A Testing Framework for Model Transformations

Yuehua Lin, Jing Zhang, and Jeff Gray

Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, Alabama, USA {liny,zhangj,gray}@cis.uab.edu

As models and model transformations are elevated to first-class artifacts within the software development process, there is an increasing need to provide support for techniques and methodologies that are currently missing in modelling practice, but provided readily in other stages of the development lifecycle. Within a model transformation infrastructure, it is vital to provide foundational support for validation and verification of model transformations by investigating and constructing a testing framework focused at the modelling level, rather than source code at the implementation level. We present a framework for testing model transformations that is based on the concepts of model difference and mapping. This framework is integrated with an existing model transformation engine to provide facilities for construction of test cases, execution of test cases, comparison of the output model with the expected model, and visualization of test results. A case study in model transformation testing is presented to illustrate the feasibility of the framework.

1 Introduction

To improve the quality of software, it is essential to be able to apply sound software engineering principles across the entire lifecycle during the creation of various software artifacts. Furthermore, as new approaches to software development are advanced (e.g., Model-Driven Software Development - MDSD), the investigation and transition of an engineering process into the new approaches is crucial. Model transformations are the heart of model-driven research that assists in the rapid adaptation and evolution of models at various levels of detail [22]. As models and model transformations are elevated to first-class artifacts within the software development process, they need to be analyzed, designed, implemented, tested, maintained and subject to configuration management. For models representing embedded systems that perform critical functions, the importance of correct model transformations is elevated further [8, 26]. A holistic approach to model transformation can help to assure that the transformation process is reusable and robust [3]. At the implementation level, testing is a popular research area that has obtained widespread attention [31]. Various tools and methodologies have been developed to assist in testing the implementation of a system (e.g., unit testing, mutation testing, and white/black box testing). In addition to source code, testing has been applied to other software artifacts (e.g., component systems [14] and executable models [6]). However, in current model transformation environments, there are few facilities provided for testing transformation specifications in an executable style. The result is a lack of an effective and practical mechanism for finding errors in transformation specifications. The goal of our research is to describe a disciplined approach to transformation testing, along with the development of the required tools, to assist in ensuring the correctness of model transformations.

In general, model transformation techniques can be categorized as either model-to-model transformation or model-to-code translation [5]. Model-to-model transformation translates between source and target models, which can be instances of the same or different meta-models. In a typical model-to-model transformation framework, transformation rules and application strategies are written in a special language, called the *transformation specification*, which can be either graphical [1] or textual [10]. The source model and the transformation specification are interpreted by the transformation engine to generate the target model. In a model transformation environment, assuming the model transformation engine works correctly and the source models are properly specified, the model transformation specifications are the only artifacts that need to be validated and verified. A transformation specification, like the code in an implementation, is written by humans and susceptible to errors. Additionally, a transformation specification may be reusable across similar domains. Therefore, it is essential to ensure the correctness of the transformation specification before it is applied to a collection of source models.

Model transformation testing as defined in this chapter focuses on *transformation specification testing* within the context of model-to-model transformation where source models and target models belong to the same meta-model. Specification testing involves executing a specification with the intent of finding errors. Such execution-based testing has several advantages that make it an effective method to determine whether a task (e.g., model transformation) has been correctly carried out. These advantages are: 1) the relative ease with which many of the testing activities can be performed; 2) the software artifacts being developed (e.g., model transformation specifications) can be executed in its expected environment; 3) much of the testing process can be automated [14]. Specification testing can only show the presence of errors in a specification and not their absence. However, as a more lightweight approach to specification verification and validation compared to model-checking [15, 25, 17] and other formal methods (e.g., theorem-proving), testing can be very effective in revealing such errors. This chapter describes our initial work into a testing framework that supports construction of test cases based on test specifications, execution of test cases, and examination of the produced results.

We define execution of a test case to involve the application of a deterministic transformation specification with test data (i.e., input to the test case) and a comparison of the actual results (i.e., the target model) with the expected output (i.e., the expected model), which must satisfy the intent of the transformation. If there are no differences between the actual target and expected models, it can be inferred that the model transformation is correct with respect to the given test specification. If there exist differences between the target and expected models, the transformation specification needs to be reviewed and modified. Within this context, model transformation testing has three core challenges:

1. **Automatic comparison of models:** During model transformation testing, comparison between two models (i.e., the expected model and the target model) must be performed to determine if there is a mismatch between the models. Manual comparison of models can be tedious, exceedingly time consuming, and susceptible to error. An effective approach is to compare the actual model and the expected model automatically with a high performance algorithm. Pre-existing graph matching algorithms are often too expensive for such a task [19]. A model transformation testing engine requires an efficient and applicable model comparison algorithm.
2. **Visualization of model differences:** To assist in comprehending the comparison results, comprehensive visualization techniques are needed to highlight model differences intuitively within modelling environments. For example, graphical shapes, symbols and colors can be used to indicate whether a model element is missing or redundant. Additionally, we needed to decorate these shapes, symbols and colors onto models even inside models. Finally, a navigation system is needed to support browsing model differences efficiently. Such techniques are essential to understanding the results of a model transformation testing framework.
3. **Debugging of transformation specifications:** After determining that an error exists in a model transformation, the transformation specification must be investigated in order to ascertain the cause of the error. A debugging tool for model transformations can offer support for isolating the cause of a transformation error. Of course, debuggers at the programming language level cannot be reused at the modeling level due to the semantic differences in abstraction between the artifacts of code and models. A model transformation debugger must understand the model representation, as well as possess the ability to step through individual lines of the transformation specification and display the model data intuitively within the host modeling environment.

