Early Experiences on Model Transformation Testing

Alessandro Tiso
alessandro.tiso@unige.it

Gianna Reggio
gianna.reggio@unige.it

Maurizio Leotta
maurizio.leotta@unige.it

Dipartimento interscuola di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università di Genova, Italy

ABSTRACT

Model transformations are the primary artifacts in Model Driven Development and like any other piece of software, must be designed, implemented and tested. Since there are not standard techniques and methods for testing model transformations (especially in the case of model-to-text transformations) we propose an approach to test them. We have used this approach to test a model transformation designed and built using our method for developing model transformations, this transformation maps profiled UML design models into Java desktop applications. We have also created a set of tools to automate the execution of tests on this transformation.

Categories and Subject Descriptors:
D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms: Design.

Keywords: Model Driven Development, Model Transformation Testing, UML.

1. INTRODUCTION

Model transformations are *the heart and the soul* [10] of Model Driven Development (MDD) and must be considered as primary artifacts when applying MDD techniques. Like any other piece of software, model transformations must be designed, implemented and tested. Testing model transformations is a problem more complex than code testing [2]. There are several factors to be considered when evaluating the complexity of this problem, in this paper we covered the following:

- there are a lot of model transformation languages, some are general purpose languages (e.g., Java or C++), others are designed for specific tasks such as Model-to-Model (M2M) transformations (e.g., ATL, QVT, Kermeta) and Model-to-Code (M2C) translations (e.g., Acceleo, Jet, XPand), the latter can be considered as a special case of model-to-text transformation language. This heterogeneity must be taken into account especially in the selection (or definition) of white-box testing technique [3].

- building a good set of models to be used as input of model transformations for the purpose of the transformation testing is a difficult problem. For example, if the input models are instances of a particular meta-model (e.g., UML, profiled UML) that can be constrained by a set of well-formedness rules, then they should contain at least one instance for each class of the meta-model [2]. Clearly, this is not viable, but it is possible to build automatically or manually a set of input models, as small as possible, but at the same time being a good representative of the whole input space; these models must be built following some criteria that are still the subject of studies.

- also the definition of oracle functions for model transformations is difficult due to the complex nature of models. When talking about oracle functions for model transformation tests we must analyze the validity of produced model; this require to analyze syntactic and semantic properties of the output [9].

- writing test cases, managing input and output models, meta-models and model transformation programs require a set of support tools, especially in the case of a chain of transformations, where various model transformation languages may be used. All these tools should be integrated as much as possible, minimizing interoperability problems.

There are other two factors to consider, that we have not addressed in this work. The former is that to test model transformations we need a specification of what the transformation has to do; in general writing specification for model transformations is not easy, and often they are described only in very informal terms. The latter is that input and output of model transformations are models, that are complex data structure [3]. They are often large and require the use of (sometimes complex) tools for their production. They are difficult to be produced by automatic generation, because they must be conform to a specific meta-model and are constrained by well-formedness rules. Finally, manually producing these models is a time consuming task.

Currently there are no standards or well established proposals available for transformation testing, especially in the case of Model-to-Code transformations and transformation chains. This can be due to the fact that it is a difficult problem to solve.

In this paper, we report on our early experience in testing model transformations that are built following a general method for obtaining structured text artifacts (i.e., a set of text files arranged in a specific hierarchies of folders), starting from a UML model, and we sketch an initial proposal for testing model transformations in this specific case.

In Sect. 2, we briefly report our method for producing model transformations, then in Sect. 3 we introduce a specific transformation, developed following our method, that will be used as case study for transformation testing. Sect. 4 describes our approach to testing transformations. Sect. 5 presents the test suite for the case study, and in Sect. 6 we show how the techniques used for testing the transformations may be adapted to test the input models. Related work and conclusions are in Sect. 7 and 8 respectively.

