A Traceability-Driven Approach to Model Transformation Testing

Nicholas D. Matragkas, Dimitrios S. Kolovos, Richard F. Paige, and Athanasios Zolotas

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK
{nicholas.matragkas,dimitrios.kolovos,richard.paige,amz502}@york.ac.uk

Abstract. Effective and efficient support for engineering model transformations is of paramount importance for automating Model-Driven Engineering (MDE) in practice. Such support should include techniques and tools for testing the correctness of model transformations. In this paper, we present a novel approach for identifying incorrect parts of model transformations by using the traceability information produced during the execution of a transformation by a transformation engine. The proposed approach relies on a transformation postprocessor in order to enrich the produced traceability information with domain-specific semantics and then to check automatically its conformance to the transformation specification.

1 Introduction

Model transformations are considered to be the "heart" and "soul" of MDE [14]. As such, they are critical to its success and thus their quality must be ensured. Due to the nature of typical MDE processes (e.g. successive transformations of models until the final implementation of the system is obtained) a fault in a transformation can have unpredictable consequences. When such a transformation is executed, it can result in a faulty model, which itself can be used as input to subsequent transformations. Thus, such a fault can be propagated to successive development steps, resulting in faults in the final implementation of the system [2]. Therefore, to ensure the quality of the developed system, efficient techniques and tools are needed to validate and verify model transformations. Following [10], model-based testing is such a technique and it can contribute to improving the quality of model transformations.

In this paper, we present a novel, tool-supported approach to identifying faults in model transformations and therefore reducing their defect density. When a transformation is executed by a transformation engine, an internal trace is generated. The proposed approach relies on a transformation postprocessor in order to enrich the produced internal trace with domain-specific semantics and then to check automatically its conformance to the transformation specification. A transformation, which generates a non-conforming trace, is erroneous and it should be corrected.

In section 2, we present the motivation for this work, while in section 3 the proposed approach is discussed. Section 4 presents how the proposed approach can be used in a test-driven manner in order to test model transformations. Finally, in section 5 we conclude this paper and we present future work to be carried out.

2 Motivation

Following [13], two main challenges are associated with model transformation testing. The first one has to do with how adequate test data (i.e. test input models) can be generated, while the second one is related to the prediction of the expected outcome of the transformation and its comparison to the actual outcome of the transformation for particular test data.

Generating test cases, manually or automatically, is a very important activity in model transformation testing. Therefore, much research work has focused on this area (e.g., [3, 7, 10]). However, the generation of test cases is out of the scope of this paper and thus we will not discuss about it in more detail. The focus of this paper is on the second challenge, namely the oracle specification problem.

According to the relevant literature, there are two types of oracle functions: complete and partial. Complete oracle functions can be specified either by providing an expected output model for each test input model [11], or by specifying the oracle on the basis of the trace links between the input and output models [9]. In the first case, to verify the correctness of the transformation, the expected and actual models are compared using model comparison algorithms for every test case. If the expected model matches the actual one then the transformation contains no faults. However, model comparison can be both complicated and computationally expensive [1]. Moreover, the tester has to specify an expected output model for every single test case. [13] suggests that for large test input models, which result in large output models, the approach of model comparison is neither practical nor efficient. In such scenarios, partial oracle functions are more appropriate. In the case of the traceability-driven oracle functions, the verification of the correctness of the transformation is performed by comparing a set of correct trace links with the trace links generated by the execution of the transformation on the various test input models. The main advantage of this approach is that the tester does not have to provide an expected output model for every actual output model. Moreover, the set of traces, which is required by such approaches can be rather small [9]. However, the performance of traceability-driven approaches relies heavily on the availability and correctness of good transformation examples in order to generate the correct trace links. Such examples could be difficult to collect [9].

The second category of approaches to oracle specification consists of partial oracle approaches. These approaches are also called specification-conformance checking approaches [9]. Instead of comparing in some way expected and actual output data of transformations, partial oracle functions test transformations against desired properties. Such properties can be either generic or custom.