This chapter focuses on Challenges 1 and 2 from above (i.e., the model difference and visualization problems) and introduces a framework for model transformation testing that is based on model comparison techniques. Theoretically, our approach renders models and meta-models as graphical notations such that the model comparison problem corresponds to graph comparison. The method and tools collaborate with an existing model-to-model transformation engine to provide facilities for construction of test cases, execution of test cases, comparison of the output model with the expected model, and visualization of test results. Currently, we assume test specification and test cases are generated manually by model transformation developers or testers; and the expected models used to compare with the actual results are also specified manually. We also recognize that the expected models in most modelling practices can be built based on the base models instead of from scratch so that it involves less manual effort. The automation provided by our testing framework is the execution of the tests for ensuring the correctness of the transformation specification. The chapter is structured as follows: Section 2 reviews background necessary to understand the remainder of the chapter. The importance of model mapping and difference is described in Section 3, along with an algorithm for model comparison. The core of the chapter can be found in Section 4, which presents the framework for model transformation testing. Section 5 illustrates the feasibility of the framework via a case study of model transformation. Section 6 is a discussion of related work. Finally, Section 7 provides concluding remarks and discusses future work.

2 Background

This section briefly introduces the concepts of model-driven software development and the role of model transformation. Specifically, the modelling tool and transformation engine used in this research is described.

The Model-Driven Architecture (MDA) [9] is an initiative by the Object Management Group (OMG) to define platform-independent models (PIM), which can be transformed to intermediate platform-specific models (PSM), leading to synthesis of source code and other artifacts [3]. The current OMG standards for defining PIMs and PSMs include the Meta Object Facility (MOF) and the UML. To provide a well-established foundation for transforming PIMs into PSMs, the OMG initiated a standardization process by issuing a Request for Proposal (RFP) on Query/Views/Transformations (QVT) [13]. Driven by practical needs and the OMG's request, a large number of approaches to model transformation have been proposed recently [5]. The primary approaches for model-to-model transformation include graph-based transformation [1], XSLT-based transformation [7], and specialized OCL-based transformation (e.g., the Embedded Constraint Language [11]). Based on these approaches, associated tools have been developed to facilitate rapid adaptation and evolution of models.

Over the past four years, we have developed a model transformation engine, the Constraint-Specification Aspect Weaver (C-SAW), which unites the ideas of aspect-oriented software development (AOSD) [20] with model-integrated computing (MIC) [18] to provide better modularization of model properties that are crosscutting throughout multiple layers of a model [11]. MIC has been refined over the past decade at Vanderbilt University to assist in the creation and synthesis of complex computer-based systems [18]. The Generic Modelling Environment (GME) [21] is a domain-specific modelling tool that realizes the principles of MIC. The GME provides meta-modelling capabilities that can be configured and adapted from meta-level specifications (representing the modelling paradigm) that describe the domain. The GME is generic enough to construct models of different domains and has been proven in practice on several dozen international research projects that were sponsored by industry and federal governments.

C-SAW is a model transformation engine that is integrated into the GME as a plug-in. C-SAW provides the ability to explore numerous modelling scenarios by considering crosscutting modelling concerns as aspects that can be inserted and removed rapidly from a model. Within the C-SAW infrastructure, the language used to specify model transformation rules and strategies is the Embedded Constraint Language (ECL), which is an extension of OCL [11]. ECL provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (e.g., size, forAll, exists, select). It also provides special operators to support model aggregates (e.g., models, atoms, attributes), connections (e.g., source, destination) and transformations (e.g., addModel, setAttribute, removeModel) that render access to modelling concepts within the GME.

There are two kinds of ECL specifications: a specification aspect describes the binding and parameterization of strategies to specific entities in a model, and a strategy, which specifies elements of computation and the application of specific properties to the model entities. C-SAW interprets these specifications and transforms the input source model into the output target model. An example ECL specification of a model transformation is provided in Section 5

The C-SAW web site is a repository for downloading papers, software, and several video demonstrations that illustrate model transformation with C-SAW in the GME (please see: http://www.cis.uab.edu/gray/Research/C-SAW/). In this chapter, we do not emphasize the aspect-oriented features of C-SAW; information on that topic can be found in [10, 11]. For the purpose of this chapter, we consider C-SAW as a general model transformation tool with ECL serving as the transformation specification language.

Although we use C-SAW in the example of Section 5, we believe the general process of testing model transformations can be adopted for any combination of modeling tool and transformation engine, provided that there is a mechanism for integration (i.e., a plug-in architecture, such as that provided in GME). Figure 1 shows the integration of GME, C-SAW and the testing Engine (the core testing tool described in this chapter). This framework is based on the concepts of model mapping and difference, as well as the techniques of model comparison presented in the next section.