2. OUR METHOD FOR DEVELOPING MODEL TRANSFORMATIONS

Here we briefly sketches our ***M**ethod for **D**eveloping **M**odel **T**ransformations* (shortly MeDMT). MeDMT specifies:

- how to define the model transformation requirements, by giving:
  - the domain of the transformation, as a specific class of well-formed UML models written using a specific profile, and conform to a given set of well-formedness rules;
  - the co-domain of the transformation, as structured textual artifacts, i.e., a set of text files organized in a specific folder hierarchy (e.g., code and configuration files for an application or a set of OWL files defining an ontology);
  - the informal definition of the correspondence between domain and co-domain elements;
- how to design the model transformation, following a specific architecture (see Figure 1) and using a specific notation based on clauses that show how fragments of the domains matching some structural patterns are transformed into elements of the co-domain.

Figure 1 shows the transformation chain used by our method, mapping input UML models into structured textual artifacts. First, a UML Input Model is checked to verify that it satisfies the well-formedness constraints defining the transformation domain, by means of a Model-to-Model transformation (M2M Conformity Check) into an Error & Warning Model. If the Error & Warning Model does not contain errors, then the UML Input Model is re-factorized by a Model-to-Model transformation (M2M Refactoring) to simplify it as much as possible, obtaining a Canonical Model; this step simplifies the creation of the subsequent Model-to-Text transformation (M2T Text Generation). Finally, the Structured

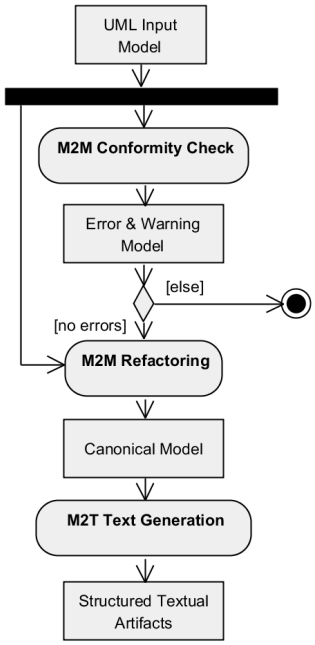

Figure 1: MeDMT Transformation Architecture

Textual Artifacts are obtained. The transformation under test is the composition of the M2M Refactoring and M2T Text Generation transformations.

In the context of the Eclipse Modeling Project, that offers a complete tools infrastructure for MDD, we chose: (1) Eclipse ATL[1] as Model-to-Model transformation language, (2) Eclipse Acceleo[2] as Model-to-Text transformation language and (3) Eclipse OCL as model query language. Eclipse ATL is a hybrid (i.e., provide a mix of declarative and imperative constructs) Model-to-Model transformation language and is accompanied by a set of tools like the ATL IDE (an editor with code completion, builder and debugger). Eclipse Acceleo is a pragmatic implementation of the OMG MOF model-to-text language standard, and it comes with a set of Eclipse plug-in for editing, debugging and launching Model-to-Text transformations. Both ATL and Acceleo allow a good modularity; this feature is very useful to accomplish the task of finding the parts of the transformation that are probably not correct during the testing procedure.

Having chosen the Eclipse Platform we have also the availability of an IDE for Java and a set of plug-in to manage effectively the other technologies used in the target project.

3. CASE STUDY

The selected case study is an application of the MeDMT method, which allows to generate desktop Java applications starting from a UML models representing a detailed design of such applications. The considered design models are made following a precise well-founded method [1]. The domain and co-domain of the case-study transformation are explained in the following.

[1]www.eclipse.org/atl/
[2]www.eclipse.org/acceleo/

The *domain* consists of UML models created following [1] using a profile that provides a set of stereotypes to identify the entities that compose the application, such as:

≪**context**≫ active classes representing entities external to the application interacting with it;

≪**boundary**≫ active classes representing entities taking care of the interaction of the application with some context entities;

≪**executor**≫ active classes representing entities performing some core application activities;

≪**store**≫ passive classes representing entities containing persistent data.

Classes can have associated invariants, and operations can have pre-conditions and post-conditions, all expressed using OCL. Any operation has an associated method which is expressed using the UML action language (since there is not a standard concrete syntax for the UML action language, for simplicity, we use a Java like notation for it). The behaviour of active classes is represented by state machines. These models should adhere to a rich set of well-formedness constraints, helping to avoid the most common design errors.

