Model-Driven Test Case Design for Model-to-Model Semantics Preservation

Christopher Gerking
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
christopher.gerking@upb.de

Jan Ladleif
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
jladleif@mail.upb.de

Wilhelm Schäfer
Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Zukunftsmeile 1
33102 Paderborn, Germany
wilhelm@upb.de

ABSTRACT

Model transformations used in model-driven software development need to be semantics-preserving, i.e., the meaning of a model must not be distorted by the transformation. Testing whether a transformation preserves the dynamic semantics of a model requires oracles such as model checkers, which explore the runtime statespace of models. The high amount of repetitive code to integrate heterogeneous transformation engines and test oracles makes the design of semantics preservation tests a tedious task. In this paper, we apply the approach of model-driven testing to the domain of model transformation. We present a visual domain-specific language for the design of model transformation tests, which reduces test cases to their essential components. Our language enables an immediate execution of test cases with precise validation feedback. We evaluate our approach in terms of a case study based on the MechatronicUML modeling language for the software development of cyber-physical systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.2.9 [**Software Engineering**]: Management—*software quality assurance*; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*model validation and analysis*

General Terms

Design

Keywords

Model transformation, test case design, semantics preservation

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...$15.00
http://dx.doi.org/10.1145/2804322.2804323

1. INTRODUCTION

Model transformations are an integral part of contemporary model-driven software development (MDSD) processes. They play the vital role of bridging the gap between platform-independent models and concrete execution platforms, generating executable software systems from abstract specifications. In order to ensure that a generated software system meets its specified requirements, model transformations in MDSD need to be *semantics-preserving*, i.e., the meaning of a model must not be distorted (only refined) by the transformation. However, proof techniques for semantics preservation during model transformations are still in the early stages of development [10]. Therefore, viable quality assurance for model transformations reduces mainly to testing approaches [2].

Test cases for model-to-model semantics preservation are characterized by a heterogeneous infrastructure in terms of tools, technologies, and methods involved. On the one hand, a variety of special-purpose model transformation languages exists [3], and requires to invoke specific transformation engines during the execution of test cases. On the other hand, different requirements in terms of testing precision give rise to various kinds of test oracles [16], which are consulted during the execution of test cases in order to assess the correctness of the transformation result. For example, validating the output model against syntactic constraints could be a sufficient oracle, when the primary goal is to exclude semantic invalidity. In contrast, model comparison [13] represents an established and more restrictive test oracle, demanding syntactic equality between the output model and a carefully selected, manually certified reference model. As syntactic equality implies semantic equivalence, model comparison is sufficient to test semantics preservation between output and reference model.

However, syntactic equality is far from a necessary condition for semantics preservation, because two disparate models can still be equivalent with respect to a selection of representative semantic properties. Therefore, model comparison often appears as a too strict oracle for the preservation of semantic properties during model-to-model transformations. Especially when the model transformation is work in progress, the structure of the output model might change frequently and cause *false positive* test failures. MDSD suffers from this problem in particular, as it involves behavioral models with intrinsic dynamic semantics, which define the runtime execution behavior of a software system. Static

reasoning at syntax level is inappropriate to argue about these dynamic semantic properties. Hence, testing semantics preservation efficiently requires a test oracle that operates beyond the syntax level, and analyzes output models in terms of their dynamic semantics. Thus, oracles need to explore the runtime statespace of the models involved, using dedicated tools for model simulation or model checking [7].

Test cases for model-to-model semantics preservation require a considerable amount of repetitive *glue code*, usually written in a general-purpose programming language that supports the integration of heterogeneous technologies. For example, a test case could invoke a specific transformation engine to transform an input model into an output model, before invoking a specific model checker to analyse the output model for certain semantic properties. Integrating such heterogeneous technologies makes the test case design for model transformations a tedious task, because the repetitive integration code is irrelevant to the essential logic of the test cases.

In this paper, we apply the approach of *model-driven testing* to the domain of model transformation. We present a visual domain-specific language (DSL) for the design of model transformation tests, which abstracts from irrelevant details and reduces test cases to their essential components. Our DSL supports the visual flow-based modeling of test cases, and enables to specify the flow of models between different components, while abstracting from the concrete execution order. The approach also enables an immediate test execution with precise visual validation feedback. We evaluate our approach in terms of a case study based on the MechatronicUML modeling language for the software development of cyber-physical systems [4].

