Testing Code Generators: a Case Study on Applying USE, EFinder and Tracts in Practice

Zijun Chen [1], Wilbert Alberts [2] and Ivan Kurtev [1,3]

[1] *Technical University of Eindhoven, De Zaale, Eindhoven 5612 AZ, the Netherlands*
[2] *ASML, De Run 6501, Veldhoven 5504 DR, the Netherlands*
[3] *Altran Netherlands, Limburglaan 24, Eindhoven 5652 AA, the Netherlands*

Abstract

A commonly found application of model transformations is in the implementation of code generators where typically a chain of model-to-model and model-to-text transformations is used. A number of approaches for testing such transformations have been proposed in the literature but still there is not much experience in applying them in large non-trivial industrial projects. In this paper we present the results of a case study for improving the testing process of an industrial code generator. We focused on two aspects: automatic support for generating an efficient suite of input test models and alleviating the test oracle problem by using lightweight transformation specifications based on the tracts approach. In both aspects OCL is heavily used as a main specification language. Our experiments involved three tools: USE, EFinder, and TractsTool. We observed that these tools and the corresponding approaches can be useful in practical testing of code generators but improvements in two main directions are needed. On one hand, the proposed approaches should be better adapted to the ecosystems and working assumptions in the industry. On the other hand, the practitioners should reconsider some of their practices in order to fully benefit from the recent academic achievements.

Keywords

Model transformation testing, code generators, classifying terms, USE, EFinder, tracts

1. Introduction

Model transformations are key operations in Model Driven Engineering. One of the main scenarios in which they are used is code generation where one or more models are transformed to executable code in a programming language. Usually, a code generator is implemented as a chain of model-to-model (M2M) and model-to-text (M2T) transformations. Like any other software artifact, these transformations need to be specified, implemented, and checked for correctness. One particular way of checking is testing. In the last decade a number of approaches for transformation testing has been proposed in the literature that address various aspects of the testing process. However, there is still little knowledge on how these approaches perform in an industrial context when applied to non-trivial transformations.

In this paper we present initial results from a case study in which the testing process of an existing industrial code generator is analyzed, points for improvement are identified and various techniques are applied and evaluated. More concretely, we focused on two aspects: generation of an efficient suite of input test models and using a form of transformation specification in order to support the development of test oracles. In this context, an efficient suite is understood as a set of models that do not have duplicated characteristics. More concretely, the space of possible models is partitioned and only a single model from each partition is selected. In order to achieve this we applied the classifying terms approach

Proceedings Name, Month XX–XX, YYYY, City, Country
EMAIL: zijunchen.work@gmail.com (Z. Chen); wilbert.alberts@asml.com (W. Alberts); i.kurtev@tue.nl (I. Kurtev)
ORCID: XXXX-XXXX-XXXX-XXXX (A. 1); XXXX-XXXX-XXXX-XXXX (A. 2); XXXX-XXXX-XXXX-XXXX (A. 3)


[1]. Furthermore, deciding if the result of a particular test is correct is not always a trivial task. This is known as the oracle problem, a challenge inherent to any testing process and in particular in model transformation testing. In order to address this challenge we experimented with the TractsTool and its corresponding technique of specifying model transformations as tracts [2].

The results of this case study brought us to two main conclusions. The techniques we investigated have the potential to solve the intended problems (generation of an efficient test suite and the oracle problem) and to improve the testing process. However, there are still some mismatches between the intended usage of the tools on one hand and the used technologies and practices in the industry on the other hand.

This paper is organized as follows. Section 2 presents the case study, a code generator for a domain-specific language (DSL) for data modeling. Section 3 analyzes the testing process of this generator and motivates the choice for tools and techniques that can improve it. Section 4 presents results from applying the selected tools. Sections 5 and 6 provide a further discussion and conclude the paper.