The *co-domain* consists of Java desktop applications built using the Spring[3] framework as glue-framework and JPA[4] with Hibernate[5] as persistence provider, in the Data Access Layer. The model transformation generates a Maven[6] project containing all the needed source code and configuration files (e.g., the Spring application contexts and the JPA configuration files). We chose this technology stack for the *co-domain* because we wanted to have a realistic case study. This choice led several benefits due to the advanced features of the technology stack components, but at the same time some technological problems in the implementation of the Model-to-Text; for space reasons we do not report on them here. We have also satisfied a non-functional requirement that consists in having all the produced code readable; indeed, our target code appears to be very similar to the one written by a developer.

This transformation was developed during the last year of a three-year PhD course as an application of the MeDMT by one of the authors and it is the object of our testing. The Model-to-Model transformation (M2M Refactoring) specified in Eclipse ATL is composed by 6 rules and 11 helpers whereas the subsequent Model-to-Text transformation (M2T Text Generation) implemented using Acceleo is composed by 61 modules, 403 templates, 73 queries, and 4 Java classes. It can be classified as a prototype developed using a "real" MDE setup (i.e., using technologies and tools actually used in the industry).

[3]www.springsource.org/
[4]www.oracle.com/technetwork/articles/javaee/jpa-137156.html
[5]www.hibernate.org/
[6]maven.apache.org/

4. MODEL TRANSFORMATION TESTING: OUR APPROACH

4.1 Kind of Testing

We consider two main approaches to model transformation testing:

checking static properties of the transformation target (static case) thus assess the presence of specific elements in the target; for instance in the case of Model-to-Text transformation, it is possible to check the presence in the output of specific strings. This is a kind of white-box testing, indeed we must know the internal structure of the transformation to know which elements search in the target (e.g., if we have to check that some expressions have been correctly transformed in Java, we have to know if the transformation adds extra round parenthesis or if the *this* are omitted).

analyzing the execution of the transformation target (dynamic case): this can be performed when the target of the transformation is code (so compilable/executable). A test case for the transformation consists of a pair formed by an input model (i.e., a UML Model) and a test case on the target code (e.g., a JUnit test case for Java) that is produced with the help of state-of-the-art techniques in the field of the co-domain of the transformation. A minimal test case just consists in compiling and running the result of the transformation. To try to automatize the generation and the execution of these kind of tests it is possible to define the code test abstractly in the source model and to extend the transformation itself to convert it in a standard test case on the target; for example if the transformation goes from UML models into Java, we can add some parts in the source model and extend the transformation to map them into JUnit tests. This is a kind of black-box testing, at least for the phase of the test cases conception, indeed only the semantics of the transformation and of the target code is considered (e.g., we can ask if the transformation of an expression E will be greater than 0 in the target, just adding in the source model $E > 0$).

4.2 Test Models and Oracle Functions

In both approaches (static and dynamic cases) we must select input models and oracle functions to be used during the tests. The idea is to build relatively small input models each one used to test a particular kind of possible domain elements instead of large models containing many different elements. For each stereotype in the UML profile used to build the input models there should be at least one test model containing it, and each pattern used in the clauses defining the transformation design should be instantiated in at least one test model. Finally, at least one real-size complete model should be used.

Checking Static Properties of the Target. In the static case, the oracle function analyzes the target obtained by a run of the model transformation, asserting the presence of snippets determined analyzing the input model. The kind of

snippets that must be checked in the target model depends on the patterns specified in the model transformation design, e.g., if in the design there is a clause that states that for each class in the model there must be a class with the same name in the target code, then the oracle function must check the presence of that class.

Analyzing the Execution of the Target. In the dynamic case we exploit the fact that target produced by the model transformation is compilable and executable. The simplest test is to check if the target code may be compiled. If compilation fails, then we have an indication that the model transformation is incorrect. The model transformation developer can find the erroneous part of the Model transformation with the help of the errors reported by the compiler and the design of the transformation itself.

Another kind of testing is produced inserting in the source model test classes and test operations, that drive the generation of executable test cases in the target code. Tests can be written using specific stereotypes for test classes and test operations, and drawing named associations between the test class and the class under test. Moreover, we can specify post-conditions on test operations indicating what action to take during the build of the generated code, depending if the post-conditions are satisfied or not. More in general, to test semantic properties of the transformation we can write, in the input model, operations whose behaviour is known and test operation to verify that the behaviour is the expected one.