In summary, the contribution of this paper is (i) a language concept for the model-driven design of test cases for semantics preservation during model-to-model transformations, and (ii) a visual DSL as an application of our concept, enabling the design and execution of semantics preservation test cases in the context of MechatronicUML.

Our paper is structured as follows: Section 2 describes our model-driven testing approach for model transformations. In Section 3, we demonstrate our approach in terms of a visual DSL for the design and execution of test cases. We discuss related work in Section 4, before concluding in Section 5.

2. MODEL-DRIVEN TESTING OF MODEL TRANSFORMATIONS

Common to the typical components of model transformation test cases (e.g., loading test models, invoking transformation engines, or consulting oracles) is their usage of models as inputs or outputs. Hence, regardless of the high amount of repetitive *glue code* that is usually required to integrate heterogeneous components, they share a common type of interface in terms of models. Based on the observation of models as primary interfaces between components, we abstract from their technological distinctions and apply the approach of *model-driven testing* to the domain of model transformation. In Section 2.1, we present a formal design approach for model transformation test cases in terms of a domain-specific modeling language. Based on this modeling approach, Section 2.2 describes the execution of test cases and how to determine the result of an execution.

2.1 Test Case Design

The core elements of a test case are its components, which we model using nodes: Each node represents one specific action, such as loading a model or verifying assertions. To accomplish its task, a node exhibits individual input ports from which it receives data. After its execution, it may issue results to its individual set of output ports. These, in turn, can be connected to input ports of other nodes, yielding a dataflow network. As we specify test cases, each node may also fail: If a node observes unexpected behavior or finds its assertions are incorrect, it issues a relevant error message and triggers the whole test case's failure. Formally, such a system can be summarized as follows:

Definition 1. A *test case* C is a 7-tuple $(V, I, O, D, opt, exe, L)$ with

- the set of nodes V,
- the set of all input ports I,
- the set of all output ports O,
- the dataflow relation $D \subseteq O \times I$,
- a function $opt : I \to \{true, false\}$ determining if an input is optional,
- a function $exe : V \to \{true, false\}$ determining if the execution of that node was successful, and
- a function $L : (V \cup I \cup O) \to \Sigma$ assigning labels to nodes and ports, with Σ the set of labels.

For a node $v \in V$ let $I(v) \subseteq I$ be its disjoint set of input ports and $O(v) \subseteq O$ its disjoint set of output ports.

The above definition describes all possible test cases, but also includes invalid ones that cannot be executed. For example, one could define an input port that is part of more than one node. To amend this, we introduce the notion of *valid* test cases:

Definition 2. A test case $C = (V, I, O, D, opt, exe, L)$ is valid iff

- every input port (analog for output ports) is assigned to exactly one node:
  $\forall i \in I(\exists!\ v \in V\ (i \in I(v)))$,
- the input ports (analog for output ports) have unique labels:
  $\forall v \in V\ (i_1, i_2 \in I(v) \Rightarrow (i_1 = i_2 \vee L(i_1) \neq L(i_2))$,
- each input port is connected to at most one output port:
  $\forall i \in I\ \neg(\exists o_1, o_2 \in O\ ((o_1, i) \in D \wedge (o_2, i) \in D))$,
- every non-optional input port is connected to at least one output port:
  $\forall i \in I\ (opt(i) = false\ \Rightarrow\ \exists o \in O : (o, i) \in D)$ and
- the dataflow relation D is acyclic, i.e., you cannot arrive at the same node using dataflows after leaving it through an output port.

