Model Transformation Testing Challenges

Benoit Baudry[1], Trung Dinh-Trong [2], Jean-Marie Mottu[1], Devon Simmonds[2], Robert France[2], Sudipto Ghosh[2], Franck Fleurey[1], Yves Le Traon[3]

[1] IRISA, Campus Beaulieu, 35042 Rennes Cedex, Rennes, France.
{bbaudry, jean-marie.mottu, ffleurey}@irisa.fr
[2]Colorado State University, 601 Howes Street, Fort Collins, CO, USA
{france, ghosh, trungdt, simmonds}@cs.colostate.edu
[3]France Télécom R&D/MAPS/EXA, 2 avenue Pierre Marzin, 22307 Lannion Cedex, France
yves.letraon@francetelecom.com

Abstract. Model transformations play a critical role in Model Driven Engineering, and thus rigorous techniques for testing model transformations are needed. This paper identifies and discusses important issues that must be tackled to define sound and practical techniques for testing transformations.

1 Introduction

Model-Driven Development (MDD) aims to provide automated support for creating and transforming software models. Effective support for model transformations is thus key to successful realization of MDD in practice. In particular, efficient techniques and tools for validating model transformations are needed. In this paper, we focus on the issues and challenges that must be addressed when developing efficient techniques for testing model transformations.

On the surface it may seem that testing model transformations is no more challenging than testing code. Like code testing, input data have to be selected and an oracle function must be defined to check the correctness of the result. However, the nature of the input and output data manipulated by transformations makes these activities more complex. Indeed, transformations manipulate models, which are very complex data structure. This makes the problem of test data generation and oracle definition very difficult in the case of model transformations.

Figure 1 shows the generic transformation framework that provides the context for discussion in this paper. A model transformation manipulates concepts that are specified in the source and target metamodels (which can be different). These metamodels describe the static structure of the models that are manipulated by the transformation. In the transformations we have developed, these metamodels conform to the MOF [1]. In some cases, these metamodels should be augmented with constraints (expressed in OCL for example) that more precisely constrain the structure of models that are manipulated by the transformation. In the case of the UML metamodel, these constraints are the well-formedness rules.

A transformation takes an input model that conforms to the source metamodel and produces an output model, which conforms to the target metamodel. In the following, we consider transformations that take a single input model and produce a single model. However, we will discuss some specific issues that can arise when

manipulating several models. We will also discuss the role contracts can play in validating model transformations.

The precondition shown in Figure 1 further constrains (in addition to the source metamodel and its associated constraints) the type of models that can be input to the transformation. The post condition specifies expected properties on the output model as well as properties that link the input and the output models. These additional constraints are of the same nature as the constraints on the metamodels, but they are specific to the transformation.

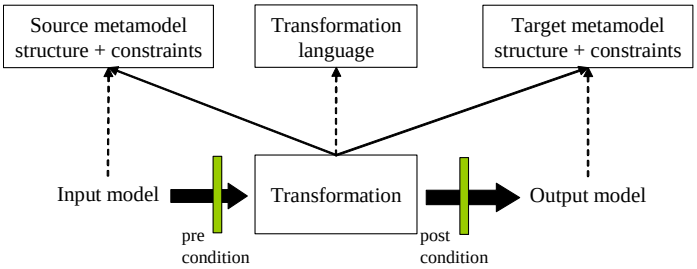

Figure 1 - General framework for model transformation

This paper discusses some of the challenges related to testing transformations based on our collective experience with specifying, implementing and testing transformations. Moreover, the paper highlights several issues related to developing efficient techniques for testing model transformations. We discuss the different solutions we have developed and applied to deal with test data generation and oracle function definition. The first issue for test data is that they must conform to the structure described by the metamodel, and they should cover all the constructs described by this metamodel. Moreover, the test data must conform to the constraints on the metamodel as well as the precondition. With respect to oracle definition, we will discuss the use of contracts and we will also present some examples that illustrate particular approaches to the oracle definition problem.