Generic properties are common for all model transformations, such as confluence or termination. On the other hand, custom properties are transformation specific properties, which have to be checked in order to test the correctness of a specific model-to-model transformation. An example of a partial oracle approach is the one presented in [4]. In this work, the authors use the Object Constraint Language (OCL) [12] to define contracts for the transformation of interest, which consists of pre- and post-conditions. The specified conditions are used to check that when the transformation is executed on the test data, it yields models, which satisfy particular properties. The main limitation of this type of approach is the fact that pre- and post-conditions might be difficult to specify in practice [4]. Moreover, many different conditions might need to be specified in order to cover all different transformation possibilities [1]. According to [15], the complexity of defining contracts for model transformations is of similar complexity to implementing model transformations and therefore it is error-prone as well.

3 Traceability-driven testing of model transformations

In this section we will present a novel approach to reducing the defect density of model transformations, that is the number of defects per model transformation rule. The proposed approach can be considered as a traceability-driven *specification-conformance checking* approach.

Figure 1 illustrates how model transformation testing is currently conducted. When a transformation specification is executed over a set of input of models, output models are generated by the transformation engine and then an appropriate oracle function (partial or complete) is defined using one of the approaches described in Section 2.

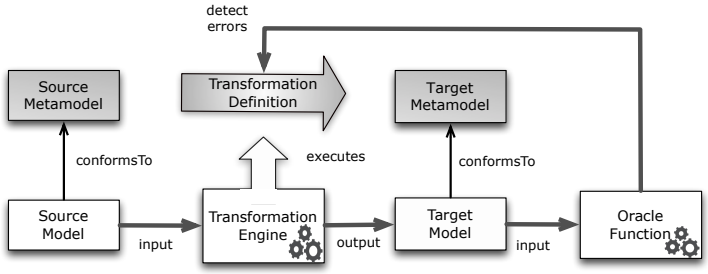

Fig. 1. Current practice of model transformation testing.

In this paper, we propose a different approach to model transformation testing (Figure 2). When a transformation is executed, an internal trace is produced by the transformation engine in addition to the output models. The format of this internal trace depends on the transformation engine, but usually it consists of trace links, which capture mappings between elements in the source model(s),

their corresponding elements in the target model(s), and the rules used to transform the source to the target elements. In the proposed approach, these internal traces can be semantically enriched with domain-specific information such as trace link types. In the spirit of MDE, the enriched trace models have to conform to a metamodel. This case-specific metamodel defines valid mappings between the metamodels of interest. Erroneous parts of a transformation therefore can be identified by generating traces, which do not conform to the traceability metamodel or which violate the metamodel's correctness constraints.

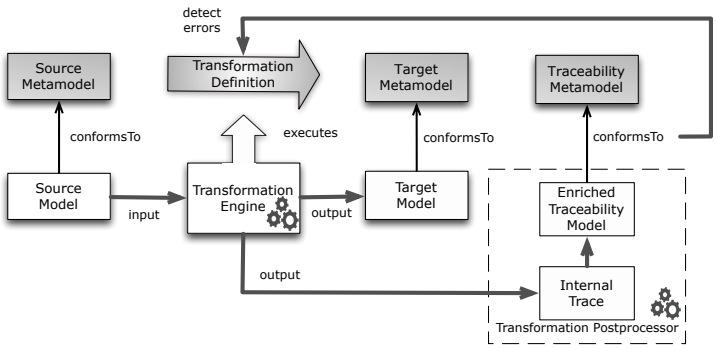

Fig. 2. Proposed approach to model transformation testing.

This approach is based on two main assumptions. First, a detailed traceability metamodel needs to be specified before testing. Second, we assume that the traceability metamodel is correct and complete (i.e. captures all the valid relationships between the metamodels of interest). In the following sections we will briefly present in more detail the proposed approach.

3.1 Specifying the domain-specific traceability metamodel

Defining the correspondences and requirements of a model transformation before its implementation is considered to be best practice [8]. Such correspondences can be modelled using different mapping languages such as transML [8], the Atlas Model Weaver [5] or the Traceability Metamodeling Language (TML) [6]. What all these approaches have in common is the way they encode mappings between metamodels. The mappings are typed, conforming to domain-specific metamodels, which are accompanied by additional correctness constraints. Mappings, which poses these characteristics, are amendable to automatic tool manipulation and analysis. In our reference implementation, we are using TML to capture such mappings.

Imagine a scenario where we want to transform a *Class* model to a *Component* model. The *Class* model conforms to the *Class* metamodel illustrated in Figure 3(a), while the *Component* model conforms to the *Component* metamodel illustrated in Figure 3(b). Such a scenario can arise in a component-based de-

velopment environment, where class diagrams are used to refine the architecture specified by component diagrams into a concrete design.