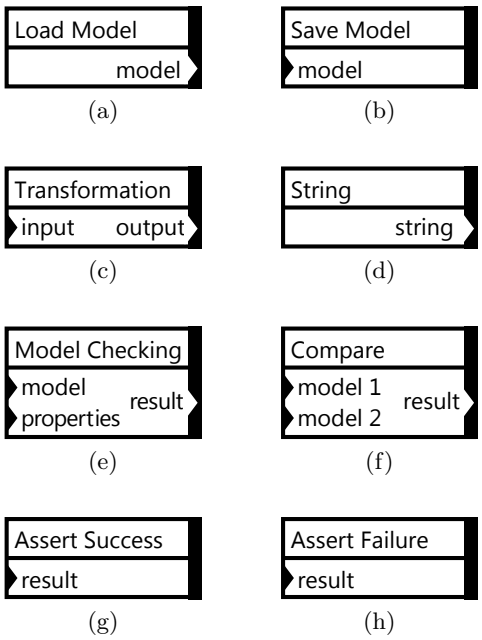

Figure 1: Different Types of Nodes Used to Test Model Transformations

The number and type of ports as well as the concrete nature of the function $exe(v)$ of a node $v \in V$ depend on its *type*. Figure 1 shows a variety of node types that we conceived as part of a visual DSL to test model transformations: Loading (a) and saving (b) models is essential, as is executing model transformations (c). A way to specify and output arbitrary strings (d) is required too, mainly to parametrize model-to-model transformations. Also, a model checker should be available (e) and two models should be comparable on a syntactical level (f). Lastly, the results of the model checking and comparison can either be asserted to be a success (g) or a failure (h).

2.2 Execution of Test Cases

Ultimately, all nodes of a test case should be executed. The order of the nodes' execution is not an arbitrary decision, however. One has to take into account the dependencies that are implied by the dataflow relation D.

Definition 3. A node $v_2 \in V$ directly depends on $v_1 \in V$ ($v_1 \rightsquigarrow v_2$) iff a dataflow $(o, i) \in D$ with $o \in O(v_1)$ and $i \in I(v_2)$ exists. The transitive closure $\rightsquigarrow^*$ of $\rightsquigarrow$ contains all dependencies.

A dependency $v_1 \rightsquigarrow^* v_2$ implies that v_1 has to be executed before v_2. This is the case whenever a node directly or indirectly requires the output of another one for its own computations. A correct order of execution needs to respect all dependencies imposed by $\rightsquigarrow^*$ and actually always exists for valid test cases:

THEOREM 1. *For every valid test case* $C = (V, I, O, D, opt, exe, L)$ *there exists a topological sorting, i.e., a bijective mapping* $ord : V \to \{1, ..., n\}, n = |V|$, *such that*

$$v_1 \rightsquigarrow^* v_2 \Rightarrow ord(v_1) < ord(v_2)\ \forall v_1, v_2 \in V\ .$$

PROOF. The dependency relation $\rightsquigarrow^*$ defines a strict partial order over V: It is irreflexive because we required acyclicity in Definition 2, and transitive because it is defined as a transitive closure (see Definition 3). Thus, the implied graph $G = (V, \rightsquigarrow^*)$ is a directed acyclic graph (DAG). For every DAG a topological sorting of its nodes exists, which in particular yields a topological sorting for every valid test case. □

There are various canonical algorithms to calculate such a topological sorting, e.g., by Kahn [11]. Once a topological sorting has been acquired, a test case can be executed in its entirety. Every node has to finish successfully in order for the whole test case to be regarded a success:

Definition 4. A test case $C = (V, I, O, D, opt, exe, L)$ with a topological sort ord is successful iff

$$\bigwedge_{j=1}^{n} exe(ord^{-1}(j)) = true\ .$$

3. CASE STUDY

The goal of this case study is to demonstrate that our DSL enables an effective test case design and execution for model-to-model semantics preservation. For evaluating our approach, we consider the MECHATRONICUML domain-specific modeling language [4], which targets the model-driven software development for cyber-physical systems. MECHATRONICUML supports modeling of behavioral contracts for real-time coordination by means of Real-Time Statecharts, a combination of UML statemachines and timed automata. One of the key features of MECHATRONICUML is the verification of these contracts against temporal logic safety properties (e.g., deadlock freedom) by means of model checking. To this end, MECHATRONICUML provides a model-to-model transformation which translates Real-Time Statecharts into timed automata that can be analyzed by the model checker UPPAAL [5]. In order to ensure reliable results, the transformation needs to preserve the semantics of the input Real-Time Statecharts, i.e., the output timed automata need to be semantically equivalent. In Section 3.1, we describe our prototypical implementation of a domain-specific testing language in the context of MECHATRONICUML, using Eclipse and a variety of its tools. Afterwards, in Section 3.2, we evaluate our approach by testing the model transformation from MECHATRONICUML to UPPAAL for semantics preservation.

