Systematic Validation of Model Transformations

Jochen M. Küster

IBM Zurich Research Laboratory
CH-8803 Rüschlikon, Switzerland
email: jku@zurich.ibm.com

Abstract. Like any piece of software, model transformations must be validated to ensure their usefulness for the intended application. Properties to be validated include syntactic correctness as well as general requirements such as termination and confluence (i.e., the existence of a unique result of the transformation for every valid input). This paper introduces the idea of systematic validation and then focusses on validation of syntactic correctness for rule-based model transformations.

1 Introduction

The current MDA initiative favors the use of model transformations within UML-based development of software systems for a number of different purposes. Model transformations are currently used for transforming business models into BPEL [9], for refactoring purposes [17] [18] and for establishing the consistency of UML models [8].

With model transformations being applied in such diverse scenarios, there is a strong need for techniques and methodologies dealing with developing model transformations. Currently, a great deal of research is being performed concerning how to express model transformations and build appropriate tool support which has led to the QVT initiative aiming at standardization [15].

In this paper, we take a different view of model transformations and focus on the situation that will arise if such a common model transformation language has been found. In our opinion, this will give rise to further problems that need to be solved in order to make model transformations practically applicable. In accordance with the famous saying that software processes are software too [16], we believe that model transformations are software too. This means that well-established software engineering techniques must be adapted and extended to model transformations. In the next section, we start with a discussion of model transformation validation. Then, in Section 3, we introduce rule-based model transformations. In Section 4, we present special techniques for ensuring syntactic correctness of these kinds of transformations.

2 Model Transformation Validation

Developing model transformations is not a trivial task. As a consequence, we think that the experiences gained from developing application software should

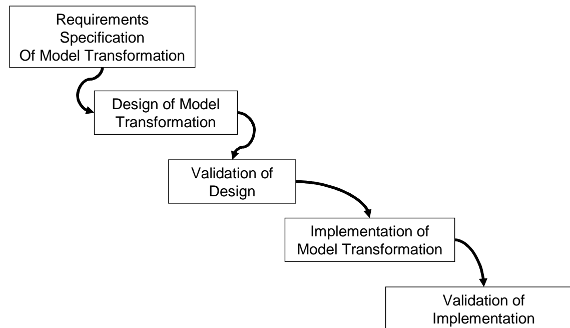

Fig. 1. A simplified waterfall lifecycle for model transformations

be applied to the situation when developing model transformations. This means that model transformations should be developed following well-established software engineering principles such as separation of concern, modularity, rigor and formality, abstraction, anticipation of change and generality. One first step into that direction would be to develop model transformations following a process that distinguishes between requirements specification, design and implementation rather than starting to implement a model transformation straightaway.

In the following, we sketch a simplified waterfall-based development approach for model transformations. We do not believe that a waterfall-based approach will be widely used in practice because it may be too restrictive. In practice, an incremental and iterative approach will probably be needed, as is the case for software development. Nevertheless, we consider a waterfall-based approach a good start for explaining our concepts.

We favor that when developing model transformations first the requirements should be specified. These may range from high-level requirements such as that statecharts must be translated to a process algebra such as CSP [11] (our running example of a model transformation) but may also include functional requirements or performance requirements (such as that a model transformation must be performed using a limited amount of memory or time).

Then the model transformation may be designed using a model transformation language. Similar to software development, different requirements may already be validated after the design phase. This might for example be correctness requirements. Then the design of the model transformation must be implemented in a model transformation tool. Afterwards, further requirements must be validated. Note that, similar to current practice in software development, different model transformation languages may be used for designing and implementing a model transformation.

In general, validation of model transformations may make use of both testing and verification techniques[1]. Traditionally, testing aims at executing the program in selected representative situations by defining suitable test cases [2]. Verification aims at completely analyzing a model transformation against cer-

[1] The distinction of validation and verification is not clearly defined in the literature. Here, we use the term validation to include verification.

tain properties. For software systems, many different verification methods have been developed, from formal correctness proofs to model checking.

