a mechanism for comparing the identity of objects where no attribute operates as a key.

needs to be limited to avoid generating an infinite number of test models.

We are currently investigating this issue, using two approaches. In the first, the structural information of the transformation, i.e. the source and tracking metamodels, can be used to facilitate test generation, as above. In the second, information in the specification (black-box) or definition (white-box) can be used to enhance the test cases by identifying interesting subsets of the source metamodels.

7 General Relevance

Although interesting in themselves, the principal significance of these observations is that they point to issues that apply generally to testing in a model driven environment.

7.1 Relationship to Testing Virtual Machines

Virtual machines are the most similar comparable technology because they operate on byte-code which most closely resembles the notion of a model instance. This is because the tests of interest are performed at a level independent of any on-disk representation of the input. Thus we are not testing the syntactic correctness of some serialisation of a model instance, but rather the correctness of the implemented behaviour of the model instance.

Furthermore, the transformation engine is the most generic of MDA components. Typically, the engine takes in a transformation (represented by a model instance), as well as a number of source and target model instances, as parameters. For meaningful execution, these models must be appropriately related; the source and target models must be typed according to the types used in the transformation. Thus testing the engine requires providing appropriate transformations as well as corresponding source and target models. Drawing on the analogy with testing virtual machine implementations, the transformation corresponds to some byte-code and the source and target models to the initial and final state as manipulated by the byte-code. Where this differs from byte-code is that it is relatively trivial to twiddle the bits of byte-code to produce more-or-less valid, but possibly semantically quite different, variations of a test case. This is possible because byte-code is an encoded representation and is untyped. The model instances that an MDA tool manipulates are strongly typed which makes meaningful random mutation much harder.

In Section 6.5 above we discussed issues surrounding automated test model generation. The work of Sirer [19] on using grammars to help automate generation of test cases for the Java virtual machine suggests that this may be adaptable to guide the automated generation of test model instances. Note however that these tests are mostly useful for detection of gross errors and performance problems rather than checking for semantics correctness since they are unable to automatically generate the required expected output of a generated test case.

7.2 Version Skew

Version skew in an MDA environment occurs when a metamodel is changed, and its instances (e.g., M_S, T, M_T, etc) are no longer valid instances of the new metamodel. In our case, since the metamodel describing the transformation language is relatively stable, the problem of version-skew between the transformation language metamodel and the test cases was not a major issue.

However, in the broader context of testing transformation specifications, it is likely to be much more of a problem. One approach to dealing with this version skew that is being explored [10, 20], is to explicitly represent the changes to the metamodel, and to semi-automatically produce a transformation specification that migrates instances of the old metamodel to instances of the new metamodel.

However, all-or-nothing nature of parsing XMI documents, with respect to unexpected XML elements, indicates that practical and mature MDA tools will need to aim for more graceful handling of error conditions.

8 Conclusion

Adopting a model-based test-driven approach to implementing the Tefkat transformation engine lead to greater productivity, a more robust implementation, and enhanced the experience for the implementers. The test infrastructure that was developed offers both insights into testing in a model-driven environment, and a practical starting point for a more general testing framework for model-driven systems.

It is not surprising that writing tests and ensuring they pass leads to more robust code. More interestingly, the test-driven approach provided structure to the implementation process, which resulted in less time being spent in deciding what to do next. It also required that a runnable, albeit incomplete, version of the system be produced in small increments. This in turn led to an increased sense of implementer satisfaction and a resulting enthusiasm to make progress, all well-documented results of the extreme programming approach.

A direct contribution to this was the upgrade of the testing harness to directly load XMI representations of test cases rather than requiring programmatically constructed test cases. Since tools were available to directly construct the test cases, it became a much less arduous task and the implementers began constructing more test cases so that each language unit was given greater test coverage.

As a result of the work reported here, a number of conclusions can be drawn about testing practices for systems built using MDA principles. Firstly, it is important to maintain test models in serialized form (such as XMI), rather than programmatic form, to better provide for the efficient definition and evolution of the test suite. Secondly, a structured approach to the definition and management of the test cases is necessary to avoid problems of test interference. The issues surrounding the comparison of models by an oracle are currently under exploration, and it is expected that the use of the <XMLUnit/> framework will allow for the provision of the required enhanced expressive power.

As discussed, the testing of a model transformation engine has considerable bearing on the testing of individual model transformations. We feel that the test harness developed for Terfkat can, with extensions to provide a more complete oracle, serve as the basis of a more general framework for testing model transformations. Associated issues, such as version skew, are the subjects of active research, and are likely to be addressed as model-driven engineering matures as a discipline.

9 Acknowledgements

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

References

1. M. Alanen and I. Porres. Difference and Union of Models. UML 2003, pages 2-17, Springer, 2003.

2. T. Bacon. A Tour of XMLUnit. Available from http://xmlunit.sourceforge.net

3. B. Beizer. Software Testing Techniques, 2nd Edition. Van Nostran Reinhold, 1990.

4. R. Binder. Testing object-oriented systems. Addison-Wesley, 2000.

5. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. Eclipse Modeling Framework. Addison & Wesley, 2003.

6. DSTC, IBM, CBOP. MOF Query/Views/Transformations, Second Revised Submission. OMG Document no. ad/2004-01-06, January 2004.

7. DSTC, Pty Ltd. Tefkat Demonstration, http://www.dstc.edu.au:8080/qvt/, April 2004.

8. K. Duddy, A. Gerber, M. Lawley, K. Raymond, and J. Steel. Model Transformation: A declarative, reusable patterns approach. EDOC 2003, pages 174-185, IEEE Computer Society, 2003.

9. A. Gerber, M. Lawley, K. Raymond, J. Steel, and A. Wood. Transformation: The Missing Link of MDA. ICGT 2002, pages 90-105, Springer 2002.

10. D. Hearnden, P. Bailes, M. Lawley, and K. Raymond. Automating Software Evolution. IWPSE 2004, September 2004.

11. JUnit, Testing Resources for Extreme Programming. http://www.junit.org, April 2004.

12. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin and V. Shishkov. Coverage-Driven Automated Compiler Test Suite Generation. Electronic Notes in Theoretical Computer Science, Vol 82, Issue 3, Elsevier 2003.

13. R. Lämmel. Grammar Testing. FASE 2001, pages 201-216, Springer 2001.

14. Object Management Group, Executive Overview: Model-Driven Architecture. Available from http://www.omg.org/mda.

15. Object Management Group (OMG), Meta-Object Facility 2.0 Core Final Adopted Specification, OMG Document ptc/2003-10-04.

16. Object Management Group (OMG), MOF 2.0 Query/Views/Transformations RFP, OMG Document no. ad/2002-04-10, April 2002.

17. Object Management Group (OMG), Unified Modeling Language (UML) Specification

18. Object Management Group (OMG), XML-Based Model Interchange (XMI) Specification.

19. E.G. Sirer. Testing Java Virtual Machines. International Conference on Software Testing And Review, San Jose, California, November 1999.

20. J. Sprinkle. Metamodel Driven Model Migration. PhD Dissertation, August 2003.

21. Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. ICDE 2003.