In section 2, we discuss some generic approaches to test generation and oracle definition that we have studied. We discuss the limitations of these approaches as well as issues that arose when developing and applying the approaches. In section 3, we go into the details of two particular transformations and show how we leverage the specificities of the transformations during testing.

2 Generic Approaches for Model Transformation Testing

2.1 Test data generation

To test a model transformation a tester has to provide input models that conform to the source metamodel. In the following, we call these test models. The intuition for a set of test models to be complete is that each class of the source metamodel is instantiated at least once in one model of the set. Moreover, we would like the properties of the classes (attributes and the multiplicities on associations) to take several representative values in the models. For example, let us consider the statechart

metamodel of Figure 2 as the source metamodel for a model transformation. Considering the values of the boolean attribute of the class STATE, we would like to have at least two instances of the STATE class for testing: one for which `isFinal` is true and another one for which it is false. In the same way, considering the `incomingTransition` association that has a 0..* multiplicity, we would like to have three instances: one with 0, one with 1 and one with more than one. More generally, by adapting category-partition testing [2] it is possible to define ranges of values for each property (as detailed in [3])and check that there is at least one instance of each property that has one value in each range.

Beyond representative values for the properties of the metamodel, we would like to define more complex expected properties for test models. The representative values and multiplicities should be combined to build relevant test models. For instance, there should be a test model which has an instance of STATE for which `isInitial` is true and for which the multiplicity for `outgoingTransitions` is greater than one.

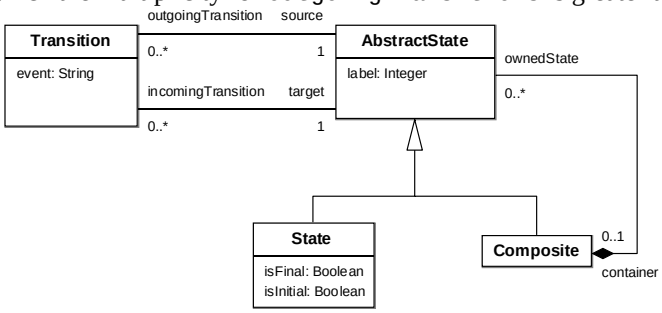

Figure 2 – Simple Composite State Machine Metamodel

To capture the notions of representative value, we adapt the category-partition testing to define ranges of values for each property of the source metamodel. The properties should then be instantiated at least once with a value in each range. To capture the idea of combination of properties, we define the notions of *object* and *model fragment*. These notions allow us defining constraints on the objects and models that should be present in a set of test models for one particular transformation.

Based on these notions, we have defined a set of test criteria that allow evaluating the quality of a set of models for testing with respect to the source metamodel coverage. We have defined a metamodel that captures the notions to define test criteria. This metamodel is also the basis to build a framework that checks the adequacy of a set of test models to the criteria. We do not detail these criteria here, and focus more on the first feedback we have on using these criteria for model transformation testing.

2.1.1 Interactive approach vs. automatic generation

We have built a tool that automatically generates a set of test models that satisfy the constraints defined by each criterion. The first issue when implementing this tool is the large number of strategies that can be used to build these models. The following are three important variation points for the generation algorithm:

- Choose the values for the properties in a particular range.

- Choose when objects go together in one model or when a new model should be generated.
- Choose how to build new objects to complete the models to make them conform to the metamodel.

A large number of case studies should be conducted to evaluate the different strategies. There might not be one strategy that is the best in every case, empirical studies should establish a classification among the strategies.

The second issue when automatically generating models is that the obtained models are difficult to interpret by a human tester. This has several consequences. First, if the tester does not understand the input model, it is difficult to establish an expected result or even properties on the expected output for this input. Second, it makes fault localization difficult. If the test case fails a generated input model, the tester must understand the test model to understand which part of the transformation have been executed when running with this input and thus understand in which part of the transformation the error might be. Third, when a developer writes a transformation, most of the time she has a set of examples (input models) in mind, and it might easier for the tester to start from these models and complete them to make them verify the criteria.