We believe that if model transformations are to succeed in practice, research into systematic validation will be needed. This includes the development of specific testing and verification approaches, tailored to and influenced by the underlying model transformation approach.

One important dimension of model transformation validation will include the properties to be checked. These include

- syntactic correctness of a model transformation. This ensures for example that the model transformation produces syntactically correct models.
- termination and confluence of a model transformation. This ensures that the model transformation always produces a unique result. In particular when following rule-based approaches to model transformation approaches this will be important.
- semantic equivalence or semantics preservation of a model transformation. This ensures that the target model is semantically equivalent to the source model or that important properties are preserved by the transformation.
- safety or liveness properties. These might include security or structural properties that should be preserved by the model transformation.

Note that, depending on the type of model transformation, the properties to be checked may vary.

A second dimension will include the underlying model transformation formalism. These may range from relational specifications [1] to graph transformation techniques [5] down to algorithmic techniques for implementing a model transformation [9] (see Czarnecki et al. [6] for a recent classification of model transformation approaches).

In the following, we will concentrate on checking syntactic correctness where we can distinguish between the following types: When a language for expressing model transformations is available, then a concrete model transformation must be syntactically correct with respect to this language. Note that such a model transformation language is not always mathematically formalized but the execution semantics may be hidden in a model transformation tool. A rather different form of syntactic correctness is obtained when looking at the result of a model transformation. There it is often desirable that the result conforms to the syntax of some target language.

3 Rule-based Model Transformations with Control Conditions

Graph transformation has been a traditional candidate for specifying model transformations on visual models. Based on a sound mathematical background, a rich theory is available (see e.g. [7]). From a practical point of view, a number of recent approaches have adapted graph transformation for expressing model

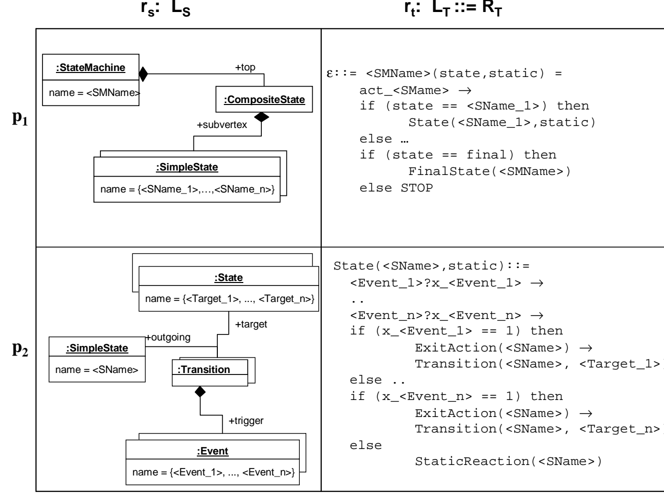

Fig. 2. Transformation rules for statecharts

transformations on UML models [5] [3]. In the following, we will briefly summarize rule-based model transformations with control conditions (based on [10], [12] and [13]) which has been our approach for specifying a complex model transformation from UML to CSP [8].

A model transformation for translating a model from a source language into a target language can be defined by a set of compound rules. Each such compound rule $r : (r_s, r_t)$ consists of two individual rules: The source transformation rule $r_s : L_S ::= R_S$ describes the transformation of the source model (with L_S representing the left hand side and R_S representing the right hand side), the target transformation rule $r_t : L_T ::= R_T$ specifies the transformation of the target model. Typically, source transformation rules will be identical transformations leaving the source model unchanged, represented as $r_s : L_S$ only.