2. Case Study: Generation of Data Repositories from Domain Data Models

The case study uses an existing DSL and its code generator developed by ASML and Altran Netherlands. The generator produces code used in the TWINSCAN lithography systems developed by ASML. It takes a data model expressed in the DSL and generates a C++ implementation of a repository service. At runtime, clients of the repository service are able to store, retrieve, modify and delete instances of data entity types defined in the input data model. The architecture of the generator is shown in Figure 1:

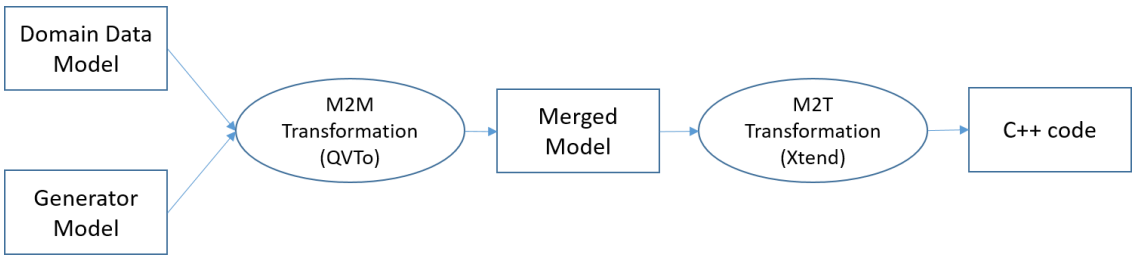

Figure 1: Architecture of the code generator

The generator consists of two transformation steps. The first one is a model-to-model transformation written in QVT operational (QVTo, https://www.omg.org/spec/QVT/). The input of this step is a data model with defined data types and a generator model. The generator model decorates the domain data model by providing information about the actual C++ implementation. This includes a mapping of the user defined primitive types to C++ types, information about the deployment of the repository in memory (for example, storage in heap memory versus shared memory), how the data is provided to the clients (direct access versus clone-based access) and other implementation specific information. The M2M transformation merges the content of the two input models into a single one that is used in the second step. All input and output models conform to their corresponding metamodels (not shown in the figure).

The second step is a model-to-text transformation written in Xtend that generates the actual code. This M2T transformation contains the main logic of the generator.

The language for domain data models resembles UML class diagrams and is relatively simple at a first glance. An example model is shown in Figure 2.

Figure 2: an example domain data model

The example contains two entity types named *A* and *B*. Instances of these types can be created and stored in the repository at runtime. An entity type has multiplicity that indicates the allowed number of instances in the repository at any given time. For example, entity type *A* has multiplicity {0..3} meaning that at most 3 instances of *A* can be stored in the repository. Furthermore, an entity type has usage restrictions shown as icons in the lower part of the type: the helmet indicates if instances of this type can be created; the pencil indicates if instances can be changed while in the repository (crossed pencil defines immutable entities) and the trash bin indicates if the instances can be deleted. Entity attributes and associations are also allowed. The trash bin on the source of the example association mandates that if an instance of *B* is deleted then all instances of *A* that have a link to it have to be deleted as well. If these instances of *A* are further related to other instances by similar associations, their deletion will trigger further deletions, a process known as *cascaded deletion*. This feature, along with the possibly intricate interplay of multiplicities and access restrictions, makes the implementation of the repository service non-trivial. It should be noted that the description given here is a simplification and the language is a part of a larger family of DSLs that supports specification of control logic and interaction with data processing algorithms. Providing more details in this direction is beyond the scope of this paper.

The current testing process of the generator is illustrated in Figure 3:

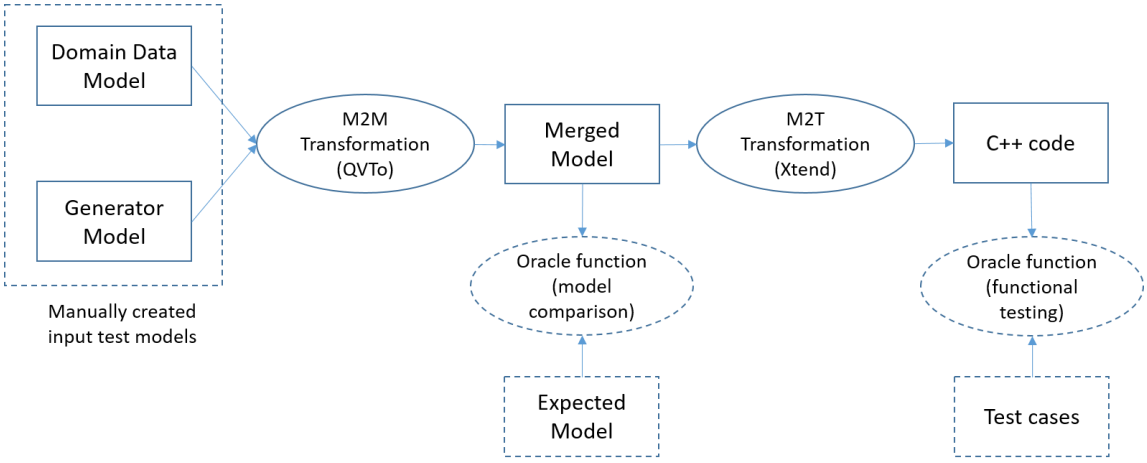

Figure 3: Testing of the code generator

The test-related artifacts are shown in dashed lines. The M2M transformation step is tested with a set of manually created input models (data and generator models). The correctness of an output model is decided by an oracle function that compares the result to a manually created expected model. Since the M2M transformation step is considered less complex, only a small set of test cases is used.

The main purpose of the testing of the M2T transformation is to demonstrate the functional correctness of the generated C++ code. This is done by manually created functional tests executed over the generated code. The test cases generally depend on the input test models. It is crucial that the test models cover all 'interesting' combinations of model elements in the input generator and data models. For example, the generator models should cover the scenarios for memory storage and provisioning, the input models should cover the combinations of various multiplicity ranges and access control. This easily leads to a significant amount of possible combinations.

The presented code generator is being used in practice and has evolved over a number of years. Experience shows that most defects are located in the M2T transformation and are often caused by an untested combination of model elements and generation options. Since the generated code is used in production software, the qualification of the generator (by means of testing) is a crucial step in the project.

3. Analysis of Testing Process and Selection of Testing Techniques

In this section we will analyze the current status and challenges in the described testing process and will motivate the choice of techniques that can improve it.

3.1. Lack of Coverage Information

Test coverage is an important indicator for the quality of a test suite and a testing process. In this case study the coverage takes different forms: (i) a metamodel coverage that shows which metalements are instantiated at least once in the input test models; (ii) transformation coverage that reflects the execution of the M2M transformation code; (iii) coverage of the M2T transformation Xtend code; (iv) coverage of the generated C++ code. Currently, there is no explicit measuring process in place that takes all these aspects into account. Overall, integrated tools that measure and visualize the coverage of the entire chain, starting from metamodel coverage, transformation, and generated code coverage are not readily available although most of this can be easily achieved by integrating existing tools. We consider this more an engineering challenge rather than a research one but we recognize its importance in practice. Development of a dashboard that shows an overview of the coverage for the entire transformation chain is an ongoing work.

3.2. Manual Creation of Input Test Models

Currently, the input test models and the C++ tests are manually created. This is a time consuming and error-prone task and it is identified as one of the main candidates for improvement. There is a number of approaches proposed in the literature that automate the generation of instance models that conform to a metamodel. In the context of model transformations it is important to achieve an effective and efficient test suite. Effective is often interpreted in terms of achieving certain degree of coverage in the transformation specification or implementation. Techniques for model generation that support this goal are not applicable in our case mainly because we use QVTo for implementation (see the next subsections). Furthermore, the engineers indicated that the most important aspect in the development of input test models is the ability to enumerate certain properties of these models (e.g. presence of certain instances or attribute values) and leave the model completion and property combinations to a tool. Another concern is the efficiency of the test suite: to make sure that the input model space is partitioned properly and only a single specimen from a given partition is used. All these considerations motivated the choice of *classifying terms* approach to be used in assisting the process of test model creation. The development of the C++ functional tests was not addressed in this case study, it is briefly discussed in the future work section.

3.3. Lack of Transformation Specifications

In our case study the requirements for the transformations are given in textual form and also on the basis of examples. A number of approaches rely on the availability of a transformation specification in a formal language or in a DSL, for example Pamomo [2][3][4]. Such specifications can solve the oracle problem and also support the automatic generation of an effective test suite that guarantees certain specification coverage. These benefits are counterweighted by the requirements that the developers need to master yet another language and tool and have to provide a complete specification of the transformation. The available time and resources in our investigation did not permit experimentation with tools like Pamomo, for example. Instead, we chose the more lightweight approach of TractsTool [1] where some form of specifications is given as a contract based on OCL expressions. This approach also nicely integrates with the already chosen classifying terms technique.