This led us to propose an interactive approach more than a completely automatic approach. It consists in assuming that the tester provides an initial set of test models manually generated. It is possible to automatically check whether these models satisfy the criteria or not. In the case the models are not complete the tool provides a list of missing elements the tester should add for the set of test models to cover the source metamodel.

2.1.2 Benefits and limitations

The first benefit of this approach is to offer systematic criteria to evaluate the adequacy of a set of models for testing a model transformation. These criteria are black-box criteria and are based on the idea that test models should cover all the structure of the source metamodel. We have defined a metamodel that captures the important notions to define different test criteria that define constraints more or less complex on the test models. Moreover this metamodel allows us to define a framework to can automatically check the adequacy of a set of test models and that allows us to conduct experiment on the generation of test models.

However, this approach is based on our preliminary work. An important limitation is that the criteria are based only on the structure of the source metamodel and do not take into account the constraints associated to the source metamodel or the precondition of the transformation that define constrains on the input models. This results in defining criteria that might impossible to satisfy completely. For example, to satisfy a particular criterion it might necessary to generate a statechart with no initial or no final state, which is not possible. If a constraint is explicated that specifies that a statechart should have one initial state and one final state, we would like the criteria to take into account. Of course, in the case of an interactive approach, if the framework tells the tester she should generate such a statechart, she can simply ignore this constraint based on her knowledge of the additional constraints.

More generally, we would like the criteria to take into account additional knowledge about the source metamodel and about the particular transformation that is

being tested. Sometimeq this knowledge is not formalized with any constraint language, it is simply expertise for the developer. In that case, it is impossible to consider it when specifying the criteria. On the other hand, if this knowledge is expressed using OCL for example, we would like the criteria to take into account. Future work should study to what extent such constraints can be taken into account when specifying the test criteria.

2.2 Oracle

Several researchers have studied the use of contracts as a partial oracle function in object oriented system [4, 5]. Here, we discuss how this approach can be adapted to define an oracle for a model transformation, and what are the issues concerning this adaptation.

2.2.1 Different types of contracts

The first issue is that the notion of contracts is not well defined in the context of model transformation. In this respect, classifying the different categories of contracts in a MDE context can be useful. Here, we propose a preliminary classification consisting of three different types of contract:

- Contracts in the specification of the transformation
- Contract applied to the output model
- Contract of the transformation

Contract in the specification of the transformation: This type of contract is assumed in Figure 1. The transformation is specified with contracts defining pre and post conditions, and transformation invariants (the idea is developed in [6]). The pre-conditions constrain the test models. These constraints can be used for test model generation. The invariants can be exploited when developing an oracle because they must be true all along the transformation, including when it ends and returns its output model. Finally the post-conditions express expected properties on the output model and properties which link input and output models. This last type of contract is the most convenient for the oracle since it allows the evaluation of the result.

Contract of the transformation: When the model transformation is implemented, it is usually decomposed into several smaller transformations or sub-transformations. These can be methods if the language is object-oriented or a set of rules in the case of a declarative language. In both cases, it is possible to define pre and post conditions for the sub-transformations and invariants. These assertions specify when a sub-transformation can be executed and the expected effect of this step in the transformation. These assertions are very similar to a design-by contract approach for object-oriented systems. They are useful for unit tests, in the sense that they specify the input domain of the unit under test (one sub-transformation) as well as a partial oracle in the form of the post condition. They are also very useful to help fault localization since, if a contract is violated at runtime, the tester knows the error has to be before that contract in the execution flow. This is studied for object oriented systems in [4].

Contract applied to the output model: These are different from the transformation's post condition in the sense that they do not relate to the input of the transformation and express expected properties only on the output. We distinguish

three levels of contract for the output. The first level is the conformance of the output model to the target metamodel. This first level should be checked by the transformation environment. For example, writing transformations in the Eclipse Model Framework (EMF) allows carry out this check. The second level concerns the constraints associated to the target metamodel. For example in UML a contract can check that two classes in the same package do not have the same name. Finally, there should be specific properties that the output of the transformation under test should have. This last level can be part of the specification of the transformation discussed above.