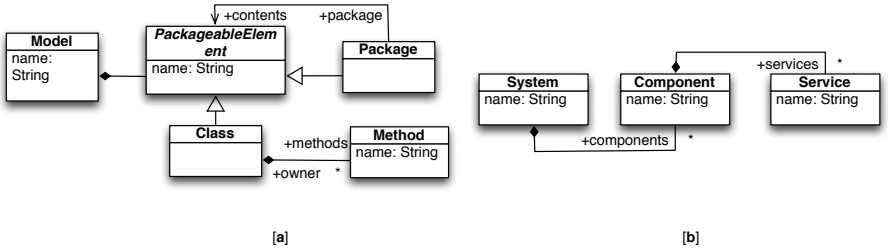

Fig. 3. [**a**] Class metamodel [**b**] Component metamodel

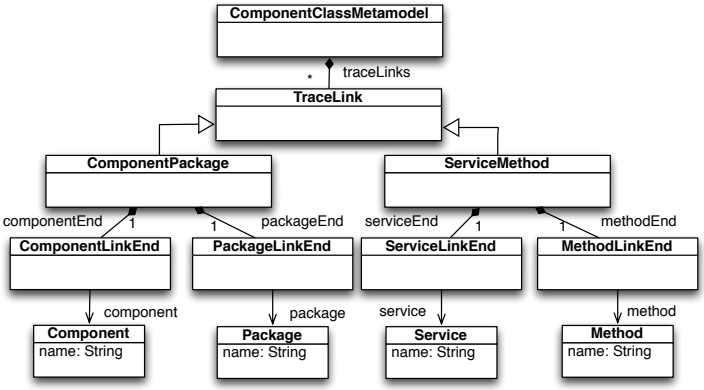

Fig. 4. Mappings metamodel

In this scenario, we want to transform instances of Package from the *Class* metamodel to instances of Component from the *Component* metamodel. Additionally, we want to transform instances of the Method meta-class to instances of the Service meta-class. These mappings can be captured in a scenario-specific TML model. This model is illustrated in Figure 4. It consists of the two aforementioned mappings, namely the *ServiceMethod* and the *ComponentPackage*. Each of the two links has references to the corresponding elements in the metamodels of interest.

Additionally, the following exemplar constraint must be satisfied by the mapping model:

> (**C1**) For each instance of Service in the *Component* metamodel there is exactly one instance of ServiceMethodTraceLink in the mapping model that links it with an instance of Method in *Class* metamodel.

When the mappings and their constraints are captured, the transformation can be implemented and tested.

3.2 Detecting erroneous transformation rules

Once the transformation is implemented, it can be executed over a set of input models. One of the assumptions of our approach is that such a set is available to the engineer.

When the transformation is executed, the transformation engine generates a generic internal trace. Such a trace is generated by most of the contemporary model transformation engines. The metamodel of the internal trace is very similar across the various transformation engines and it is illustrated in Figure 5.

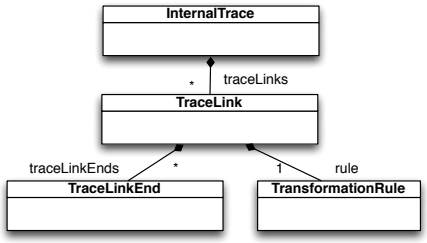

Fig. 5. Internal trace metamodel

We propose the use of a transformation's engine postprocessor, which can use this internal trace in conjunction with the mappings specification described in section 3.1 in order to generate links with case-specific semantics. In the case where the transformation is error-free, the resulting set of case-specific mappings should conform to the TML specification. If there are errors in the implementation of the transformation, then it will generate invalid mappings.

To enrich the internal trace of the transformation engine with case-specific semantics, the postprocessor attempts to match the various links of the internal trace to types of links in the TML model. This can be done by matching the link end types and their cardinalities and by checking whether any mapping constraints are violated.

3.3 Examples of error types

The proposed approach to model transformation testing can detect only errors, which generate invalid traces. Therefore, it can be used ideally in combination with other transformation testing approaches in order to minimise as much as possible the defect density. In this section we will provide some examples of transformation error types, which can be detected by our approach.

