Barriers to Systematic Model Transformation Testing

Benoit Baudry[1], Sudipto Ghosh[2], Franck Fleurey[3], Robert France[2], Yves Le Traon[4], Jean-Marie Mottu[5]

[1] INRIA, Campus de Beaulieu, 35042 Rennes, France

bbaudry@irisa.fr

[2] Colorado State University, Fort Collins, CO, USA

{france, ghosh}@cs.colostate.edu

[3] SINTEF, Oslo, Norway

Franck.Fleurey@sintef.no

[4] Telecom Bretagne, 2 rue de la Châtaigneraie, CS 17607, 35576 Cesson Sévigné, France

yves.letraon@enst-bretagne.fr

[5] Université de Rennes 1, Campus de Beaulieu, 35042 Rennes, France

jmottu@irisa.fr

1 Introduction

Model Driven Engineering (MDE) techniques [1] support extensive use of models in order to manage the increasing complexity of software systems. Appropriate abstractions of software system elements can ease reasoning and understanding and thus limit the risk of errors in large systems. Automatic model transformations play a critical role in MDE since they automate complex, tedious, error-prone, and recurrent software development tasks. Airbus uses automatic code synthesis from SCADE models to generate the code for embedded controllers in the Airbus A380. Commercial tools for model transformations exist. Objecteering and Together from Borland are tools that can automatically add design patterns in a UML class model, Esterel Technologies have a tool for automatic code synthesis for safety critical systems.

Other examples of transformations are refinement of a design model by adding details pertaining to a particular target platform, refactoring a model by changing its structure to enhance design quality, or

reverse engineering code to obtain an abstract model. These software development tasks are critical and thus the model transformations that automate them must be validated.

A fault in a transformation can introduce a fault in the transformed model, which if undetected and not removed, can propagate to other models in successive development steps. As a fault propagates further, it becomes more difficult to detect and isolate. Since model transformations are meant to be reused, faults present in them may result in many faulty models.

Model transformations constitute a class of programs with unique characteristics that make testing them challenging. The complexity of input and output data, lack of model management tools, and the heterogeneity of transformation languages pose special problems to testers of transformations. In this paper we identify current model transformation characteristics that contribute to the difficulty of systematically testing transformations. We present promising solutions and propose possible ways to overcome these barriers.

2 Testing Transformations: A Small Example

A simple example of a model transformation consists of *flattening* a state machine. Figure 1 illustrates a transformation that takes a hierarchical state machine as an input and produces a flattened state machine as an output. The output is semantically equivalent to the input but all hierarchical states are removed.

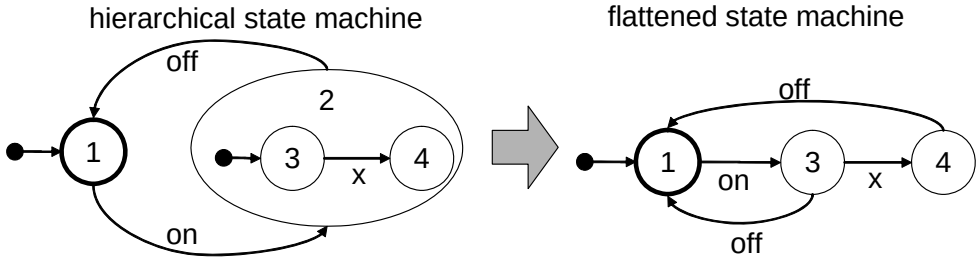

Figure 1 - Model Transformation Example: Flattening a Hierarchical State Machine.

The sets of possible input and output models for a transformation are each described by a metamodel. A metamodel is a set of classes, relationships, and multiplicities that define the concepts and the structure of a modelling language. A metamodel can be thought of as a grammar for a language. A model transformation can thus be viewed as a *grammarware* program, that is a program

whose input domain is defined by a grammar (*e.g.*, compilers, static analyzers, and reverse engineering tools).