If the execution of the tests generated in the target model fails, the error can be in the transformation or in the model, except in the case of two errors one in the model and the other in the transformation that compensate each other, but this can be considered a really rare case, also because the errors in the two field are of a very different nature; thus an investigation of the source model and of the part of the transformation involved in the test is needed. In case of success, we gain in confidence on the correctness of the transformation, also if the case of two compensating errors cannot be totally excluded.

We can consider that in the case of very simple behaviour, qualitatively speaking, the probability of erroneous definition of the model is lower than the probability of erroneous definition of the transformation, so to consider the transformation incorrect.

Taking advantage of the modularity of the model transformations and the way in which the input models are built, in the case of failure of a specific test, we can know, with a discrete approximation, which modules of the transformation are probably not correct.

Regression Tests. A very simple oracle functions is the one that compares the output of a specific run of the model transformation with the expected output, and in case of textual targets the comparison is made without considering white spaces and line breaks.

The expected output is generated by the transformation itself and its correctness is assessed by the transformation developer (possibly with the help of available tools, depending on the nature of the produced artifacts); she/he corrects the transformation until reaching a satisfactory result (i.e., a correct output). This process is feasible because the expected output is human readable.

Since each test model is representative of a particular subset of the domain of the transformation, during each run of the transformation only a subset of the modules that compose it are activated. Moreover, it must produce the same project (not only semantically equivalent, but syntactically the same) for the same test model; this simplifies the comparison between the expected project and the one output of the transformation run.

Regression testing is useful in the case in which new features are added to model transformation or it is refactorized; in the other cases new versions of the expected projects are required.

5. TEST SUITE FOR THE CASE STUDY

All the stereotypes that identify the entities composing the applications and their features appear at least in one model in the set of models used for testing the transformation, as well as all the main patterns for the input models appearing in the clauses in the specification of the design of the model transformation itself. Each test model contains mainly elements with a specific class stereotype (see Section 3), so we have:

datatypes test model containing mainly ≪datatype≫;

executors test model containing mainly classes stereotyped with ≪executor≫;

boundaries test model containing mainly classes stereotyped with ≪boundary≫;

stores test model containing mainly classes stereotyped with ≪store≫.

We have also a test model in which all those stereotypes appear, that represents the design of a small Java desktop application. This model contains 12 classes, 3 datatypes, 7 state machines and 90 operations.

Table 1 shows a summary of the complete test suite for the case study, where we can see that all the small test models are used for static case, dynamic case and regression test, instead the small sized application is not used for regression testing.

Test Input Model	Dynamic Case	Static Case	Regression Tests
Data type	✓	✓	✓
Executor	✓	✓	✓
Boundary	✓	✓	✓
Store	✓	✓	✓
small sized application	✓	✓	–

Table 1: Test Suite

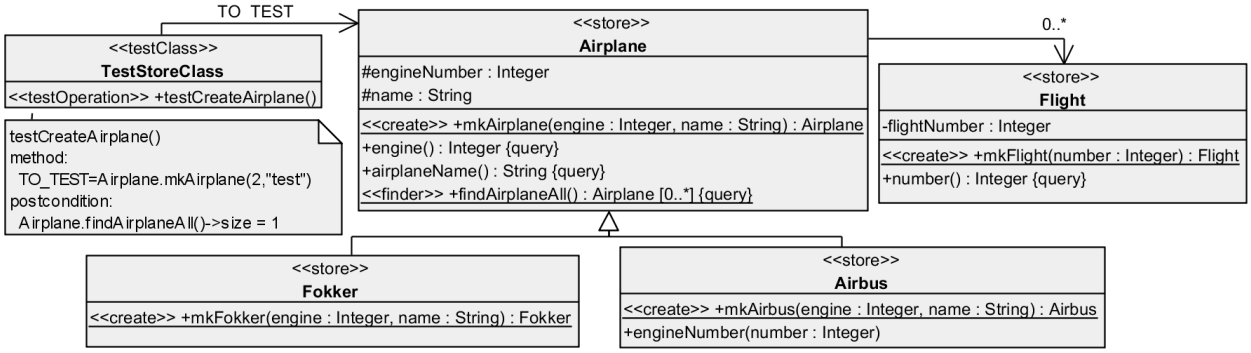

Figure 2: Test On Model

Figure 2 shows a fragment of the store test model that contains mainly classes stereotyped with ≪store≫, some parts of this model will be used by the transformation to generate some tests on the target of the transformation itself.