In Figure 2, two compound rules p_1 and p_2 for translating UML statecharts to CSP are shown (based on [12]). Here, each rule $r : (r_s, r_t)$ consists of a UML part and a CSP part. Concerning p_1 in the figure, r_s is the UML part on the left, r_t is the CSP part on the right. Both r_s and r_t can be viewed as graph transformation rules [4]. Regarding the rule p_1 in the figure, L_S is the UML model shown, L_T is the epsilon (denoting emptiness) in the CSP part and R_T is the CSP expression shown on the right.

Source and target rules are coupled by the ability to use shared variables. Such variables are denoted by $\langle variable \rangle$. For example, SMName is such a variable in the compound rule p_1. Furthermore, we allow the use of multi-objects on the left side of the source transformation rule and target transformation rule to denote sets of meta model instances. Each rule where a multi-object occurs on the left side of the source transformation rule can be viewed as a collection of rules, where each rule in the collection fixes the number of objects in the multi-object.

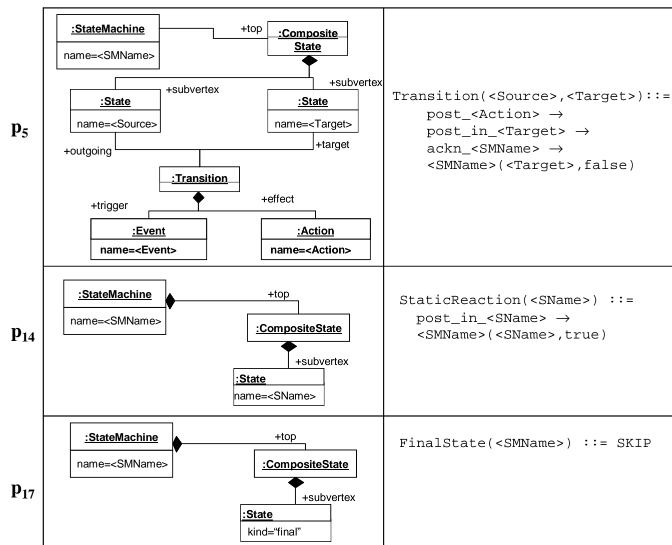

Fig. 3. Transformation rules for statecharts

For translation of a source to a target model, we briefly describe how a compound rule is applied, assuming that $L_S = R_S$ and that $X = \{x_1, .., x_n\}$ is the set of variables of L_S:

1. An occurrence of the left side L_S of the source transformation rule is searched within the source model, such an occurrence is called *source match*.
2. Having found a source match, the variables are given concrete values, leading to a variable instantiation denoted X^I.
3. The left side L_T of the target transformation rule is instantiated with the values of the variables, denoted also by $L_T(X^I)$.
4. An occurrence of the instantiated left side of the target transformation rule is searched within the target model. Such an occurrence is called *target match*.
5. The right side R_T of the target transformation rule is instantiated with the values of the variables.
6. The occurrence of the instantiated left side is replaced with the instantiated right side R_T of the target transformation rule.

In order to specify model transformations with control, compound rules must be assembled to a so-called transformation unit with a control expression specifying how the rules are applied. We will assume that rules are organized in rule sets, which are then organized in a sequence of rule sets where each rule set can be considered as a layer. Within a rule set, rules may be applied nondeterministically.

Syntactically, we express layers of rule sets as follows: Assuming three rules p_1, p_2, p_3, then $\langle\{p_1, p_2\}, p_3\rangle$ specifies two layers, the first one containing p_1, p_2 and the second one containing p_3. This means that first all rules within the first layer are applied and then the ones in the second layer. For each rule set, a simple

marker indicates whether it is applied once or as long as possible. For example, $p \downarrow$ denotes that the rule p is iterated until it cannot be applied anymore. For the example of statechart to CSP translation, we use the control expression of $\langle p_1, p_2 \downarrow, p_5 \downarrow, p_{14} \downarrow, p_{17} \rangle$.

Our concept of rule-based model transformation has been validated in the consistency workbench [8], specifying two large transformation units for translating statecharts and collaborations to CSP.