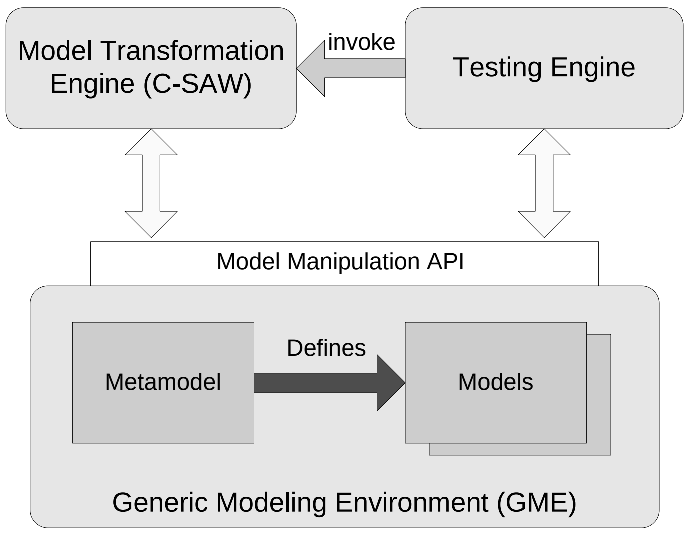

Model Transformation Engine (C-SAW)
invoke
Testing Engine
Model Manipulation API
Metamodel
Defines
Models
Generic Modeling Environment (GME)

Fig. 1. Integration of GME, C-SAW and testing Engine

3 Detecting the Differences between Models

A distinction between actual and expected results is critical in software testing. Model comparison is vital to model transformation testing in order to discover the mapping and differences between the output model and the expected model. From a mathematical viewpoint, GME models can be rendered in a graphical representation, permitting graph-theoretic operations and analysis to be performed. This section explores model difference based on graphical notations and presents an algorithm for model comparison.

3.1 Graph Representation of Models

In GME, meta-models are described in UML class diagrams and OCL constraints, which define the schema and constraint rules for models. GME models can be represented as typed and attributed multi-graphs that consist of a set of vertices and edges [1]. The following definitions are given to describe a GME model.

Vertex. A vertex is a 3-tuple (name, type, attributes), where name is the identifier of the vertex, type is the corresponding meta-modeling element for the vertex, and attributes is a set of attributes that are predefined by the meta-model.

Edge. An edge is a 4-tuple (name, type, src, dst), where name is the identifier of the edge, type is the corresponding meta-modeling element for the edge, src is the source vertex and dst is the destination vertex.

Graph. A graph consists of a set of vertices and a set of edges where the source vertex and the destination vertex of each edge belong to the set of vertices. Thus, a graph G is an ordered pair (V, E), $\forall e \in E, Src(e) \in V \wedge Dst(e) \in V$.

Elements and Features. In GME, a model can be represented as a graph. We define a model element as a vertex or an edge in a graph. A feature is any attribute of a model element.

In practice, more graphical representations are needed to describe complicated model systems (e.g., GME also supports model hierarchy and multiviews [21]). However, as a first step towards model comparison, the following discussions on model mapping and difference are based on these definitions.

3.2 Model Mapping and Difference

The comparison between two models, M1 and M2, always produces two sets: the mapping set (denoted as MS) and the difference set (denoted as DS). The mapping set contains all pairs of model elements that are mapped to each other between two models. A pair of mappings is denoted as Map $(elem^1, elem^2)$, where $elem^1$ is in M1 and $elem^2$ is in M2, which may be a pair of vertices or a pair of edges. A vertex v^1 in M1 mapping to a vertex v^2 in M2 implies matching names and types, disregarding whether their features are matching or not. An edge e^1 in M1 mapping to an edge e^2 in M2 implies matching names and types (i.e., source vertices are a mapped pair, and destination vertices are also a mapped pair). When finding the mappings between two models, the name match and type match are considered as a structural mapping to simplify the process of model comparison. This mapping is based on the syntax of a modelling language. In complicated situations, two models can be syntactically different but semantically equivalent, which is also acceptable in the testing context. However, these situations are not considered in this chapter.

The difference set is more complicated than the mapping set. The notations used to represent the differences between two models are operational terms that are considered more intuitive [2]. There are several situations that could cause two models to differ. We define DS = M2 - M1, where M2 is the actual output model, and M1 is the expected model in model transformation testing. The first differing situation occurs when some modelling elements (e.g., vertices or edges in graph representation) are in M1, but not in M2. We denote this kind of difference as New (e^1) where e^1 is in M1, but not in M2. The converse is another situation that could cause a difference (i.e., elements in M2 are missing in M1). We denote this kind of difference as Delete (e^2) where e^2 is in M2, but not in M1. These two situations occur from structural differences between the two models. A third difference can occur when all of the structural elements are the same, but a particular value of an attribute is different. We denote this difference as Change $(e^1, e^2, f, val^1, val^2)$ where e^1 is in M1 and e^2 is in M2, which are a pair of mapping vertices, f is the feature name (e.g., name or an attribute), val^1 is the value of $e^1.f$, and val^2 is the value of $e^2.f$. Thus, the difference set actually includes three sets: DS = N, D, C where N is a set that contains all the New differences, D is a set that contains all the Delete differences, and C is a set that contains all the Change differences.

3.3 Model Comparison

In GME, models can be exported and imported to an XML representation. A possible approach to the model comparison problem is to compare the XML representations of two models. However, with this approach, applications need to be developed to handle the results retrieved from the XML comparison in order to indicate the mapping and difference on models within the modeling environment. A more exact approach is to regard model comparison as graph comparison so that model elements (e.g., vertices and edges) can be compared directly via model navigation APIs provided by the underlying modeling environment.