Classes stereotyped by ≪store≫ are transformed in a set of Java classes and configuration files that manage the persistence of objects of these classes in a database. For instance, we can test that the transformation works as expected writing (1) an operation in the input model whose method triggers the creation of one or more row in the database tables and (2) a post-condition to verify that expected data are actually in the database (using for example a query operation of the same stereotyped class).

The class *TestStoreClass* stereotyped by ≪testClass≫ contains one operation stereotyped by ≪testOperation≫ (see Figure 2). This operation has a method (shown on the class diagram in the note attached to the operation) that specifies the behavior of the operation. The operation has one post-condition, specified by an OCL expression (also this one is shown in the note attached to the operation).

During the transformation, if the test generation parameter is enabled, the class stereotyped by ≪testClass≫ is translated into one JUnit test case, and the method associated with each test operation is translated into one JUnit test method. The OCL is compiled in Java language (during the same run of the transformation that generates all the code), and then used to evaluate the post-condition in the JUnit test method. During the build of the project the JUnit test is activated, a fresh instance of an in-memory database is created and configured with scripts automatically derived from the model, and finally the JUnit test is executed.

Moreover, it is possible to specify in the model if the build of the generated Maven project must fail in case of test failed, or must produce in any case (test failed or not) the artifacts corresponding to the model.

Regarding the static case, if the test generation parameter is enabled, then the transformation can use some elements of the input model (e.g., name of classes, properties of classes etc.) to define parameters for a static analysis tool, and write them in the configuration files (parameters of custom check rules) of the Maven project generated by the model transformation under test. These parameters are then used by custom rules of the static analysis tool to verify that the expected snippets are present in the generated code or if there are elements not expected in the generated project. A set of report files are generated during the build of the Maven project, showing the results of applying these rules.

For example, if we want to check the presence of some attributes in a class we can specify the following property:

```
<property name="attributes"
value="Airplane#name,Airplane#engineNumber" />
```

This property is related to a specific custom rule written for the static analysis tool, that is able to verify the presence of the two attributes, `name` and `engineNumber`, in the class `Airplane`. The rule is then activated during the Maven build and produces files that report the results of its application. This is only a simple example of the possibilities given by the use of a static analysis tools for checking the presence of code snippets in the target project.

Testing execution is automated exploiting the features of Maven. All the tests generated by the model transformation consist of Java code and configuration files placed both in the same project in which the application code is generated, but in a separate folder, that Maven recognizes as the tests container. The developer has only to activate the build process. During the build process tests are executed and report files are produced.

6. TESTING OF THE INPUT MODELS

The techniques used in the dynamic case to test the transformation correctness by means of tests on the transformation target may be used also as a way to perform some tests on the input models. Once that the confidence on the correctness of the transformation has reached an acceptable level, we can use test classes and test operations to define tests on the model elements (UML classes with their operation and, in case of active classes, the behaviour defined by means of state machines), using constraints, as class invariants, pre-conditions and post-conditions for operations. The extended transformation will transform them into tests on the target. The failure of such tests "should" denote a problem in the model, since we are now assuming that the transformation is correct. This goes in the direction of using only models also for what concern testing. A user of the transformation may completely forget about the structure and the technology used in the target of the transformation, and concen-

trate herself/himself on producing and checking the input models, thus increasing the level of abstraction used in the development of software applications, as claimed by MDD.

7. RELATED WORK

Esther Guerra in her work [6] considers Model-to-Model transformations and starting from a formal specification written using her own specification language can derive oracle functions and generate a set of input test models that can be used to test the model transformation using *transML* [7] a family of modelling languages proposed by the same author and others. In our case, the input models dedicated to test and the oracle functions are not automatically generated, but we are driven also by a specification (we use the design of the model transformation), though not formal, to generate input test models and oracle functions.