4 Checking Syntactic Correctness of Transformation Units

To ensure the syntactic correctness of a transformation unit consisting of compound rules, we have to deal with the following problems: With regards to a rule, both the UML part as well as the CSP part must be syntactically correct. Furthermore, the variables used in the CSP part must also occur in the UML part. With regards to a transformation unit, non-terminals created must also be deleted by later rule applications because otherwise the target model might contain these non-terminals. Further, it must be ensured that all rules of a transformation unit are reachable, i. e. there are derivations that can make use of the rule. In the following, we explain how we can statically check the rules.

4.1 Correctness of a Rule

The goal of checking a rule is to ensure that there are no syntactic correctness errors in the UML rule and no syntactic correctness errors in the CSP rule. Checking the UML rule is rather straightforward as it involves simply the decision of whether the left side conforms to the UML metamodel. Here we have to take into account that the UML rule need not be a completely valid UML model (with regards to the OCL constraints specified in [14]). In general, concerning OCL constraints, these can be divided into positive ones requiring the existence of model elements and negative ones requiring the non-existence of model elements. For UML rules, only the negative ones are required to hold. With regards to p_1, this rule involves an extract from a statechart but does not represent a valid statechart.

Checking the CSP rule requires a method of dealing with non-terminals and variables used in it. We propose to check a CSP rule for syntactic correctness by ignoring all non-terminals (i. e. treating them as terminals). For each variable, we propose to replace it with a non-terminal as well. The resulting CSP extract can then be checked aginst the CSP syntax. However, this only gives us an approximation of syntactic correctness of the overall CSP model that will be created in the transformation.

All variables used in the CSP rule must also be used in the UML part. If a variable is used in the CSP rule but not declared, then a *variable mismatch* occurs. Such variable mismatches can easily be found by comparing the set of variables of the CSP rule and the UML rule. In Figure 2, the rule p_1 is not

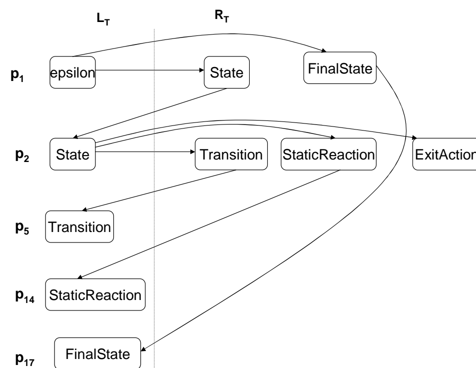

Fig. 4. Non-terminal dependency graph

variable correct. The variable *SMame* used in r_t has not been declared in r_s. Note that the variable correctness criteria can be easily verified by a simple algorithm that computes the set of variables of the left and right-hand sides.

4.2 Correctness of Non-Terminals

In our approach, the left side of a rule should only contain non-terminals of layers previously executed because otherwise there will never be a match for this rule (because the non-terminal has not been created). We suppose that all non-terminals in the CSP part must be marked as such by the software engineer when developing the transformation unit. On the basis of this idea, we realize that a transformation unit has no superfluous non-terminals if

- each non-terminal on a left side of a rule has been created by a rule in a layer above the rule, and
- each non-terminal on a right side of a rule is removed by a rule in a layer below this rule.

For calculating whether or not a transformation unit has superfluous non-terminals, we construct a non-terminal dependency graph (a directed graph) as follows:

- Each non-terminal of a rule becomes a vertex of the graph.
- Each vertex of L_T is connected to all vertices of R_T of the same rule.
- Each vertex originating of R_T of a rule is connected to all vertices of L_T of another rule representing the same non-terminal.

Applying the non-terminal dependency graph to the previous example rules shows that the non-terminal ExitAction is created by rule p_2 but not removed by any other rule. Using the non-terminal dependency graph we can detect whether or not a non-terminal can be removed in principle. As the actual rule application also depends on the matching of L_S and L_T, this can only represent a necessary condition, not a sufficient one.