Figure 2 shows a metamodel for the structure of hierarchical state machines. A hierarchical state machine contains states that have a number of incoming and outgoing transitions. The metamodel distinguishes between two types of states, simple states that can be initial or final states, and composite states that can contain composite or simple states. This metamodel is a formal description of the language of statecharts.

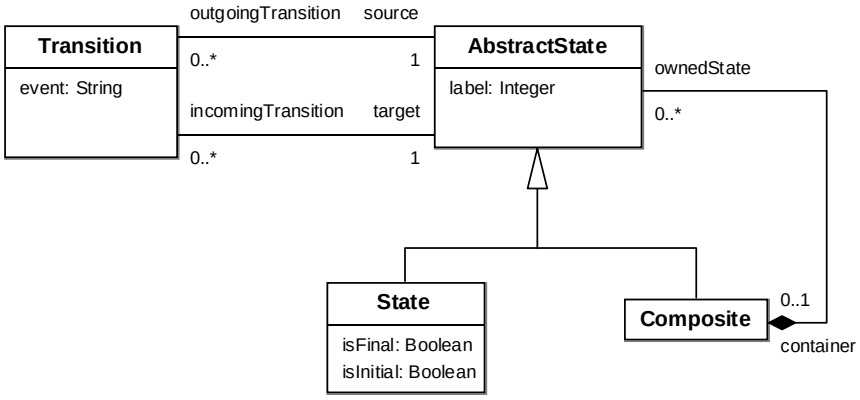

Figure 2 - A Hierarchical State Machine Metamodel.

In addition to describing metamodels with classes, attributes, and associations, it is usually necessary to define constraints that relate the concepts in the metamodel more precisely. For example, the metamodel shown in Figure 2 does not constrain a composite state to contain only one initial state. This constraint must be added to the metamodel but it cannot be defined using modelling languages such as EMOF. The Object Constraint Language (OCL) is commonly used to define these additional constraints. The constraint on the composite state could be written as follows:

context Composite

inv : self.ownedState → select(AbstractState as | as.oclIsTypeOf(State)) → select(AbstractState s | s.oclAsType(State).isInitial) → size()=1

To test the transformation shown in Figure 1, we must perform the following activities:

1 *Generate test data*: We need to generate input models that conform to the input metamodel of the transformation. The input models that are generated as inputs for a transformation are called *test models*. In our example, this requires generating several state machines that conform to the metamodel of Figure 2. Test models are manually or automatically generated in the form of graphs of metamodel instances. Figure 3 displays one test model as a graph of instances of the hierarchical state machine metamodel classes shown in Figure 2. Other test models may also be needed, to satisfy test adequacy criteria, such as a state machine with two hierarchical states or one with no hierarchical state.

2 *Define test adequacy criteria*: Since it is clearly not possible to test a transformation with all possible input models, we must define criteria to efficiently and effectively select test models. Test adequacy criteria drive the selection of a subset of test models that will be sufficient for testing. This helps reduce the time and effort spent on testing. In our example, a test criterion for the flattening transformation could require that each class of the metamodel is instantiated in at least one test model. No well-defined criteria exist for model transformation testing. This challenge must be tackled to make testing practical and systematic.

3 *Construct an oracle*: For software testing, the oracle determines if the result of a test case is correct. There are several ways to construct an oracle for a model transformation. If the expected model is available (e.g., from previous regression tests), the oracle can compare the output model with the expected model. If the expected model is not available (as is usually the case), a partial oracle that checks expected properties of the output model should be constructed. For example, such an oracle can check that the output model contains no composite states and has the same number of simple states as the input model.

We now analyze three major characteristics of MDE based model transformation that make these testing activities challenging.