3.4. Achieving an Effective Test Suite

Some approaches for measuring the quality of the test suite use mutation testing techniques [5][6]; others are driven by achieving maximum transformation coverage [7][8]. Unfortunately they are not directly applicable to our M2M transformation written in QVTo since the proposed techniques work mainly for ATL. ATL is very popular in academia, however, QVTo as an OMG standard remains attractive for many industrial applications. We believe that most of the techniques available for ATL can be transferred to the QVTo language.

3.5. Testing Model-to-Text Transformations

In [9] the problem of testing M2T transformations is reduced to testing M2M transformations by treating the produced text as a model conforming to a very generic metamodel for textual artifacts (folders, files, lines of text, etc.). In principle this approach can be applied in our case to ensure that certain structural properties are present in the generated code. Ideally, these properties should entail functional correctness of the code. In our case study, however, one of the main requirements is to produce evidence that the generated C++ code is correct by means of functional tests. In addition, we need an indication of the expected performance of the generated repository. This is the reason that we did not select the mentioned approach for further experimentation.

4. Application of the Selected Techniques and Tools

We first present our experience in developing and using classifying terms for generating test models. Then the observations from applying the TractsTool are given.

4.1. Using Classifying Terms for Generation of Input Test Models

The classifying terms were identified based on features of the input models considered by the developers as important. Some of them reflected situations that were not properly tested in the earlier versions of the code generator and led to defects later.

Here are some examples of classifying terms for the domain data models given as text:

- Number of entity types with lower bound multiplicity of 0. The characteristic values here are 0 and at least 1
- Number of associations with deletion at the source end
- Lower bound of association target end multiplicity. Characteristic values are 0, a fixed number and infinity
- Upper bound of association target end multiplicity. Characteristic values are the same as above

Examples of classifying terms for the generator models:

- Memory deployment of the generated repository (heap versus shared memory)
- Communication mode (intra versus interprocess)
- Visibility level of the generated software module in the scope of the global software architecture

Each of these terms brings a number of partitions based on the term's characteristic values. In our case study the product of the terms was formed and the corresponding OCL expressions were specified as prescribed by the classifying terms approach. The two input metamodels, their associated validity constraints (in OCL) and the OCL expressions for the classifying terms were used to automatically generate input test models.

The classifying terms approach was initially applied with the USE tool [10]. Unfortunately this tool is not directly applicable to our case because it lacks import and export features for Ecore models and metamodels (at the time of the execution of this research). Therefore, we switched to another tool – EFinder [11], which is built on top of USE and adds the required bridge to the EMF/Ecore ecosystem along with an useful abstraction over various OCL dialects.

Before obtaining the desired set of input test models, a number of challenges were faced, some of them caused by limitations in EFinder, and others caused by the way OCL is used in our DSLs. In the following they are described in details.

- *Usage of Java operations in the metamodels*. The metamodel of the domain data language contains few Java operations that implement navigations over models. These operations are used in the OCL well-formedness constraints (a feature supported by the Eclipse OCL

implementation). This mix of Java and OCL hinders the application of tools like USE and EFinder. In our case, it was possible to ignore the constraints using Java. In the general case we recommend using OCL helpers to implement useful navigation operations;

- *Underspecified well-formedness constraints*. We detected a situation in which USE and EFinder created undesired models that conform to the metamodel and the OCL constraints. This is a sign that the combination of the DSL metamodel and the well-formedness constraints is incomplete. The problem is related to the composition relations in the metamodel. In the domain data DSL, every model element except the root model container have to be contained by another element. The composition relations generally follow the pattern shown in the left hand side of Figure 4. This metamodel fragment does not require that instances of *Part* are always contained by a *Container*. When EFinder is invoked with such a metamodel, sometimes models like the one shown on the right hand side in Figure 4 are created. In practice, such models cannot be created by the user because the used grammar or graphical editor do not allow it. However, tools like USE and EFinder use the metamodel and OCL constraints directly. The solution is either to modify the metamodel by making a bi-directional reference or to add a suitable OCL constraint;