The non-terminal dependency graph can be further used to explore which rules are unreachable: A rule of a transformation unit is said to be unreachable if the non-terminals of its left side has no incoming edge in the non-terminal dependency graph (except for the first rule).

With regards to Figure 4, the non-terminal dependency graph shows that here all rules are reachable. Note that calculating the non-terminal dependency graph of a transformation unit is rather straightforward. As a consequence, unreachable rules and superfluous non-terminals can be statically detected (before execution) and displayed to the software engineer for correction.

5 Conclusion

The success of model transformation technology will, amongst other things, depend on systematic approaches to designing and implementing model transformations. In this paper, we have sketched the ideas of a model transformation development process and elaborated on the properties that should be validated for a model transformation. Then we have introduced rule-based model transformations with control expressions. For our concrete model transformation approach, we have presented a technique for checking syntactic correctness that can be used to ensure a basic form of it when defining the rules.

We strongly believe that there is a demand for the elaboration of such validation approaches. In particular, testing approaches to model transformation validation should also be explored. For example, when developing a model transformation, the software engineer should be given a means to generate test cases systematically. In the future, we also plan to develop tool support for validation of model transformation approaches.

Acknowledgements: The author thanks Jana Koehler, Rainer Hauser and Shane Sendall for their advice and Reiko Heckel for discussions related to the topic of this paper.

References

1. D. Akehurst and S. Kent. A Relational Approach to Defining Transformations in a Metamodel. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *LNCS*, pages 243–258. Springer-Verlag, 2002.
2. R. Binder. *Testing Object-Oriented System Models.* Addison Wesley, 1999.
3. P. Braun and F. Marschall. BOTL - The Bidirectional Objekt Oriented Transformation Language. Technical report, Fakultät fr Informatik, Technische Universität München, Technical Report TUM-I0307, 2003.
4. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, pages 163–245. World Scientific, 1997.

5. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models . In *Proceedings 17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 267–270, Edinburgh, UK, September 2002.
6. K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *Proceedings OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
8. G. Engels, R. Heckel, and J. M. Küster. The Consistency Workbench - A Tool for Consistency Management in UML-based Development. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 356–359. Springer-Verlag, 2003.
9. R. Hauser and J Koehler. Compiling Process Graphs into Executable Code. In *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering,*. Springer-Verlag, October 2004. To appear.
10. R. Heckel, J. M. Küster, and G. Taentzer. Towards Automatic Translation of UML Models into Semantic Domains. In H.-J. Kreowski and P. Knirsch, editors, *Proceedings of the Appligraph Workshop on Applied Graph Transformation*, pages 11–22, March 2002.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. J. M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, March 2004.
13. J. M. Küster, R. Heckel, and G. Engels. Defining and Validating Transformations of UML Models. In J. Hosking and P. Cox, editors, *IEEE Symposium on Human Centric Computing Languages and Environments (HCC 2003) - Auckland, October 28 - October 31 2003, Auckland, New Zealand, Proceedings*, pages 145–152. IEEE Computer Society, 2003.
14. Object Management Group (OMG). *UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02*, August 2003.
15. Object Management Group (OMG). *QVT-Merge Group. MOF 2.0 Query/Views/Transformations, Revised Submission. OMG document ad/2004-04-01*, April 2004.
16. L. Osterweil. Software Processes are Software too. In *Proceedings of the 9th International Conference on Software Engineering,*, pages 2–13. IEEE Computer Society, March 1987.
17. P. v. Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards Automating Source-Consistent UML Refactorings. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications. 6th International Conference, San Francisco, October 20 -24, USA, Proceedings*, volume 2863 of *LNCS*, pages 144–158. Springer-Verlag, 2003.
18. J. Whittle. Transformations and Software Modeling Languages: Automating Transformations in UML. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language. 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*, volume 2460 of *LNCS*, pages 227–242. Springer-Verlag, 2002.