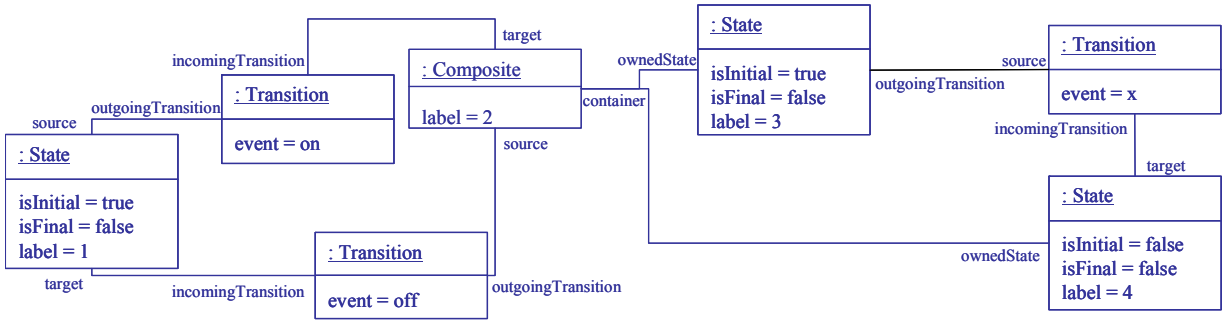

Figure 3 - Test Model for the Flattening Transformation.

3 Characteristics of Model Transformations that Impact Testing

a Complex Input and Output Data

Model transformations manipulate data of a complex nature. The input and output models are graphs of objects that are often large. Sometimes, the input or output models contain multiple views (e.g., UML class diagram and sequence diagram views), and thus consistency of views manipulated by transformations becomes a concern.

The structure of the graphs is constrained by a metamodel and the objects in the graphs are instances of the classes defined in the metamodel. The metamodels can themselves be large and complex structures. Moreover, additional constraints can be expressed in the metamodel, usually with the OCL. This increases the complexity of the metamodels; OCL is a rich language with which it is possible to define complex constraints relating a large number of elements in the metamodel.

The complexity of the data manipulated by a transformation affects the generation of test models. Manual test generation is error-prone because of the large number of metamodel instances that must be created, and the relationships and attribute values that must be set. Automatic test data generation is a complex constraint solving problem because it requires synthesizing a graph that satisfies a large set of multiplicity and OCL constraints, and test adequacy criteria. The main challenge for automatic resolution of complex constraints is handling time and memory when exploring very large solution spaces.

Since the metamodel completely describes the input domain of a transformation, it provides a basis for defining test adequacy criteria. It is possible to define a large number of criteria, such as instantiate

all the meta-classes, combine different values for the properties, and combine instances of different meta-classes. However, lack of historical data on the types of errors typically found in transformations makes it difficult to determine the effectiveness of these criteria and the fault models they can target.

The complexity of the output data complicates the oracle problem. It is difficult to manually or automatically build the expected test result. When the expected output model is available, the oracle needs to compare two models, i.e., two graphs of objects. In this case, the oracle problem is as complex as the graph isomorphism problem, which is NP-complete. If the oracle is specified by listing expected properties of the output model, the construction of this oracle is complicated by the complexity of the output metamodel that describes the output model. The tester must consider numerous concepts and relationships to define the expected properties of the output model.

b Model Management Environments

MDE is currently in the research stage and thus development environments lack adequate support for model manipulation [2]. For testing model transformations, support is needed for building, editing, visualizing, and analyzing models.

The construction of models involves either writing a program that builds the meta-class instances and sets all the properties, or using model editors generated from a metamodel (e.g., the default tree editor generated by EMF) to manually build the instances and set values for all attributes and references. Writing a program is error-prone. Using a generated editor instead of a tailor-made editor for a language is tedious because they do not provide language-specific icons, dialog boxes for setting attributes values or assistance for checking the completeness of the model (e.g., all attributes have been assigned a value). This makes the manual definition of test models difficult and error-prone.

Visualizing output models is difficult because graphical editors (e.g., for UML or domain-specific languages) often do not provide adequate support for layout of diagrams that are produced by a transformation or exported from another tool. A confusing layout complicates manual analysis of the model, and the visual comparison of two graphical representations. A possible solution is to query the output model and check some properties using OCL analyzers.