An important challenge for a trustable transformation process is to be able to express these contracts at the right level of abstraction and propose methodologies to assist the writing of such contracts. Concerning testing, all these types of contracts are useful for defining the oracle since they all express properties on the result of the transformation at different levels of detail.

2.2.2 Using OCL to express contracts

The OCL seems to offer a good support to express contracts for a model transformation. However, several issues have to be considered when using OCL in this purpose. First, as discussed in [6], the OCL might not perfectly appropriate to write contracts. Second, it can be practically difficult to write a constraint that links the input and the output models. For example, this can consist in checking that the elements from the input model are still in the output model. The issue is the following: an OCL expression has a unique `context`, which means that the constraint can concern only one metamodel. To solve this problem it is necessary to create a third metamodel, like the one presented Figure 3 that relates the source and target metamodels. In the context of this metamodel, it is possible to write a contract that relates input and output models. For example, considering the UML2RDMS transformation we can express that all the persistent classes should be transformed in tables and each one's attribute should be transformed in columns.

```
context UmlToRdbms inv:
self.modelUml.elements
->select(e| e.oclIsTypeOf(Class)and
            e.stereotype->exists(s|s.name='persistent'))
->collect(ec|ec.oclAsType(Class))
->forAll(cp|self.database.tables
            ->one (t|   t.name=cp.name and
                  cp.feature->select(f|f.oclIsTypeOf(Attribute))
                     ->collect(fa|fa.oclAsType(Attribute))
                  ->forAll (a|t.columns->one (tc|tc.name=a.name and
                                              tc.type=a.type.name
))))
```

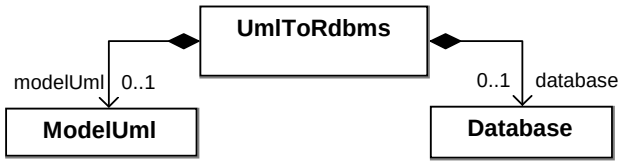

Figure 3- Metamodel used to check the constraints with OCL

To conclude this section, important issues for model transformation testing have been identified. Some of these issues are related to the design and the specification of

a model transformation. The presented techniques also have inherent limits that are due to the fact that they are very generic and thus can not leverage specific knowledge from specific transformations. In the following section we introduce two specific case studies and show how specific knowledge of the transformation can help improve the testing process.

3 Specific Approaches for Testing

This section introduces two case studies where model transformations were developed and adapted solutions for testing have been used leveraging specificities of the transformations. In the first case, the transformation generates an executable model that can thus be tested. In the second case, we discuss the use of patterns to specify the transformation and how this design approach can help for testing.

3.1 Testing the output of the transformation

In our model testing approach [7], we transform UML designs under test (DUT) into Java programs. The transformation is automated using a prototype tool called UMLAnT, and that is implemented as an Eclipse plug-in. Beside transformation mechanism, UMLAnT also provides a framework that allows developers to use the generated program to animate the execution of the DUT. A DUT consists of (1) a class diagram that captures the structural aspect of the system and (2) a set of activity diagrams that capture the behavioural aspect of the system. OCL is used in the class diagram to specify class invariants and operation pre- and post-conditions. Activity diagrams are specified using JAL [8], a Java-like action language that represents action semantics described in UML 2.0 specification. Each activity diagram describes sequence of actions in one operation.

The outputs of the transformation are executable Java programs that animate the execution of behaviours modelled in the DUT. As testers step through operations during execution of the models, the views are updated. The animated object diagrams show the creation and deletion of objects and links, as well as the modification of attribute values. The animated sequence diagrams show the messages exchanged between objects during execution. If the transformation from DUT to Java is correct, then the animation will visualize exactly the execution of the design under test.

