Formal Verification Techniques for Model Transformations Specified By-Demonstration

Sebastian Gabmeyer
Institute of Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstraße 9-11/188-3
Vienna, Austria
gabmeyer@big.tuwien.ac.at

ABSTRACT

Model transformations play an essential role in many aspects of model-driven development. By-demonstration approaches provide a user-friendly tool for specifying reusable model transformations. Here, a modeler performs the model transformation only once by hand and an executable transformation is automatically derived. Such a transformation is characterized by the set of pre- and postconditions that are required to be satisfied prior and after the execution of the transformation. However, the automatically derived conditions are usually too restrictive or incomplete and need to be refined manually to obtain the intended model transformation.

As model transformations may be specified improperly despite the use of by-demonstration development approaches, we propose to employ formal verification techniques to detect inconsistent and erroneous transformations. In particular, we conjecture that methods drawn from software model checking and theorem proving might be employed to verify certain correctness properties of model transformations.

Advisor:	Prof. Dr. Gerti Kappel
Webpage:	`www.big.tuwien.ac.at/staff/sgabmeyer`

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Pre- and post-conditions*

General Terms

Verification

Keywords

Model transformations, model checking, theorem proving, by-demonstration specification, model-driven development

1. INTRODUCTION

Developers implementing a software system usually have a sound understanding of the conditions which are required to be satisfied prior and after the execution of some method. For example, when developing a hashtable API, we (intuitively) reckon that the underlying data structure that stores the entries of the hashtable needs to be properly allocated in memory and initialized. We can express such a requirement as a postcondition of the initialization procedure of the hashtable and as a precondition of all methods modifying the hashtable.

A similar situation is encountered in the context of model-driven development: Here, the primary artifacts are models and model transformations, which conform to the abstract syntax defined by a *metamodel*. Usually, the metamodel defines structural or behavioral restrictions on its conformant models, also referred to as model *instances* or, if the context is unambiguous, just models. For example, a metamodel defining a language for state-transition systems may require that each model instance has a initial state, which has no incoming transition. Then, no model instance may violate this condition at any time, and, in particular, the application of any model transformation to a conforming state-transition model may not remove the initial state or introduce an incoming transition for the initial state.

In the following we refer to the set of pre- and postconditions of a model transformation as the model transformation's *contract*, and the structural and behavioral restrictions defined in the context of the metamodel as the metamodel's *constraints*.

By-Demonstration. Thinking in terms of pre- and postconditions is most naturally supported by so-called *by-demonstration* approaches. Compared to conventional development methods by-demonstration approaches allow the developer to quickly prototype the functionality of a system under development. In essence, a by-demonstration environment receives as input the set of preconditions that must be satisfied prior to the execution of the function/transformation to be specified, and the set of postconditions that must be satisfied after the execution of the function. From this information the by-demonstration environment derives (semi-)automatically the required steps to transform a program/model from a state that satisfies the preconditions to a state that satisfies the postconditions.

In the following we outline a representative work-flow to specify a model transformation with the EMF Modeling Operations (Emo)[1] tool, which provides a by-demonstration model transformation development workbench for the Eclipse Modeling Framework (Emf) [5]. With Emo the modeler defines an *initial* model that describes the preconditions required to be satisfied prior to the execution of the transformation. Next, the modeler demonstrates the transformation and performs the modifications on the initial model that best reflect the effects of the desired transformation. The *revised model* is obtained once all modifications have been demonstrated and it describes the conditions required to be satisfied after the transformation has been applied to a conformant model. Thus, the revised model defines the transformation's postconditions.

After the demonstration has been performed Emo compares the initial to the revised model. Based on the set of detected differences Emo derives an initial contract for the demonstrated transformation. In general, the automatically generated contract is too restrictive and, apart from the simplest transformations, incomplete. Therefore, a manual refinement step is usually necessary to correct and complete the generated model transformation. The necessity of this manual refinement step may, however, not always be immediately apparent as will be shown in the use case scenario presented in the next section, where a conceivable simple transformation might yield unintended results. The rest of this paper is then structured as follows.

Outline. The next section describes the problem of improperly specified model transformations and motivates the necessity of appropriate verification techniques by means of an example. Then, an overview of related work is presented in Section 3 and the necessary steps towards a solution for the problem are outlined in Section 4 together with possible contributions. The paper closes with a list of authored publications.