3.1 Implementation

The architectural basis is laid out by two separate metamodels which we model using the Eclipse Modeling Framework (EMF, [19]). Figure 2 shows an overview of our architecture. As EMF is a commonly used standard framework for model-driven software development it makes our test case models easily usable, allowing for a straightforward integration with existing software. The execution logic is added to the metamodel by taking advantage of the EMF Validation Framework[1]: We supply our own strategy to calculate a topological sorting (see Theorem 1), which the EMF Validation Framework uses to execute our nodes in the correct order. Furthermore, a graphical editor implementing our

[1] `https://projects.eclipse.org/projects/modeling.emf.validation`

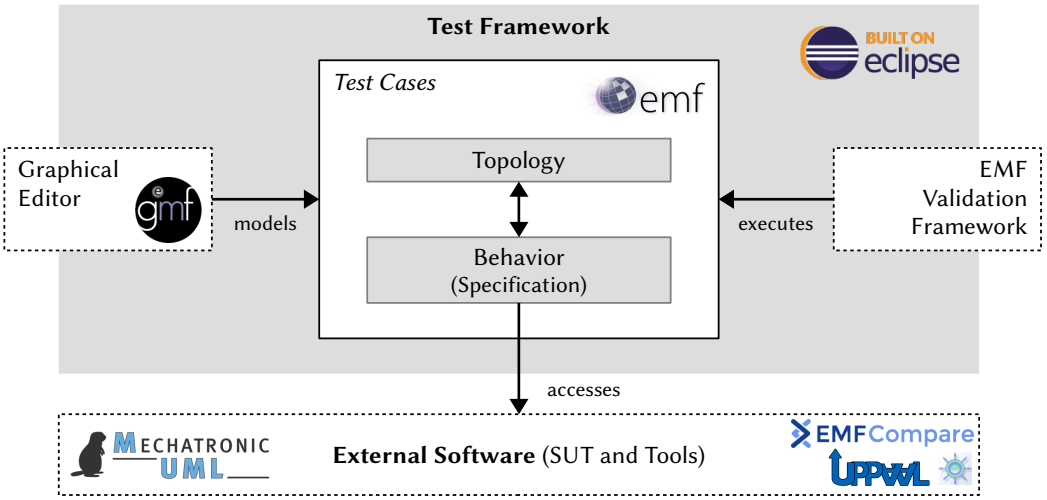

Figure 2: Relationship Between Components in our Framework's Architecture

concrete syntax (see Figure 1) is realized using the Graphical Modeling Framework[2] (GMF).

One metamodel (labeled *Topology*) contains all the topological aspects of the test cases, while the other (labeled *Behavior*) uses a strategy pattern to easily define and implement new node types. They are intertwined such that each topological node has access to its particular set of behavioral instructions. The EMF Validation Framework accesses the topological level to calculate the topological sorting, and afterwards the behavioral level to execute the test case.

The nodes themselves may access any external tool that they need to perform their computations. In our case, we implement the node types given in Figure 1. For this we employ QVTo, an Eclipse integration of the QVT Operational Mappings model transformation standard [18], Uppaal as an external model checker, and EMF Compare [6] to compare arbitrary EMF models. Additionally, our implementation is tailored for use with the MechatronicUML tool suite. It supports the specification of temporal logic properties using a domain-specific variant of the Timed Computation Tree Logic [1], called MTCTL, as well as the transformation of MechatronicUML models to Uppaal-compatible timed automata in order to conduct model checking.