For regression testing, testers need to compare the output models produced by two versions of the transformation because a test model that was used in testing a previous version of the transformation should produce the same output model. Thus, we need sophisticated model comparison tools.

c Heterogeneity of Transformation Languages and Techniques

The OMG has defined a model transformation standard called QVT. However, there exist a large number of model transformation languages and techniques. Transformations can be implemented with general purpose programming languages (e.g., Java) or languages dedicated to model transformations (e.g., Query/View/Transformation – QVT)). The MTIP workshop [3] organized at MoDELS'05 was concerned with developing "an increased understanding of the relative merits of different model transformation techniques and approaches". The workshop organizers specified several model transformations and asked the authors to implement them. Eight papers presented 13 different techniques, which were divided into three categories**:** graph transformation related approaches, declarative and rule based approaches, imperative and related approaches. Moreover, there are several tool-specific model transformation languages, such as Objecteering, MetaEdit+, and XMF-Mosaic. The approaches presented in the workshop reflected the diversity of existing transformation languages and techniques.

Since it is impossible to know if any one of the many techniques will fit all the needs for model transformations, the testing techniques need to take this diversity into account. The diversity has a strong impact on the definition and the selection of effective white-box test adequacy criteria. We cannot choose one language as a reference and develop test criteria that are based on language elements.

4 Promising approaches to overcoming the barriers to model transformation testing

a Complex input and output data

The complexity of constraints that define the input domain is the main challenge for automatic test model generation. A possible solution to address this problem is a constructive approach where models are built first and the constraints are checked afterwards. Following this idea, Brottier et al. [4]

consider only the class diagram definition of the input metamodel to generate objects and assemble them according to specific criteria in order to build complete models. Ehrig et al. [5] analyze the metamodel to generate rules that create instances of all non-abstract classes and links between the instances. The major limitation of these approaches is that a large number of generated models do not satisfy the complete set of constraints and thus the transformation cannot process them for testing.

Another approach is to use SAT solvers to deal with a larger amount of constraints and generate instances that satisfy the constraints. This declarative approach reduces the burden on the test generation tool but in turn has the limitations of current automatic constraint solvers. To tackle the limitation of constraint solvers, a usual approach is to study heuristics and meta-heuristics to prune the search for a solution [6].

The definition of an oracle is challenging because even the simplest expected result can be complex and thus the description of this result can be error-prone. One solution that can be applied when the model transformation generates an executable model is to test the output model directly. Another approach is to have a partial oracle that checks only specific properties of the output instead of checking the result completely. This approach, similar to the JUnit approach for Java programs, has been studied by Solberg et al. [7] and Mottu et al. [8]. Solberg et al. propose patterns to express pre- and post-conditions for the transformation. These patterns are templates that describe expected features of input and output models and can be viewed as a specialization of the source and target metamodels. Mottu et al. propose to use Design by Contract™ when building a model transformation and illustrates how the contracts can be used as an oracle for model transformation testing.

We need to develop techniques for the precise definition of requirements for model transformations. Currently, the requirements tend to be very informal and cannot be processed by tools to automatically generate the expected result. Researchers in the requirements engineering community are currently exploring techniques to automatically extract logical formulae from natural languages. Another solution is to ask the modellers to express the expected behaviour as rules that could then be interpreted by an oracle.

b Model Management Environments

A number of ongoing work around model management, both in industry and academia, could be integrated or adapted for testing model transformations.

Work on model differencing can be used to develop an oracle that compares the model produced after execution of a test case with an expected model. As mentioned earlier, this can be very useful in regression testing of transformations. The EMFCompare tool, based on the algorithm proposed by Xing et al. [9], is now available in the Eclipse framework. It detects matches and differences between two models, based on the similarity of their types, their names, and the values of their attributes and relations.

Versioning of models can greatly benefit testing. It can be used to define an oracle that compares models and detects conflicts. These conflicts can in turn be used to track the error back to its source. Also, versioning of input and output metamodels could be used for regression testing: if we know the parts of the metamodels that evolve, we could detect the test models that are still relevant and the ones that need to evolve. CVS Model is an open source initiative that proposes a tool for versioning of models. The conflict detection mechanisms are based on the work by Reiter et al. [10].

c Heterogeneity of Transformation Languages and Techniques

The heterogeneity of model transformation languages is particularly challenging for the definition of test criteria. There are two possible ways to tackle this issue: either have specific criteria and associated test generation techniques for each particular language or have black box techniques that ignore the actual language used for the transformation.