2. PROBLEM DESCRIPTION

Even EMO operations are not immune to bugs. Consider the following example. Given a metamodel describing a simple UML-like class diagram that consists of named classes and associations. Associations are have two association ends and a name. An association end has a multiplicity and may either be a composition or plain end. A constraint on the metamodel states that if an association end is set to a composition end then the other association end must be plain. Next, we define model transformation for instances of this metamodel with the following use case in mind: a modeler wants to select from the set of associations those whose name is set to has and alter their association end to a composition end if its multiplicity is set to one. A modeler employing a by-demonstration approach starts with an initial model consisting of two classes and an association with name has and an association end with multiplicity one. During the demonstration of the transformation the association end with multiplicity one is changed to a composition and an initial transformation is generated.

Figure 1 depicts on the left hand side three models to which the above specified model transformation is applied and displays the results on the right hand side. The application of the transformation to the first two models yields

[1]http://www.modelversioning.org/
emf-modeling-operations

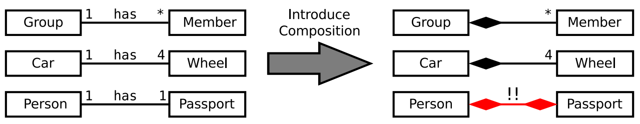

Figure 1: Executing the *Introduce Composition* transformation on three different models

the intended results. The application to the third model reveals an error in the model transformation as it can be applied twice resulting in a model that violates the well-formedness constraint of the metamodel, which states that only one association end may define a composition. To fix this problem the preconditions of the model transformation need to be strengthened. To detect problems of this kind, verification techniques like model checking can be consulted. Therefore, a formal model of EMO's transformation concept is required. In this paper, we provide such a model based on algebraic graph transformations.

The problem may be summarized as follows: Given a metamodel, the metamodel's constraints, and the set of model transformation contracts we want to ensure that no execution of a sequence of model transformations applied to an instance model results in a non-conformant model, i.e., a model that is no longer an instance of the metamodel.

3. RELATED WORK

Various verification techniques have been presented for model transformations of which an overview is given in the following. In [16], Varró et al. propose to verify directed, attributed, and typed graphs with the spin model checker [11]. Hereby, the meta-model, the graph transformations, and an initial model are first translated to an intermediate representation, namely *transition system.* Not till then, the encoding into correspondent Promela code, i.e., the input language of the spin model checker, is performed. On the one hand, this requires an upper bound on the graph transformation system and a preprocessing step at compile that generates the state-space before the translation to Promela code is initiated. On the other hand, the state space may be reduced by this preprocessing step as transitions whose guards are shown to be unsatisfiable may be eliminated. The properties to be verified by spin are specified via *property graphs*, which are automatically translated to LTL formulae [14]. An implementation of their approach is provided by CheckVml.

Another model checking approach is provided by Rensink et al. in [10]. Their implementation, called Groove, represents graphs as labeled transition systems supporting, in addition to the usual node and edge labels, typing via node type labels and attributes by means of special edges. Instead of using an external model checker like Varró et al., they implement a custom model checking algorithm which allows the states in the state space to be represented by graphs [14]. Further, safety and reachability properties, which are based on a temporal logic over graphs introduced in [13], may be expressed by graphs as well. Yet, using graphs to encode the correctness properties and states requires the model checking algorithm to perform sub-graph isomorphism checks, which might become quite expensive.

In contrast to the above described approaches, Boronat et al. propose a (purely) algebraic techniques [3]. That is, they represent model transformations as theories in *rewrit-*

ing logic in order to verify their correctness. Metamodels are hereby converted to correspondent algebraic data types and instances of these metamodels are specified by means of terms over the induced algebraic data types. Graph transformations are represented by their algebraic pendant, namely rewriting rules. The chosen algebraic encoding allows them to employ MAUDE's reachability analysis to verify safety properties and its model checking features to prove liveness properties expressed in LTL.

With a focus on UML component and composite structure diagrams, which are combined with activity diagrams to model the dynamic behavior of the components, Bisztray et al. propose to verify so-called *architectural refactorings* with compositional semantic mappings using *Communicating Sequential Processes* to ensure that the behavior is preserved by the refactoring [2].