Error type I: A possible transformation error might be introduced when an engineer transforms an entity of the source metamodel to an invalid entity in the target metamodel. For this example, consider the scenario where the engineer transforms erroneously instances of the Class meta-class to instances of the Component meta-class. The transformation rule for this transformation is illustrated in Listing 1.1.

Listing 1.1. Class2Component ETL transformation - error type I

```
rule Class2Component transform s : ClassModel!Class
to t : ComponentModel!Component {
    t.name = s.name;
  }
```

When this transformation is executed the transformation postprocessor attempts to generate a traceability model which conforms to the TML model in Figure 4. However, in doing so it can not find any valid link types for the *Class2Component* rule, since there is no link type, whose link ends point to the two meta-classes of the transformation rule. As a result, the transformation postprocessor generates a warning marker next to the rule that caused this problem. This is illustrated in Figure 6.

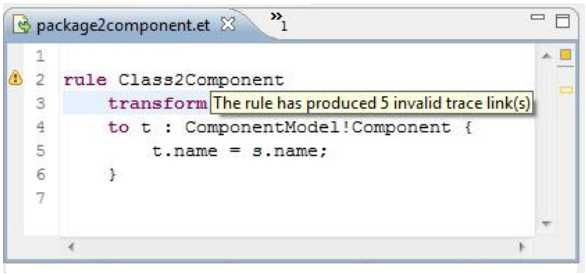

Fig. 6. ETL editor with warning markers.

Error type II: This error type describes transformation rules, which create trace links with wrong cardinalities. Imagine for example that an engineer transforms an instance of the Package meta-class to two instances of the Component meta-class. This transformation rule is illustrated in Listing 1.2. The post processor of the transformation engine will generate a warning, since in the TML model the relationship between instances of the Package meta-class and instances of the Component meta-class is a 1-to-1 relationship and not 1-to-2 as indicated by this rule.

Listing 1.2. Class2Component ETL transformation - error type II

```
rule Class2Component transform s : ClassModel!Package
  to t1 : ComponentModel!Component, t2 : ComponentModel!Component {
```

```
    t1.name = s.name +'1';
    t2.name = s.name +'2';}
```

Error type III: The third type of errors describes erroneous rules which create mappings, which conform to the traceability metamodel but they violate the correctness constraints, which accompany the traceability metamodel. Imagine for example that for every Package meta-class of the *Class* metamodel there is a mapping of type Package2Component. The implementation of this constraint is illustrated in Listing 1.3.

Listing 1.3. EVL constraint

```
context Package {
  constraint OneForEachPackage{
    check : Package2ComponentTraceLink.all.exists(e|e.Package.target =
        self)
    message : 'No links of type Package2Component found for Package ' +
        self
  }
}
```

A possible transformation rule might transform instances of the Package meta-class to instances of the Component meta-class, but only when they contain instances of the Class meta-class. This constraint is expressed as a guard in line 3 of the transformation illustrated in Listing 1.4.

Listing 1.4. Class2Component ETL transformation - error type III

```
rule Class2Component
transform s : ClassModel!Package to t : ComponentModel!Component {
    guard : s.contents.select(c|c.isTypeOf(Class)).size()>0
    t.name = s.name;
  }
```

When this transformation rule is executed, it generates a traceability model which conforms to the traceability metamodel. However, if an instance of the *Package* meta-class does not contain at least one instance of the Class meta-class, then this instance will not be used to generate a corresponding instance of the *Component* meta-class and therefore no trace link will be generated for this particular instance. This violates the constraint of Listing 1.4 Therefore, a validation error will be generated when the validation is executed by the transformation engine's postprocessor.

4 Incremental development and testing of model transformations

In Section 2 we discussed various limitations to specification conformance checking approaches. One of these limitations has to do with the complexity involved

in defining a complete specification for a transformation. To address this issue, we propose the incremental co-development of the transformation and its specification in a test-driven manner. This process is illustrated in Figure 7.