Fleurey et al. [11] proposes black-box test adequacy criteria to evaluate the quality of models used as test models. The benefit of this approach is that it can be used with any transformation language. However, the criteria are based solely on the ability of test models to cover the structures defined by the input metamodel of a transformation. Although the criteria provide an initial measure of the quality of test models, this approach should be augmented to take into account the intent of the transformation.

A white-box approach can also be used to define test criteria. For example, Küster et al. [12] define a template language to generate test models based on the structure of the rules used to implement the transformation. The drawback of this approach is that it is tightly coupled to the transformation language and would need to be adapted or completely redefined for another transformation language. On the other hand, a white-box approach must cover the individual transformation steps that make up a transformation. This forces the tester to generate test models that exercise each step. Thus, the criteria are more detailed and effective than black-box criteria that do not capture the mechanics of a transformation.

5 Conclusion

We need to take the MDE-specific challenges into account when developing testing solutions for model transformations. We identified some of the major challenges in this paper.

Promising solutions to some of the testing problems exist, but more work is needed. In addition to the development of new and efficient solutions, there is a need for a benchmark of realistic models and model transformations in the model transformation community. This is needed to experimentally validate and compare solutions to the complex engineering issues associated with model transformation. We discussed the issues for testing, but there are other open problems related to the evolution, maintenance, debugging, and design of model transformations.

References

1. Schmidt, D.C., *Model-Driven Engineering.* IEEE Computer, 2006. **39**(2): p. 25 - 31.
2. France, R. and B. Rumpe, *Model-Driven Development of Complex Software: A Research Roadmap*, in *Future of Software Engineering 2007*, L. Briand and A. Wolf, Editors. 2007, IEEE - CS Press.
3. Bezivin, J., B. Rumpe, A. Schurr, and L. Tratt, *Model Transformations in Practice Workshop*. 2005: Supplemental Proceedings of the MoDELS'05 conference. p. 120 - 127.
4. Brottier, E., F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. *Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool*. in *ISSRE'06 (Int. Symposium on Software Reliability Engineering)*. 2006. Raleigh, NC, USA.

5. Ehrig, K., J.M. Küster, G. Taentzer, and J. Winkelmann. *Generating Instance Models from Meta Models*. in *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*. 2006. Bologna, Italy.

6. Korf, R.E., *Space-efficient search algorithms.* ACM Computing Surveys, 1995 **27**(3): p. 337 - 339.

7. Solberg, A., R. Reddy, D. Simmonds, R. France, and S. Ghosh, *Developing Service Oriented Systems Using an Aspect-Oriented Model Driven Framework.* International Journal of Cooperative Information Systems, 2006.

8. Mottu, J.-M., B. Baudry, and Y. Le Traon. *Reusable MDA Components: A Testing-for-Trust Approach*. in *MoDELS'06*. 2006. Genova, Italy.

9. Xing, Z. and E. Stroulia. *UMLDiff: An Algorithm for Object-Oriented Design Differencing*. in *Automated Software Engineering (ASE'05)*. 2005. USA.

10. Reiter, T., K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. *Models in Conflict - Detection of Semantic Conflicts in Model-based Development*. in *proceedings of 3rd international workshop on model-driven enterprise information systems (MDAIS-2007), in conjunction with ICEIS'07*. 2007. Madeira, Portugal.

11. Fleurey, F., B. Baudry, P.-A. Muller, and Y. Le Traon, *Towards Dependable Model Transformations: Qualifying Input Test Data.* Software and Systems Modeling, 2007.

12. Küster, J.M. and M. Abd-El-Razik. *Validation of Model Transformations - First Experiences using a White Box Approach*. in *MoDeVa'06 (Model Design and Validation Workshop associated to MoDELS'06)*. 2006. Genova, Italy.