We assessed the correctness of our UML to Java transformation approach using testing approaches. We used UMLAnT to generate executable Java code from a set of UML models that contain class diagrams and JAL segments. We carefully reviewed the input models before the transformation to ensure that they were syntactically correct and that the behaviour described in the JAL segments was the intended one. We then checked if the generated programs exhibited correct behaviour to assess the correctness of the transformation.

The input models were built so that they contained every type of constructs in the input metamodel. In our case, the input metamodel is the subset of UML metamodel that describes concepts in class diagram views and action semantic views. For example, input activity diagrams contain all primitive actions and control structure

that are supported by our transformation approach. Furthermore, our input models also cover several combinations of the metamodel constructs.

Concerning the oracle function, we leveraged the fact that the outputs of our transformation are executable. Therefore, we validated the output Java program by testing them. We generated test cases based on the DUT using a set of test adequacy criteria described in [9]. We then executed the programs with the generated input. In some cases, the generated programs could not be compiled, indicating that there were errors in the output. In some other cases, the programs could be compiled, but some of the test cases we ran against the generated Java programs failed, also meaning that there were errors in the programs. Given that the input models were known to be correct, an error in the output meant that there was an error in the transformation. Using this technique we were able to detect and remove a number of errors in UMLAnT. This increased our confidence in the prototype implementation.

Generally, testing can be used to validate the correctness of output models in any transformation which output is executable. This practice is widely used among compiler community. The technique requires a mechanism to obtain a set of correct input models that contains instances of all input metamodels elements. From these inputs, it is possible to derive an oracle (test cases or properties) that we will run against the output. Given that the research in model driven development will probably result in many new model transformation techniques, it is desirable to develop a set of test criteria to select models that can be used for testing transformation techniques. It will also help if researchers and practitioners could develop a set of benchmark UML models for empirical studies on the correctness and scalability of proposed techniques.

3.2 Pattern-based approach for model transformation specification

Patterns can be used to aid the process of testing transformations. The generic framework for model transformation presented in Figure 1 can be augmented with an intermediate pattern layer between the source and target metamodels as illustrated in Figure 4. The figure augments Figure 1 by adding a Source Pattern and a Target Pattern. The source pattern is a subset of instances of the source metamodel and the target pattern is a subset of instances of the target metamodel. In addition, the input model has to be an instance of the source pattern and the output model an instance of the target pattern.

In our research [10] we have used patterns in specifying transformations. These patterns are class diagram templates or interaction diagram templates that describe features expected of input and output models. This use of patterns illustrates the use of two-layer oracles where one oracle is more coarse-grained and the second oracle is more fine-grained. For example, in the figure, the input model must conform not only to the source metamodel, but also to the source pattern. The figure also differentiates between an input oracle and an output oracle. A transformation must produce correct output for the given input based on the coarse-grained and fine-grained input and output oracles. Correctness can be addressed in a number of ways:

1. The transformation algorithm can encapsulate the features and constraints expected of input and out models. However, this will tend to make the algorithm complex since it has to address the constraints on all potential models.
2. Constraints specific to the input model can be associated with the input model but this will make input models more complex. Similarly, constraints specific to the output model can be associated with the output model.
3. A third approach is to use patterns to characterize input and output models. In this way, model properties and constraints that are the same for all model instances represented by the design pattern are specified as part of the pattern. This reduces the need to encapsulate such information directly in the transformation algorithm. In addition, only constraints that differ for each input model need to be specified for each input model. By default, a design pattern has implicit constraints insofar as a pattern specifies structural and behavioural features expected in conforming models. Therefore patterns can reduce the number of constraints that must be associated with each input model as happens for example, when only a coarse-grained metamodel oracle is used.