The graph matching problem can be defined as finding the correspondences between two given graphs. However, the computational complexity of general graph matching algorithms is the major hindrance to applying them to practical applications. For example, the complexity of graph isomorphism is a major open problem and many other problems on graphs are known to be NP-hard [19]. Thus, it is necessary to loosen the constraints on graph matching to find solutions in a faster way. In model transformation testing, one of the goals is to automate model comparison with less labor-intensive integration. It is well-known that some parts of model comparison algorithms are greatly simplified by requiring that each element have a universally unique identifier (UUID) which is assigned to a newly created element and will not be changed unless it is removed [2]. To simplify our model comparison algorithm, it is necessary to enforce that every model element have a unique identifier, such that the model comparison algorithm is based on name/id matching. That is, the corresponding elements are determined when they have the same name/id, which will simplify the comparison algorithm. For example, to decide whether there is a vertex (denoted as v^2) in M2 mapped to a vertex (denoted as v^1) in M1, the algorithm first needs to find a vertex with the same name/id as v^1's and then judge whether their type and attributes are equivalent.

Figure 2 presents an algorithm to calculate the mapping and the difference between two models. It takes two models (M1: the expected model and M2: the target model) as input, and produces two sets: the mapping set (MS) and the different set (DS) that consists of three types of differences (N: the set of the New differences, D: the set of the Delete differences, and C: the set of the Change differences). When this algorithm is applied to model comparison within the testing framework, its output can be sent to the visualization component to display test results and to generate test reports.

```
Input: M1, M2
Output: MS, DS {N, D, C}

Begin
1. For each vertex $v^1$ in M1
      If there is a vertex $v^2$ in M2 mapped to $v^1$
         Mark $v^1$ and $v^2$ mapped
         Add pair ($v^1$, $v^2$) to MS
         For each attribute $f$ of $v^1$ and $v^2$,
            If the value of $v^1$ ($val^1$) does not equals to
            the value of $v^2$ ($val^2$)
               Add ($v^1$, $v^2$, $f$, $val^1$, $val^2$) to C
      Else
         Add $v^1$ to N
2. For each edge $e^1$ in M1
      If there is a edge $e^2$ in M2 mapped to $e^1$
         Mark $e^1$ and $e^2$ mapped
         Add pair ($e^1$, $e^2$) to MS
      Else
         Add $e^1$ to N
3. For those elements in M2 without mapped mark
      Add them to D
End
```

Fig. 2. A ModelComparison algorithm

4 A Framework for Model Transformation Testing

In the context of C-SAW, a model transformation is performed by interpreting the ECL transformation specification. This section describes a framework (see Figure 3) for testing model transformation specifications that assists in generating tests, running tests, and documenting tests automatically and efficiently.

There are three primary components to the testing framework: test case constructor, test engine, and test analyzer. The test case constructor consumes the test specification and then produces test cases for the to-be-tested transformation specification. The generated test cases are passed to the test engine that interacts with C-SAW and the GME to exercise the specified test cases. Within the testing engine, there is an executor and a comparator. The

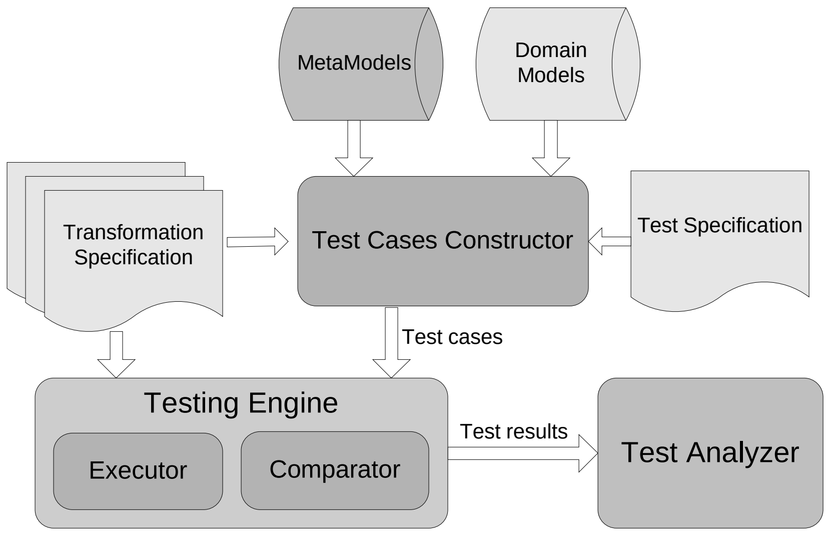


Fig. 3. A Framework for Model Transformation Testing

executor is responsible for executing the transformation on the source model and the comparator compares the target model to the expected model and collects the results of comparison. The test analyzer visualizes the results provided by the comparator and provides a capability to navigate among any differences.

4.1 Test Case Constructor

In testing a model transformation, there are different ways to construct test cases. For example, tests can be constructed according to different coverage criteria or test specifications [24]. In our testing framework, we assume tests are planned by the user and defined in a textual test specification. A test specification defines a single test case or a suite of test cases in order to satisfy the testing goal. A simple test case specification defines: a transformation specification to be tested (i.e., an ECL specification file describing a single strategy or a collection of strategies); a source model and an expected model (i.e., a specification of the expected result); as well as the criteria for determining if a test case passed successfully (such as the equivalence between the target and expected models). The test specification can be written by a user or generated from a wizard by selecting all of the necessary information directly from the browser within the GME. A test specification example is shown in the case study of Section 5.1. The test case constructor interprets the specification to retrieve the necessary information involved in a test case, such as: the to-be-tested ECL specification file, the input model, the expected model, and the output model. The test case constructor composes the specification with the input model to generate an executable test case. The work of the test case constructor can be done manually, but is more effective when performed automatically.