Lin et al. in their work [8] present a framework for the construction of test cases, execution of test cases, comparison of the output model with an expected one, and visualization of differences. We also, during regression testing, compare output generated by the model transformation with an expected one, but in our case output is a Java project plus configuration files.

Giner and Pelechano in [5] show the definition of a test-driven method for developing Model-to-Model transformations. Applying this method, at each development cycle the transformation is extended to cover a new test case, then it is validated according to the specification expressed using transformation examples as test-cases (in a formal way). The specification is used to generate the input model, that is transformed applying the model transformation under test obtaining the output model. The result is then compared with the expected one. There are many differences and similarities with our approach. Differences are: (i) first this method is applied in Model-to-Model transformation (while our is Model-to-Text); (ii) second our approach is not test driven. There are similarities in the way they capture the requirement of the transformation (using transformation examples as test cases) and the way we design the transformation (using patterns).

Ciancone et al. in their work [4] present an approach for unit testing of QVT-Operational[7] transformations using a white-box way. Using this approach the transformation developer is able to define and execute unit tests using the same QVT-Operational language used for the definition of the transformation. We have two languages involved in our transformation (Acceleo, ATL); considering only the Model-to-Text transformation, the language used, Acceleo, has only the possibility to write the transformation and it is not able to define the test cases at the time in which the transformation is developed.

8. CONCLUSIONS AND FUTURE WORK

This work presents a preliminary version of an approach for testing model transformations, that has been applied in practice on a model transformation (our case study) designed and built using our method for developing model transformations. We have followed the approach for testing the transformation, that it is able to generate desktop applications from precise UML design models [1]. Moreover, it has led us to deal with the tools and the frameworks that support the tasks that must be accomplished to write and test model transformations. We can conclude that simple approaches and effective support tools are very important to develop reliable model transformations in effective way.

In the future, we intend to generalize and formalize this approach adding to MeDMT (our method for developing transformation from UML models into structured textual artifact) detailed guidelines for building (1) input test models and (2) test cases on the result of the transformation, starting from the design of the transformation itself. We plan to add the automatic triggering of the Maven build process after the target project generation. Moreover, we plan to execute some experiments to assess the effectiveness of our approaches.

[7] a language designed for writing transformations

9. REFERENCES

[1] E. Astesiano and G. Reggio. Towards a well-founded UML-based development method. In *SEFM 2003*. IEEE Computer Society, 22-27 September 2003.

[2] B. Baudry, T. Dinh-trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon. Model transformation testing challenges. In *Proceedings of IMDT workshop in conjunction with ECMDA'06*, 2006.

[3] B. Baudry, S. Ghosh, F. Fleurey, R. B. France, Y. L. Traon, and J.-M. Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, 2010.

[4] A. Ciancone, A. Filieri, and R. Mirandola. MANTra: Towards model transformation testing. In *Proceedings International Conference on the Quality of Information and Communications Technology*, pages 97–105, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[5] P. Giner and V. Pelechano. Test-driven development of model transformations. In A. Schürr and B. Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 748–752. Springer, 2009.

[6] E. Guerra. Specification-driven test generation for model transformations. In Z. Hu and J. de Lara, editors, *ICMT*, volume 7307 of *Lecture Notes in Computer Science*, pages 40–55. Springer, 2012.

[7] E. Guerra, J. de Lara, D. Kolovos, R. Paige, and O. dos Santos. Engineering model transformations with transML. *Software and Systems Modeling*, pages 1–23. 10.1007/s10270-011-0211-2.

[8] Y. Lin, J. Zhang, and J. Gray. A testing framework for model transformations. In *Model-Driven Software Development - Research and Practice in Software Engineering*, pages 219–236. Springer, 2005.

[9] J.-M. Mottu, B. Baudry, and Y. L. Traon. Model transformation testing: oracle issue. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, ICSTW '08, pages 105–112, Washington, DC, USA, 2008. IEEE Computer Society.

[10] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, 2003.