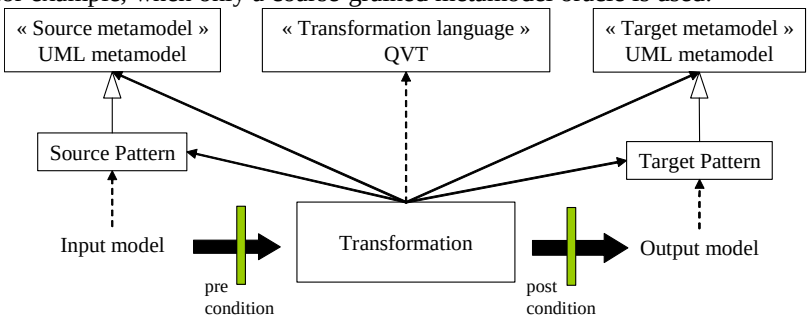

Figure 4 – Transformation with a two-layer oracle

The approach presented in this section supports this testing paradigm through an input oracle (a source pattern) that indicates the validity of input models and an output oracle (a target pattern) that indicates the validity of output models. The source pattern can be very helpful to design accurate test data since it specializes the source metamodel and thus reduces the set of possible input models for the transformation. The target patterns can be very helpful to design an efficient oracle function for the transformation. Indeed, considering the three levels of contracts on the output defined in previous section, these patterns add another level, making the oracle based on contracts more precise.

Several issues must be tackled in this approach. First, in the same way we do not we have to defined techniques to take OCL constraints into account for input data generation, we must also consider taking the source patterns into account. Yet, this is an important issue to make test generation efficient. A second issue concerns the oracle function. The more precise and focused your target pattern, the more efficient your oracle will be, but in the same way, if the pattern narrows the output domain to much it might detect correct models as faulty. So it is important now to define sound methods to design these patterns and to check their validity against what the transformation developer actually wants.

4 Conclusion

In this paper, we have presented several experiments to tackle the issues of model transformation testing. We have especially emphasized two issues: adequate test input generation and design of the oracle function. Based on these different experiences, we have identified important challenges to make model transformations trustable. The first issue is to have well defined criteria that can assess the quality of test data. Black-box criteria should be based on the coverage of the source metamodel structure, but they should also take into account the different constraints defined either on the metamodel or as pre conditions of the transformation. Concerning the oracle function, it seems that will be very difficult to have generic solutions. Here we have talked about different level of contracts that could be associated to any transformation. But also believe that it will be necessary to develop more specific solutions for different categories of transformations. For example, if the output model is executable it can be tested to assess its correctness. In that perspective, a challenge is to categorize transformations and to propose solutions for the oracle in the different cases.

5 References

[1] OMG. *MOF 2.0 Core Final Adopted Specification*. Accessed 2005. http://www.omg.org/cgi-bin/doc?ptc/03-10-04

[2] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 1988. **31**(6): 676 - 686.

[3] F. Fleurey, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. *Proceedings of MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04)*, Rennes, France, November 2004.

[4] B. Baudry, J.-M. Jézéquel, and Y. Le Traon. Robustness and Diagnosability of Designed by Contracts OO Systems. *Proceedings of Metrics'01 (Software Metrics Symposium)*, London, UK, April 2001. IEEE Computer Society Press, Los Alamitos, CA, USA.

[5] L.C. Briand, Y. Labiche, and H. Sun. Investigating the Use of Analysis Contracts to Improve the Testability of Object Oriented Code. *Software Practice and Experience*, 2003. **33**(7).

[6] E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. *Proceedings of Workshop OCL and Model Driven Engineering*, Lisbon, Portugal, 2004.

[7] T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A Tool-Supported Approach to Testing UML Design Models. *Proceedings of ICECCS'05*, Shanghai, China, June 2005.

[8] T. Dinh-Trong, S. Ghosh, and R. France, *JAL: Java like Action Language Specification, Version 1.1*. 2006, Department of Computer Science, Colorado State University.

[9] A. Andrews, R. France, S. Ghosh, and G. Craig. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability*, 2003. **13**(2): 95 -127.

[10] A. Solberg, 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.