4.2 Test Engine

The test engine will load and exercise each test case dynamically. Figure 4 shows an overview for executing a model transformation test case. During test case execution, the source model and the transformation specification serve as input to the executor, which performs the transformation and generates a target model. After execution, the comparator takes the target model from the executor and compares it to the given expected model. The result of the comparison should be collected and passed to the test analyzer to be visualized. If the target model matches the expected model, the test is successful. If the two models do not match, the difference is highlighted to help find the errors. During the executor and comparator steps, the meta-model is used to provide the required structure and constraints for test case execution and model comparison (i.e., type information from the meta-model will be used to perform the model comparison).

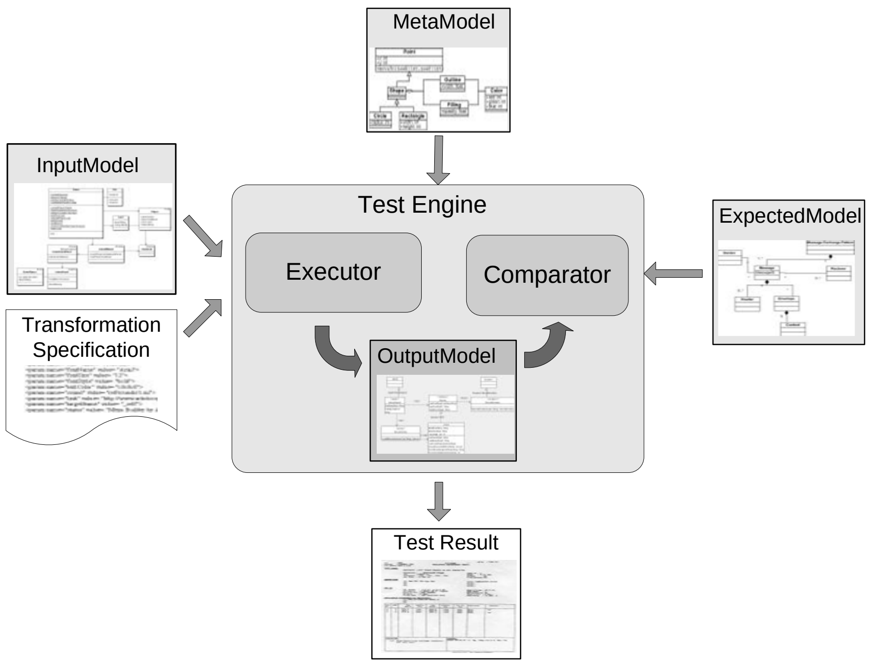


Fig. 4. Overview of a Test Case Execution

As mentioned in Section 3, there are several situations that could cause the output model and the expected model to differ. If there are differences between the target model and the expected model, it suggests that there is an error in the transformation specification. These differences are passed to an analyzer that can visualize the difference clearly in a representation that is easily comprehended by a human.

4.3 Test Analyzer

When rendered in a graphical notation, models typically have complicated internal data structures; thus, it is difficult to comprehend the differences by visual observation. It is also hard for humans to locate and navigate the differences between two models without automated assistance. To solve the visualization problem, a test analyzer is introduced to indicate the differences directly on the model diagrams and attribute panels within the modeling environment.

The test analyzer provides an intuitive interface for examining the differences between two models. It contains model panels, attribute panels, a difference navigator and a difference descriptor, as described in the following list.

1. **Output model panel and attribute panel:** displays the target output model and its attributes. Various shapes and colors are used for marking numerous types of difference: a red circle is used to mark the "New" difference, a green square is used to mark the "Delete" difference, and a "Change" difference is highlighted in blue. Generally, structural differences are shown on the diagram and feature differences are shown on the attribute panel. An example is given in Section 5.3.
2. **Expected model panel and attribute panel:** displays the expected model diagram and attributes. It illustrates the expected results for the tested model transformation.
3. **Difference navigator:** hierarchically displays the differences between two models.
4. **Difference descriptor:** textually shows the detailed description of the currently selected difference.

5 Case Study: Example Application of Model Transformation Testing

To illustrate the feasibility and utility of the transformation testing framework, this section describes a case study of testing a model transformation. This case study is performed on an experimental platform, the Embedded System Modelling Language (ESML), which is a domain-specific graphical modelling language developed for modelling real-time mission computing embedded avionics applications [10]. There are over 50 ESML component models that communicate with each other via a real-time event-channel mechanism.

The model transformation task of the case study is: 1) find all the data atoms in a component model, 2) create a log atom for each data atom, and 3) create a connection from the log atom to its corresponding data atom. The type (i.e., the "kind" attribute) of the generated log atom is set to "On Method Entry." Suppose that Figure 5 represents the initial ECL model transformation specification to accomplish the prescribed transformation of the case study. This specification defines two strategies. The "FindData" strategy specifies the search criteria to find out all the "Data" atoms. The "AddLog" strategy is executed on those data atoms identified by FindData. The AddLog strategy specifies the behavior to create the log atom for each data atom. Before this specification is applied to all component models and reused later, it is necessary to test its correctness.

```
strategy FindData()
{
   atoms() → select(a | a.kindOf() == ``Data'') → AddLog();
}

strategy AddLog()
{
   declare parentModel : model;
   declare dataAtom, logAtom : atom;
   dataAtom := self;

   parentModel := parent();

   logAtom := parentModel.addAtom(``Log'', ``LogOnMethodEntry'');
   parentModel.addAtom(``Log'', ``LogOnRead'');
   logAtom.setAttribute(``Kind'', ``On Write'');
   logAtom.setAttribute(``MethodList'', ``update'');
}
```