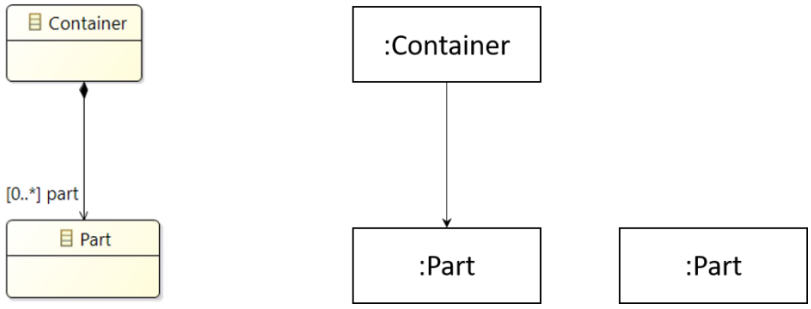

Figure 4: Composition relation that admits objects without a container

- *No distinction between errors and warnings*. It is known that if an OCL constraint fails during validation an error is reported. In other words, OCL does not assign degrees of severity to failing constraints. In our case study there is a distinction between errors and warnings. In case of failure, some constraints produce validation errors and others just warnings. The check of the first kind is wrapped in an operation called *asError*, the second kind is wrapped in an operation *asWarning* which returns *null* in case of failure. In order to use EFinder we excluded these operations. We believe that the distinction between errors and warnings in OCL deserves more systematic attention. Furthermore, a more practical implementation should provide a choice to the user to include/exclude particular OCL constraints given in an OCL document before invoking EFinder;
- *Support for multiple input models*. EFinder supports multiple metamodels as input but the generated output is contained in a single model. In our case we need this single model to be split: one data model and one generator model. To handle this we implemented a post-processing step that splits the generated output to two models. Clearly, EFinder (and USE) solves the fundamental problem of finding a model that is a solution for given constraints. A practical solution further requires adding an adaptation layer that handles the case when the flat result needs to be decomposed;

After solving the enumerated obstacles it was possible to use EFinder for generation of test models. The classifying terms were created by the first author. From this perspective, it still remains unclear how easy for the developers is to come up with the OCL formulation of the classifying terms. Although, the engineers are generally well experienced with OCL, our anecdotal evidence shows that the developers think in terms of small model patterns that have to be present in the input models, possibly

extended with OCL constraints. Potentially, the classifying terms can be synthesized from such more visual representations. An interesting work in this direction is presented in [12][13].

4.2. Using TractsTool and Tracts to Alleviate the Oracle Problem

The TractsTool was used in the testing of the M2M transformation in order to eliminate the current test oracle that uses model comparison. Instead, a tract was defined that specifies the desired properties of the generated models as OCL expressions. This relieves the developers from the task to create a complete reference model.

Generally, we see a good potential in the tracts approach in alleviating the oracle problem. Of course, the quality and the completeness of the tracts is crucial in this approach. A major concern is that the tool is not actively developed anymore and still needs improvements before an industrial application.

5. Discussion and Future Work

In this section we provide a reflection at a more global level by comparing the approaches and tools found in the academic literature and the needs identified in the industry.

We observe a degree of mismatch between the technological assumptions taken in the research tools and the current status of industrial ecosystems. This is first of all evident at the level of used M2M transformation languages. Significant amount of work is based on ATL whereas in practice QVTo is often used because it is a standard. Furthermore, many practical model transformations are written in a GPL like Xtend, Java or Python. In general, we need a better picture on how transformations are implemented in practice. The work of Cabot et al. [14] is a promising step in this direction.

We also observe insufficient awareness of the practitioners about the benefits of using tools like USE and EFinder that are based on strong theoretical foundations (SAT and SMT solving). This is probably due to their early phase of development and degree of immaturity. Raising such awareness in developers should also lead to a more disciplined use of OCL and metamodeling in order to fully benefit from these tools.

Testing a M2T transformation that produces executable code is challenging and reducing it to the problem of testing M2M transformation is often not sufficient in practice. A possible approach is to test the entire generator (or transformation chain) as a black box especially when it is implemented in a GPL. Techniques for test model generation like classifying terms are still applicable but they do not alleviate the oracle problem. We plan to investigate the applicability of model-based testing when the input language has dynamic semantics. Another interesting research direction is metamorphic testing that was recently applied for M2M transformations [15] possibly extended with ideas from compiler metamorphic testing. The main challenge here is the identification of a suitable metamorphic relation that may vary per input DSL.