3.2 Evaluation

The evaluation of our approach is based on the guidelines for case studies by Kitchenham et al. [12]. We consider four different MechatronicUML software models of interconnected transportation systems (e.g., autonomous cars, trains, or miniature robots). Our models include an overall amount of thirteen contracts for real-time coordination behavior such as overtaking or collision avoidance. All contracts are equipped with temporal logic verification properties expressed using MTCTL. According to the MechatronicUML semantics, all the attached properties hold on the given models. Our expectation is that the transformation from MechatronicUML to Uppaal preserves these semantics. Thus, the evaluation hypothesis for our evaluation is that our approach allows to design test cases which transform the given MechatronicUML models to Uppaal, and then check for semantics preservation by model checking the given properties on the output timed automata. To this end, we also prepare an erroneous variant of our model transformation, which deliberately introduces semantic distortions between input and output model. We regard our hypothesis as fulfilled if the execution of our test cases clearly separates the semantics-preserving model transformation from the semantics-distorting variant.

Figure 3 illustrates the pattern that we used to design our test cases. Using a *Load Model* node, we first load one of the exemplary MechatronicUML models which already includes a temporal logic property (meaning that once a certain state `x` becomes active, the state `y` will invariably be reached). A *String* node is used to specify the name of the particular coordination contract to transform in a particular test case. Both nodes act as inputs to a third node of type *Transformation*, which represents the execution of our model transformation from MechatronicUML to Uppaal using the QVTo engine. The two outputs of the transformation (a network of Uppaal timed automata, and the translated TCTL properties) connect to a node of type Uppaal, which invokes Uppaal's command line verification tool. Finally, we use an *Assert Success* node to express that the expected model checking result is *true* for all verified properties.

According to the above pattern, we design one test case for each of the thirteen contracts to test. Initially, the *Transformation* nodes in all our test cases refer to the semantics-preserving variant of our model transformation from MechatronicUML to Uppaal. After the test case design, we run all our test cases using our integration with the EMF Validation Framework described in Section 3.1. We observe that our tests run successfully, as the final *Assert Success* node in each of our test cases can be executed without any deviations from our specified expectations. In the next step, we redesign all of our test cases to refer to the semantics-distorting variant of our model transformation. Again, we execute all of our test cases and observe the test results. All of our thirteen test cases fail after switching to the semantics-distorting model transformation, because at least one of the specified MTCTL properties can not be ver-

[2] `http://www.eclipse.org/modeling/gmp/`

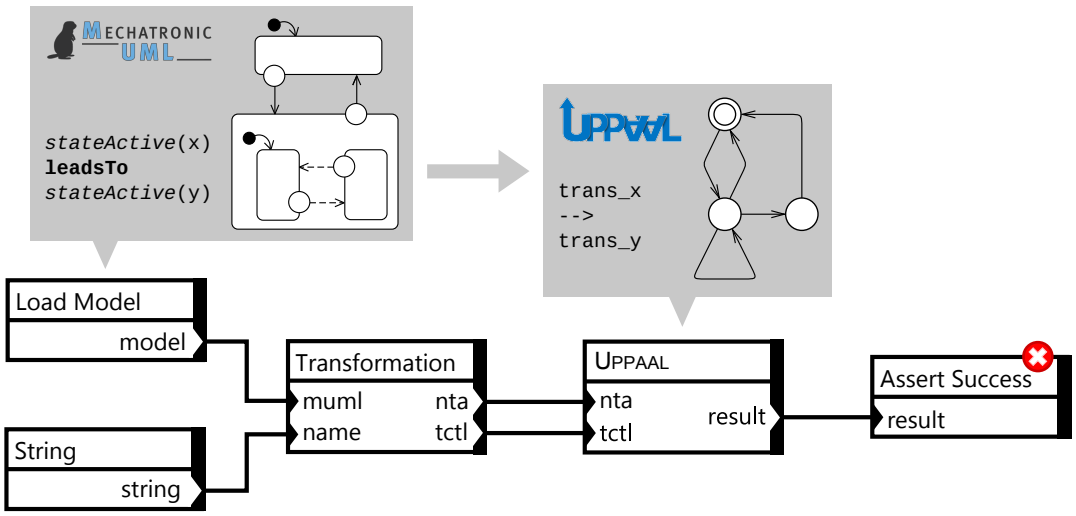

Figure 3: Failing Test for Semantics Preservation of a Model-to-Model Transformation

ified successfully by the UPPAAL model checking. Figure 3 depicts a failing execution of an exemplary test case. The graphical editor gives a precise feedback, by marking the *Assert Success* node as the point of failure.