Fig. 5. An Untested Transformation Specification

5.1 Test Specification Definition

A test specification is used to define the test plan. In this example, there is only one test being performed, with the test specification defined in Figure 6:

```
Test test1
{
   Specification file: ‘‘C: \ ESML \ ModelComparison1 \ Strategies
                             \ addLog.spc’’
   Start Strategy: FindData
   GME Project: ‘‘C: \ ESML \ ModelCompariosn1 \ modelComparison1.mga’’
   Input model: ‘‘ComponentTypes \ DataGatheringComponentImpl’’
   Output model: ‘‘ComponentTypes \ Output1’’
   Expected model: ‘‘ComponentTypes \ Expected1’’
   Pass: Output1 == Expected1
}
```

Fig. 6. An Example Test Specification

A test is composed of a name (e.g., “test1”) and body. The test body defines the locations and identifiers of the specification file, the start strategy, a GME Project built for testing, the input source and output target models, the expected model, as well as the criteria for asserting a successful pass.

A sample input model and expected model are shown in Figure 7. The input model contains a Data atom named “numOfUsers.” According to the transformation task, a log atom should be created for this data atom and a connection should be generated from the log atom to the data atom.

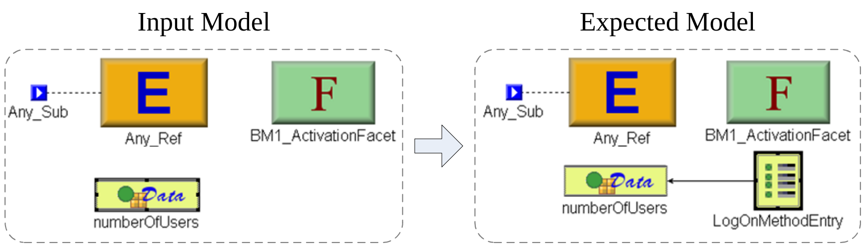


Fig. 7. The Input Model and the Expected Model

5.2 Test Case Construction

According to the test specification, the test engine generates a new ECL specification that can be executed by the executor (see Figure 8). The italicized text is the ECL code to be tested and the other ECL code (i.e., the "test1" aspect) is added by the test case generator in order to construct an executable specification that composes strategies with an aspect. In this case, strategies are the to-be-tested ECL code and the aspect specifies the input model as the source model to be transformed.

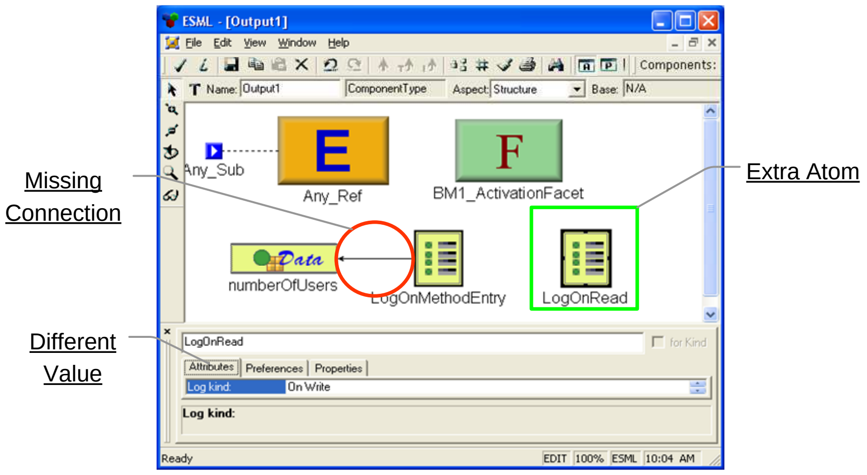
```
strategy FindData()
{
   atoms() → select(a | a.kindOf() == ''Data'') → AddLog();
}

strategy AddLog()
{
   declare parentModel : model;
   declare dataAtom, logAtom : atom;
   dataAtom := self;

   parentModel := parent();

   logAtom := parentModel.addAtom(''Log'', ''LogOnMethodEntry'');
   parentModel.addAtom(''Log'', ''LogOnRead'');
   logAtom.setAttribute(''Kind'', ''On Write'');
   logAtom.setAttribute(''MethodList'', ''update'');
}

aspect test1()
{
   rootFolder( ).findFolder(''ComponentTypes'').models().
   select(m | m.name().endWith(''DataGatheringComponentImpl''))→ FindData();
}
```

Fig. 8. The Executable Test Specification in ECL

5.3 Test Results and Difference Indication

After the test case is executed, the output target model and the expected model are sent to the comparator, which performs a comparison using the model comparison algorithm and passes the result to the test analyzer. Figure 9 shows the output model with the difference indication when compared to the expected model. There are three differences indicated on the model diagram:

- Difference 1: missing connection from LogOnMethodEntry to numberOfUsers, which is a "New" difference and marked by a red circle.
- Difference 2: an extra atom "LogOnRead" is inserted, which is a "Delete" difference and marked by a green square.
- Difference 3: the kind attribute of the LogOnMethodEntry has a different value ("On Write") from the expected value ("On Method Entry"), which is a "Change" difference and highlighted in blue.