To the best of our knowledge Martin Strecker presents the only verification approach using theorem proving techniques and employs for this purpose the interactive proof assistant ISABELLE [15]. Hereby, graphs are set-theoretically specified, graph transformations are represented by a *path formula* and an *action*. The path formula express the application condition of the graph transformation with a subset of first-order logic, while the action describes the performed rewriting steps if the path formula is satisfied.

4. PROPOSED SOLUTION & CONTRIBUTIONS

With our work we aim to identify inconsistent and erroneous model transformations fast. Hence, we want to prevent the execution of improperly specified transformations and verify beforehand if the transformation's contract satisfies the constraints of the metamodel. For this purpose, we plan to employ either model checking or theorem proving techniques developed for the verification of software artifacts. We observe that the metamodel and its constraints provide a specification against which we may verify the model transformation contracts. In accordance with formal verification techniques the specification is represented as a logical formula, which we denote by ϕ. We want verify if the contracts constitute a logical model $\mathcal{M}$ of our specification ϕ, i.e., $\mathcal{M} \models \phi$. Alternatively, the contracts themselves may be expressed as a set of formulas Γ. Then the verification procedure succeeds if we can conclude that the specification ϕ follows from Γ, i.e., we find a proof for $\Gamma \vdash \phi$.

Now the question arises as to how the specification and the model transformation contracts may be encoded in such a manner that either model checking or theorem proving techniques may be employed to verify the model transformation contracts against the specification defined by the metamodels and its constraints.

The PhD thesis's proposed solution progresses along four milestones, each resulting in a distinct contribution.

Benchmark Suite. Transformations are encountered in almost all application scenarios of MDE, among others, model evolution, model refactorings, code generation, and model conversions. To obtain a thorough understanding of the problem domain we will aggregate a set of representative model transformations for each application scenario. As a result we contribute a benchmarking suite consisting of the set of identified, representative model transformations, which produce valid instances of their respective metamodel, and a set of erroneous model transformations that produce non-conformant models. As such the benchmarking suite may be used for evaluation purposes (see below). The overall aim of assembling this benchmarking suite is to identify common characteristics among different kinds of model transformations and establish an understanding on how these characteristics might influence the selection of the verification technique.

Verification Formalisms. Many different formalisms have been proposed in the literature of software verification of which some might be applied successfully to the verification of model transformations. A non-exhaustive list of possible formalisms might start with Pepper's *simple calculus for program transformations* [12], followed by Back's dissertation topic on a *calculus of refinements for program derivations* [1], and continue with more recent contributions as the *ρ-calculus* [6], the *update calculus for expressing type safe program updates* [8], and a *higher-order calculus for graph transformations* [9]. To gain an overview of the different formalisms a survey will be conducted to evaluate their suitability in regard to the verification of model transformations. The benchmarking suite devised in the previous work package will be employed to perform this evaluation.

Encoding. After a suitable formalism has been identified we will propose an encoding capable

- to verify model transformations with an existing model checker, or
- to prove the correctness of the model transformation within an interactive theorem proving environment.

For model checking, an intuitive approach to encode transformations is to view graphs as states and transformations as transitions between them. Thus, if we fix a start graph the (non-deterministic) application of the set of transformations to this start graph and all its successor graphs yields the state-space against which a specification, expressed as graph, can be checked by means of searching the state space for an isomorphic (sub-)graph. This encoding, however, may produce a huge state space even for small instances [16]. For theorem proving, logics of varying complexity may be employed. For example, in [15] a subset of first-order logic is used, which encodes types and associations of the metamodel as unary predicates and associations as binary predicates. Alternatives are rewriting logic [3], which is used by Maude to verify, e.g., concurrent systems, or monadic-second order logic [7], which is capable to express many interesting graph properties, yet, with the disadvantage of increased complexity.

In previous work [4] we have established a formalization of EMO's transformation concepts on the basis of graph transformation theory. This formalization defines the semantics for model transformation created with EMO, and, in the future, we will integrate the chosen encoding into this formal foundation. The final result will then be an extension of EMO, which provides an implementation of our approach for the verification of model transformations.