In summary, our case study successfully separates the semantics-preserving model transformation from its semantics-distorting variant. We therefore regard our evaluation hypothesis as fulfilled, and thus conclude that our approach enables the effective testing of model-to-model semantics preservation.

4. RELATED WORK

In this section, we discuss related work in terms of existing frameworks for model transformation tests, as well as alternative testing techniques for semantics preservation.

Küster et al. [14] describe four test design techniques for the incremental development of model transformations, and discuss their integration into a test framework. Whereas our DSL covers several of these techniques (such as integrity testing against the syntactic constraints induced by the target metamodel, or model comparison against reference outcomes), none of the mentioned techniques explicitly addresses testing at the dynamic semantics level.

García-Domínguez et al. [8] present the EUnit framework for testing of model management tasks such as model transformations. Similar to our approach, they enable modeling of executable test cases by means of dataflow networks. In contrast to our approach, the Epsilon Object Language used for the textual specification of test cases is less abstract than our visual DSL. Although the presented framework is highly extensible, the authors do not explicitly address testing at the dynamic semantics level using model checking or comparable techniques.

Model transformation contracts [9] represent a contrary approach for testing model transformation outputs at syntax level. In general, a contract consists of syntax constraints over the input/output models, whereas one single constraint may also refer to both models and describe a certain relation between model elements. Thus, a contract may restrict the output model's syntax depending on specific syntactic characteristics of the input model, or vice versa. If the specified characteristics are stable (i.e., remain unaffected by changes to a transformation which is work in progress), contracts can reduce the number of false positive test failures in comparison to plain model comparison approaches [13]. Therefore, we regard the specification of contracts as a promising extension to our DSL for test case design. In particular, contracts specified in the scope of a trace model [15] could help to define more precise contracts by referring explicitly to the relations between particular input/output elements recorded during a transformation.

Varró and Pataricza [20] explicitly address the testing of dynamic semantics preservation by means of model checking. In contrast to our approach, they propose a model checking for both input and output models in order to compare the results. Whereas our DSL basically supports this design technique, model checking an input model given in terms of a DSL requires its dynamic semantics to be fully formalized and operational, which is usually a barrier to a successful implementation of the proposed technique. In comparison, we focus on model checking only the output model.

Narayanan and Karsai [17] analyze the semantic equivalence of particular input/output models with respect to a given property. To this end, they check the bisimilarity between particular runtime snapshots, and therefore require an exploration of the runtime statespace for both models. In contrast, whereas our approach explores the statespace of the output model as well, it increases the applicability by employing a general-purpose model checking tool for this task.

5. CONCLUSION AND FUTURE WORK

In this paper, we propose a model-driven design approach for semantics preservation tests in the scope of model-to-model transformations. We provide a concept for a domain-specific modeling language, which abstracts from the repetitive code required to integrate different technologies for loading test models, invoking transformation engines, or consulting oracles. Our modeling approach also enables the test execution with appropriate validation feedback.

Our case study reports the successful implementation of the aforementioned language concept in terms of a test de-

sign language in the context of MechatronicUML, a domain-specific modeling language for the software development of cyber-physical systems. We design a range of test cases including the transformation from MechatronicUML to timed automata, and the verification of particular temporal logic properties on these automata, using the Uppaal model checker as test oracle. The execution of these test cases successfully separates the semantics-preserving model transformation from a semantics-distorting variant.

Design engineers for test cases benefit from our approach, as they require less effort to create executable test cases that integrate different technologies. Both our language concept and implementation are highly extensible in terms of different transformation engines/languages, or alternative tools used as oracles.

Future work on our approach encompasses the integration of alternative testing techniques to our DSL. As discussed in Section 4, model transformation contracts [9] represent a promising approach towards testing model transformations by specifying syntactic relations between input/output models. Especially promising is the approach of specifying such contracts in the scope of a trace model [15], which explicitly relates particular input/output elements and therefore enables a more precise contract definition. Additionally, future work includes design support for parametrized test cases, differing only in terms of their particular input data. Our evaluation demonstrated that test case design often reduces to a common pattern, such that designers highly benefit from a parametrized approach.