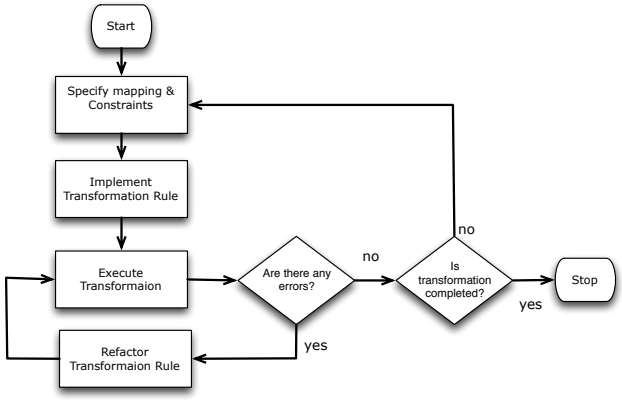

Fig. 7. Process for the co-development of model transformations and specifications.

Initially the engineer specifies a single mapping between the metamodels of interest. This mapping should be atomic in the sense that it will be generated by a single transformation rule. Once the first mapping is defined, the corresponding transformation rule can be implemented and then executed over a set of input models. If the transformation engine postprocessor detects any errors, the engineer can change the implementation of the transformation rule and re-execute it. This process should continue until the postprocessor produces no errors when the transformation rule is executed. Then, the engineer can continue in a similar manner to implement the rest of the transformation.

The benefits of using this incremental approach is twofold. First, since the mapping and its corresponding transformation rule are developed together, the engineer can understand in more depth the various requirements of a particular transformation rule. Moreover, by building the specification in small increments and by testing the transformation after each increment, the complexity of defining the entire specification, as well as implementing the entire transformation, in one go is reduced.

5 Conclusion and future work

In this paper, we presented a novel specification-conformance checking approach to model transformation testing. The proposed approach relies on the traceability information generated by transformation engines in order to identify erroneous transformation rules. Moreover, the proposed approach can be implemented in an incremental manner and thus reducing the complexity of developing transformation specifications and implementations in one go. In the future, we would

like to investigate the integration of the proposed approach with other model transformation testing approaches and how such an integration improves the testing results.

References

1. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA workshop on Integration of Model Driven Development and Model Driven Testing. Bilbao, Spain (Jul 2006), http://www.irisa.fr/triskell/publis/2006/baudry06b.pdf
2. Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM 53, 139–143 (Jun 2010), http://doi.acm.org/10.1145/1743546.1743583
3. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: Proc. ISSRE'06. pp. 85–94. IEEE Computer Society, Washington, DC, USA (2006)
4. Cariou, E., Marvie, R., Seinturier, L., Duchien, L.: Ocl for the specification of model transformation contracts. In: in Proceedings of Workshop OCL and Model Driven Engineering (2004)
5. Didonet, M., Fabro, D., Bèzivin, J., Valduriez, P.: Weaving models with the eclipse amw plugin. In: In Eclipse Modeling Symposium, Eclipse Summit Europe (2006)
6. Drivalos, N., Kolovos, D.S., Paige, R.F., Fernandes, K.J.: Software language engineering. chap. Engineering a DSL for Software Traceability, pp. 151–167. Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-00434-6_10
7. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on. pp. 29–40 (2004)
8. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: transml: a family of languages to model model transformations. In: Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I. pp. 106–120. MODELS'10, Springer-Verlag, Berlin, Heidelberg (2010), http://dl.acm.org/citation.cfm?id=1926458.1926470
9. Kessentini, M., Sahraoui, H., Boukadoum, M.: Example-based model-transformation testing. Automated Software Engineering 18(2), 199–224 (2011)
10. Lin, Y., Zhang, J., Gray, J.: A testing framework for model transformations. Model-Driven Software Development pp. 219–236 (2005)
11. Mottu, J.M., Baudry, B., Traon, Y.L.: Model transformation testing: oracle issue. In: Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on. pp. 105–112 (2008), http://dx.doi.org/10.1109/ICSTW.2008.27
12. Object constraint language, version 2.2. OMG Group (2010), version 2.2
13. Schönböck, J.: Testing and Debugging of Model Transformations. Ph.D. thesis, Faculty of Informatics, Vienna University of Technology (2011)
14. Sendall, S., Kozaczynski, W.: Model transformation: The heart and soul of model-driven software development. IEEE Software 20, 42–45 (2003)
15. Vallecillo, A., Gogolla, M., Burgueño, L., Wimmer, M., Hamann, L.: Formal specification and testing of model transformations. In: Proc. SFM'12. pp. 399–437. Springer-Verlag, Berlin, Heidelberg (2012)