We also consider investigating the benefits of using explicit transformation specifications which would allow the application of tools like Pamomo.

Final discussion point concerns the validity of our observations. Our conclusions are derived from a single case study, however, we consider it rather representative since the reported challenges are observed in other projects executed in Altran Netherlands. Still, the case study reflects the practices in a single company with projects mainly from the area of high-tech embedded systems.

6. Conclusions

We presented initial results from a case study of testing an industrial code generator that uses a combination of M2M and M2T transformations. A number of challenges were identified and two techniques proposed in the literature were applied: the classifying terms and tracts approaches. The case study revealed that they have the potential to improve the current testing practices with respect to the problems of input test generation and oracle function definition.

We also observed that the used academic tools still need an additional engineering work in order to bring them to the required quality for industrial application. However, the underlying fundamentals are

solid and useful. The presented analysis of the transformation process illustrates some of the industrial needs in the area of model transformations and is a starting point for further research.

In this paper the study was mostly focused on M2M transformations. The M2T transformation component is still not sufficiently addressed. The required functional testing of the generated code involves a lot of manual work mainly because of a missing reference point that specifies the intended code behavior in a more abstract way.

7. References

[1] Martin Gogola, Antonio Vallecillo, Loli Burgueno and Frank Hilken. "Employing classifying terms for testing model transformations". ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS) (2015): 312-321

[2] Martin Gogolla and Antonio Vallecillo. "Tractable model transformation testing". European Conference on Modelling Foundations and Applications (2011): 221-235

[3] Esther Guerra, Juan de Lara, Manuel Wimmer, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Johannes Schonbock and Wieland Schwinger. "Automated verification of model transformations based on visual contracts". Automated Software Engineering (2015)

[4] Esther Guerra and Mathias. "Specification-driven model transformation testing". Software and Systems Modeling (2015): 623-644

[5] Esther Guerra, Jesus Sanchez Cuadrado, and Juan de Lara. "Towards effective mutation testing for ATL". ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)(2019): 78-88

[6] Javier Troya, Alexander Bergmayr, Loli Burgueno, and Manuel Wimmer. "Towards systematic mutations for and with ATL model transformations". IEEE 8th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)(2015): 1- 10

[7] Carlos A Gonzalez and Jordi Cabot. "Atltest: a white-box test generation approach for ATL transformations". International Conference on Model Driven Engineering Languages and Systems (2012): 449-464

[8] Carlos Gonzalez and Jordi Cabot. "Test data generation for model transformations combining partition and constraint analysis". In International Conference on Theory and Practice of Model Transformations (2014): 25-41

[9] Manuel Wimmer and Loli Burgueno. "Testing M2T/T2M transformations". International Conference on Model Driven Engineering Languages and Systems (2013): 203-219

[10] Martin Gogolla. "Model development in the Tool USE: Explorative, Consolidating and Analytic Steps for UML and OCL models". ICDCIT (2021): 24-43

[11] Jesus Sanchez Cuadrado and Martin Gogolla. "Model finding in the EMF Ecosystem". Journal of Object Technology (2020) 19(2): 1-21

[12] Kristof Marussy, Oszkar Semerath, Aren Babikian and Daniel Varro. "A Specification Language for Consistent Model Generation based on Partial Models". Journal of Object Technology (2020), 19(3): 3, 1-22

[13] Kristof Marussy, Oszkar Semerath and Daniel Varro, "Automated Generation of Consistent Graph Models with Multiplicity Reasoning," IEEE Transactions on Software Engineering, doi: 10.1109/TSE.2020.3025732.

[14] Loli Burgueno, Jordi Cabot, and Sebastien Gerard. "The Future of Model Transformation Languages: an Open Community Discussion". Journal of Object Technology (2019), 18(3), 7: 1-11

[15] Javier Troya, Sergio Segura and Antonio Ruiz-Cortes. "Automated inference of likely metamorphic relations for model transformations". Journal of Systems and Software, (2018), 136: 188-208