6. ACKNOWLEDGMENTS

We thank Stefan Dziwok for providing test models for our case study. Christopher Gerking is member of the PhD program "Design of Flexible Work Environments – Human-Centric Use of Cyber-Physical Systems in Industry 4.0", supported by the federal state of North Rhine-Westphalia.

7. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2–34, 1993.

[2] B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model transformation testing challenges. In H. Eichler and T. Ritter, editors, *Proceedings of the ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing*. Fraunhofer IRB, 2006.

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

[4] S. Becker, S. Dziwok, C. Gerking, C. Heinzemann, W. Schäfer, M. Meyer, and U. Pohlmann. The MechatronicUML method: Model-driven software engineering of self-adaptive mechatronic systems. In P. Jalote, L. Briand, and A. van der Hoek, editors, *36th International Conference on Software Engineering (ICSE Companion 2014)*, pages 614–615, New York, 2014. ACM.

[5] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal – a tool suite for automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, volume 1066 of *LNCS*, pages 232–243, Berlin/Heidelberg, 1996. Springer.

[6] C. Brun and A. Pierantonio. Model differences in the Eclipse Modelling Framework. *CEPIS Upgrade*, 9(2):29–34, Apr. 2008.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge/London, 2000.

[8] A. García-Domínguez, D. S. Kolovos, L. M. Rose, R. F. Paige, and I. Medina-Bulo. EUnit: A unit testing framework for model management tasks. In J. Whittle, T. Clark, and T. Kühne, editors, *Model Driven Engineering Languages and Systems, 14th International Conference, MODELS 2011*, volume 6981 of *LNCS*, pages 395–409, Berlin/Heidelberg, 2011. Springer.

[9] M. Gogolla and A. Vallecillo. *Tract*able model transformation testing. In R. B. France, J. M. Küster, B. Bordbar, and R. F. Paige, editors, *Modelling Foundations and Applications, 7th European Conference, ECMFA 2011*, volume 6698 of *LNCS*, pages 221–235, Berlin/Heidelberg, 2011. Springer.

[10] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, and H. Wehrheim. Showing full semantics preservation in model transformation – a comparison of techniques. In D. Méry and S. Merz, editors, *Integrated Formal Methods, 8th International Conference, IFM 2010*, volume 6396 of *LNCS*, pages 183–198, Berlin/Heidelberg, 2010. Springer.

[11] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, Nov. 1962.

[12] B. Kitchenham, L. Pickard, and S. L. Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[13] D. S. Kolovos, R. F. Paige, and F. A. Polack. Model comparison: A foundation for model composition and model transformation testing. In *Proceedings of the 2006 International Workshop on Global Integrated Model Management*, pages 13–20, New York, 2006. ACM.

[14] J. M. Küster, T. Gschwind, and O. Zimmermann. Incremental development of model transformation chains using automated testing. In A. Schürr and B. Selic, editors, *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, volume 5795 of *LNCS*, pages 733–747, Berlin/Heidelberg, 2009. Springer.

[15] N. D. Matragkas, D. S. Kolovos, R. F. Paige, and A. Zolotas. A traceability-driven approach to model transformation testing. In B. Baudry, J. Dingel, L. Lucio, and H. Vangheluwe, editors, *Proceedings of the Second Workshop on the Analysis of Model Transformations, (AMT 2013)*, volume 1077 of *CEUR Workshop Proceedings*, 2013.

[16] J.-M. Mottu, B. Baudry, and Y. Le Traon. Model transformation testing: oracle issue. In *2008 IEEE International Conference on Software Testing, Verification and Validation Workshop (ICSTW'08)*, pages 105–112. IEEE, 2008.

[17] A. Narayanan and G. Karsai. Towards verifying model transformations. *Electronic Notes in Theoretical Computer Science*, 211:191–200, Apr. 2008.

[18] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Number formal/15-02-01. 2015.

[19] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2nd edition, 2008.

[20] D. Varró and A. Pataricza. Automated formal verification of model transformations. In J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors, *2rd International Workshop on Critical Systems Development with UML (CSD-UML 2003)*, pages 63–78, 2003.