Evaluation. Finally, we will perform an evaluation of our solution and compare it to existing approaches. This evaluation will investigate the

- efficiency of the solution with respect to run-time and memory consumption;
- effectiveness of the solution with respect to the identification of true erroneous model transformations in the benchmark set compared to the number of false positives or false negatives;
- usability, which will be assessed by means of a user study.

For the purpose of evaluation and communication of (intermediate) results we identified a list of possible conferences, workshops, and journals to which we may submit ongoing work:

1. *Conferences and workshops:* ICSE, FASE, MODELS, POPL, ICST, VOLT, ICMT, TOOLS, TAP, SPLASH, GPCE, OOPSLA, SLE, ECMFA, TACAS
2. *Journals:* Journal on Software and System Modeling, Journal of Object Technology

5. AUTHORED PUBLICATIONS

- P. Brosch, S. Gabmeyer, G. Kappel, M. Seidl: "On formalizing EMF modeling operations with graph transformations"; 5th International workshop on UML and Formal Methods, UML&FM'2012, 2012.
- P. Brosch, U. Egly, S. Gabmeyer, G. Kappel, M. Seidl, H. Tompits, M. Widl, M. Wimmer: "Towards Semantics-Aware Merge Support in Optimistic Model Versioning"; Proceedings of the Models and Evolution Workshop @ MoDELS'11, 2011
- S. Gabmeyer: "Formalizing EMF Modeling Operations based on Graph Transformation Theory"; GTTSE Summerschool 2011, Student Presentations

6. ACKNOWLEDGMENTS

This work was partially funded by the Vienna Science and Technology Fund under grant ICT10-018.

7. REFERENCES

[1] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[2] D. Bisztray, R. Heckel, and H. Ehrig. Verification of Architectural Refactorings by Rule Extraction. In J. Fiadeiro and P. Inverardi, editors, *FASE*, volume 4961 of *LNCS*, pages 347–361. Springer, 2008.

[3] A. Boronat, R. Heckel, and J. Meseguer. Rewriting Logic Semantics and Verification of Model Transformations. In M. Chechik and M. Wirsing, editors, *FASE*, volume 5503 of *LNCS*, pages 18–33. Springer, 2009.

[4] P. Brosch, S. Gabmeyer, G. Kappel, and M. Seidl. On Formalizing EMF Modeling Operations with Graph Transformations. *SIGSOFT Softw. Eng. Notes*, 37(4):1–8, July 2012.

[5] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer. The Operation Recorder: Specifying Model Refactorings By-Example. In *Companion to OOPSLA 2009*, pages 791–792. ACM, 2009.

[6] H. Cirstea, L. Liquori, and B. Wack. Rewriting Calculus with Fixpoints: Untyped and First-Order Systems. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *LNCS*, pages 147–161. Springer, 2004.

[7] B. Courcelle. Monadic Second-Order Logic for Graphs: Algorithmic and Language Theoretical Applications. In A. H. Dediu, A.-M. Ionescu, and C. Martín-Vide, editors, *LATA*, volume 5457 of *LNCS*, pages 19–22. Springer, 2009.

[8] M. Erwig and D. Ren. An update calculus for expressing type-safe program updates. *Science of Computer Programming*, 67(2-3):199 – 222, 2007.

[9] M. Fernández, I. Mackie, and J. S. Pinto. A Higher-Order Calculus for Graph Transformation. *ENTCS*, 72(1):45–58, 2007.

[10] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, and M. Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)*, pages 1–26, 2009.

[11] G. Holzmann. The model checker SPIN. *IEEE Trans. on Soft. Eng.*, 23(5):279 –295, 1997.

[12] P. Pepper. A simple calculus for program transformation (inclusive of induction). *Science of Computer Programming*, 9:221–262, December 1987.

[13] A. Rensink. Towards model checking graph grammars. In S. Gruner and S. L. Presti, editors, *Workshop on Automated Verification of Critical Systems (AVoCS), Southampton, UK*, volume DSSE-TR-2003-02 of *Technical Report*, pages 150–160, Southampton, 2003.

[14] A. Rensink, A. Schmidt, and D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Graph Transformations*, volume 3256 of *LNCS*, pages 219–222. Springer, 2004.

[15] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *ENTCS*, 203(1):135–148, 2008.

[16] D. Varró. Automated formal verification of visual modeling languages by model checking. *Software and Systems Modeling*, 3:85–113, 2004.