Name: Output1 ComponentType Aspect: Structure Base: N/A

LogOnRead

Log kind: On Write

Log kind:

Fig. 9. The Output Model with Difference Indication

5.4 Correction of the Model Transformation Specification

According to the test results, it is obvious that there are three corrections that need to be made to the transformation specification. One correction to be added will create the connection between LogOnMethodEntry and numberOfUsers. The second correction is to delete one of the lines in the specification (i.e., the line that adds a LogOnRead: "parentModel.addAtom("Log", "LogOnRead")"). The third correction is to change the value of the Kind attribute from "On Write" to "On Method Entry." The modified transformation specification is shown in Figure 10 with the corrections underlined. We do not imply that Figure 10 is automated - the transformation in this figure represents the correct specification that would be required after observing the test results.

```
strategy FindData()
{
   atoms() → select(a | a.kindOf() == ‘‘Data’’) → AddLog();
}

strategy AddLog()
{
   declare parentModel : model;
   declare dataAtom, logAtom : atom;
   dataAtom := self;

   parentModel := parent();

   logAtom := parentModel.addAtom(‘‘Log’’, ‘‘LogOnMethodEntry’’);
   ~~parentModel.addAtom(‘‘Log’’, ‘‘LogOnRead’’);~~
   logAtom.setAttribute(‘‘Kind’’, ‘‘On MethodEntry’’);
   logAtom.setAttribute(‘‘MethodList’’, ‘‘update’’);

   parentModel.addConnection(‘‘AddLog’’, logAtom, dataAtom);
}
```

Fig. 10. The Corrected Transformation Specification

6 Related Work

There are a variety of techniques for validation and verification of model and model transformation (e.g., model checking [17, 25], simulation [30] and formal proof [28]). Model checking is a widely used technique for validation and verification of model properties (e.g., the Cadena model checking toolsuite [15], the SPIN Model checker [17] and the CheckVML tool [25]). However, transformation specification testing is different from model checking in its focus on the correctness of transformations that also requires engineering processes [27]. A mathematically proven-based technique for validating model transformations is proposed in [28], however, such approach requires mathematical description and analysis of models and transformation rules. Compared with these approaches, our contribution to transformation testing is a lightweight approach to finding transformation faults by executing them within the model transformation environment without the need to translate models and transformation specifications to mathematical representations. Based on an RFP issued by the OMG for a standard test instrumentation interface (TII) for executable models (see "Model-Level Testing and Debugging Interface" [12]), Eakman presents an approach to test executable platform independent models where instrumentation code is added during transformation from a PIM to a PSM [6]. This approach requires execution at the source code levels, but our contribution does not involve source code executions.

Toward realizing our vision of execution-based testing of model transformations, we need to solve the problems of model difference detection and visualization. There have been some work on model difference detection and visualization in current modelling research [2, 23]. Algorithms for detecting differences between versions for UML diagrams can be found in [23] where UML models are assumed to be stored as syntax trees in XML files or in a repository system. Several meta-model-independent algorithms regarding difference calculation between models are presented in [2], which are developed primarily based on some existing algorithms on detecting changes in structured data [4] or XML documents [29]. In the above approaches, a set of operations such as "create" and "delete" are used to represent and calculate model differences. However, these research results have not been used to construct testing tools within model transformation environments. In addition, the commonly proposed technique to visualize the model differences is coloring [23]. However, coloring alone is not sufficient in visualizing model differences in complicated modelling systems that support model hierarchy and composition. Our research involves utilizing shapes, symbols and textual descriptions to advance visualization of model differences and constructing a navigation system for browsing model differences.

7 Conclusions and Future Work

This chapter presents a testing framework for model transformation that provides the facilities for test case construction, execution of test cases, comparison of the output model with the expected model, and visualization of test results. This framework is based on the concepts of model mapping and model difference to provide a technique for model comparison. The main purpose of the framework is to automate testing of model transformation to ensure the correctness of the changes made to a source model; the focus is neither on generation of model-driven test cases for the implementation (programming language) development phase nor on Test-Driven Development of Models [16]. This initial framework is integrated within the C-SAW model transformation engine and the GME modeling environment. Although we target GME models in this chapter, we believe the general idea of transformation testing is also applicable to other modelling tools that can support integration with a plug-in architecture. To utilize the framework in other modelling tools, the test engine would need to adjust to the tool-specific model semantics in order to support model mapping and difference within the host modelling tool.

There are several opportunities to expand the work presented in this chapter. Models are often represented visually as a hierarchical graph with complex internal representations. Several modelling tools (e.g., the GME) capture the hierarchy and containment within models explicitly, whereby the modeler recursively clicks on higher-level models in order to reveal the contents of the containment. The hierarchical nature of models makes it difficult to calculate and observe visually the mapping and difference between two models. Moreover, the multiple views of models also increase the complexity of transformation testing. Other research issues that will be investigated in the near future include: study on effectiveness of this approach to detecting errors in transformation specifications, and evaluation of test adequacy and coverage in the context of model transformation test cases.

We are beginning work to address the debugging issue of Challenge 3, as mentioned in Section 1. To assist in locating the particular transformation specification error, it is necessary to investigate the concept of a debugger for a model transformation engine. This would allow the step-wise execution of a transformation to enable the viewing of properties of the transformed model as it is being changed. The testing tool suite and the debugging facility together will offer a synergistic benefit for detecting errors in a transformation specification and isolating the specific cause of the error. All of the tools will be designed to integrate seamlessly within the host modelling environment.

8 Acknowledgements

This project is supported by the DARPA Information Exploitation Office, under the Program Composition for Embedded Systems (PCES) program.

References

1. Aditya Agrawal, Gábor Karsai, and Ákos Lédeczi. An end-to-end domain-driven software development framework. In 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) - Domain-driven Track, pages 8–15, Anaheim, California, October 2003.
2. Marcus Alanen and Ivan Porres. Difference and union of models. In Proceedings of the UML 2003 Conference, volume 2863 of LNCS, pages 2–17, San Francisco, California, October 2003. Springer-Verlag.
3. Jean Bézivin, Nicolas Farcet, Jean-Marc Jézéquel, Benot Langlois, and Damien Pollet. Reflective model driven engineering. In Proceedings of UML 2003 Conference, volume 2863 of LNCS, pages 175–189, San Francisco, California, October 2003. Springer-Verlag.
4. Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 493–504, Montreal, Canada, June 1996.

5. Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In OOPSLA Workshop on Generative Techniques in the Context of Model-Driven Architecture, Anaheim, California, October 2003.
6. Gregory T. Eakman. Classification of model transformation approaches. In UML Workshop: Model Driven Architecture in the Specification, Implementation and Validation of Object-oriented Embedded Systems, Anaheim, California, October 2003.
7. J. Clark (Ed.). Xml transformations (xslt) version 1.1, w3c working draft. http://www.w3.org/TR/xslt11, December 2000.
8. Jonas Elmqvist and Simin Nadjm-Therani. Intents, upgrades and assurance in model-based development. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, Model-driven Software Development – Volume II of Research and Practice in Software Engineering. Springer, 2005.
9. David S. Frankel. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley and Sons, 2003.
10. Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An approach for supporting aspect-oriented domain modeling. In Generative Programming and Component Engineering (GPCE 2003), volume 2830 of LNCS, pages 151–168, Erfurt, Germany, September 2003. Springer-Verlag.
11. Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. Communications of the ACM, pages 87–93, October 2001.
12. Object Management Group. Model-level testing and debugging rfp. OMG Document Number : realtime/03-01-12, 2001.
13. Object Management Group. Mof 2.0 query/views/transformations rfp. OMG Document Number : ad/2002-04-10, 2002.
14. Mary Jean Harrold. Testing: A road map. In Proceedings of the Future of Software Engineering, pages 61–82, Limerick, Ireland, May 2000.
15. John Hatcliff, Xinghua Deng, Matthew B. Dwyer, Georg Jung, and Venkatesh Prasad Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In Proceedings of the 25th International Conference on Software Engineering, pages 160–173, Portland, Oregon, May 2003. IEEE Computer Society.
16. Susumu Hayashi, Pan YiBing, Masami Sato, Kenji Mori, Sul Sejeon, and Shuusuke Haruna. Test driven development of uml models with smart modeling system. In Proceedings of the UML 2004 Conference, volume 3237 of LNCS, pages 295–409, Lisbon, Portugal, October 2004. Springer-Verlag.
17. Gerard J. Holzmann. The model checker spin. IEEE Transactions on Software Engineering, Special issue on formal methods in software practice, pages 279 – 295, May 1997.
18. Gábor Karsai, Miklos Maroti, Ákos Lédeczi, Jeff Gray, and Janos Sztipanovits. Type hierarchies and composition in modeling and meta-modeling languages. IEEE Transactions on Control System Technology, special issue on Computer Automated Multi-Paradigm Modeling Modeling, pages 263–278, March 2004.
19. Samir Khuller and Balaji Raghavachari. Graph and network algorithms. ACM Computing Surveys, pages 43–45, March 1999.
20. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with aspectj. Communications of the ACM, pages 59–65, Octorber 2001.

21. Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Peter Volgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gábor Karsai. Composing domain-specific design environments. IEEE Computer, pages 44–51, November 2001.
22. Andreas Metzger. A systematic look at model transformations. In Sami Beydeda, Matthias Book, and Volker Gruhn, editors, Model-driven Software Development – Volume II of Research and Practice in Software Engineering. Springer, 2005.
23. Dirk Ohst, Michael Welle, and Udo Kelter. Differences between versions of uml diagrams. In European Software Engineering Conference/Foundations of Software Engineering, pages 227–236, Helsinki, Finland, September 2003.
24. Stephen R. Schach. Testing: Principles and practice. ACM Computing Surveys, pages 277–279, March 1996.
25. A. Schmidt and D. Varró. Checkvml: A tool for model checking visual modeling languages. In Proceedings of the UML 2003 Conference, volume 2863 of LNCS, pages 92–95, San Francisco, California, October 2003. Springer-Verlag.
26. Janos Sztipanovits. Generative programming for embedded systems. In Generative Programming and Component Engineering (GPCE), volume 2487 of LNCS, pages 32–49, Pittsburgh, Pennsylvania, October 2002. Springer-Verlag.
27. D. Varró and A. Pataricza. Generic and meta-transformations for model transformation engineering. In Proceedings of the UML 2004 Conference, volume 3237 of LNCS, pages 290–304, Lisbon, Portugal, October 2004. Springer-Verlag.
28. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. Science of Computer Programming, pages 205–227, 2002.
29. Yuan Wang, David J. DeWitt, and Jin-Yi Cai. X -diff: An effective change detection algorithm for xml documents. In Proceedings of the 19th International Conference on Data Engineering, pages 519–530, Bangalore, India, March 2003.
30. L. Yilmaz. Verification and validation: automated object-flow testing of dynamic process interaction models. In Proceedings of the 33rd conference on Winter simulation, pages 586–594, Arlington, Virginia, 2001.
31. Hong Zhu, Patrick Hall, and John May. Software unit test coverage and adequacy. ACM Computing Surveys, pages 367–427, December 1997.