Behavioral Refinement of Graph Transformation-Based Models

Reiko Heckel[a] Sebastian Thöne[b]

[a] *Faculty of Comp. Science, Electrical Eng., and Math.,* reiko@upb.de

[b] *International Graduate School Dynamic Intelligent Systems,* seb@upb.de

University of Paderborn, Germany

Abstract

Model-driven software engineering requires the refinement of abstract models into more concrete, platform-specific ones. To create and verify such refinements, behavioral models capturing reconfiguration or communication scenarios are presented as instances of a *dynamic meta-model*, i.e., a typed graph transformation system specifying the concepts and basic operations scenarios may be composed of. Possible refinement relations between models can now be described based on the corresponding meta-models.

In contrast to previous approaches, refinement relations on graph transformation systems are not defined as fixed syntactic mappings between abstract transformation rules and, e.g., concrete rule expressions, but allow for a more loose, semantically defined relation between the transformation systems, resulting in a more flexible notion of refinement.

Key words: MDA and model transformation, consistency and co-evolution, refinement of graph transformation systems

1 Introduction

Model-driven software development is based on the idea of refining abstract models into more concrete ones, a recent example being the *Model-Driven Architecture (MDA)* put forward by the OMG[1]. Here, platform-specific details are initially ignored at the model level to allow for maximum portability. Then platform-independent models are refined by adding implementation details required for the mapping to a given target platform. Thus, at each level, more assumptions on the resources, constraints, and services of the chosen platform are incorporated into the model.

[1] www.omg.org/mda/

The set of models conforming to a modeling language is often defined by a *meta-model*, i.e., a class diagram with constraints describing the individual elements of the model and their composition. For behavioral models, this approach is extended towards a *dynamic meta-model*, formalized as a *typed graph transformation system* [3]. Informally, a typed graph transformation system consists of (1) a *type graph* to define the vocabulary of allowed model elements and their relationships, (2) a set of *constraints* to further restrict the valid models, and (3) a set of *graph transformation rules*. Type graph and constraints can be seen as analogous to the classical, static meta-model.

Thus, a model that conforms to a given (static) meta-model is represented as an *instance graph* of the type graph. One can think of the type graph as a UML class diagram and of the instance graph as a corresponding UML object diagram conforming to the types and constraints of the class diagram.

In the case of dynamic systems evolving at run-time, a single instance graph models the system state at a certain point in time only. For also modeling system evolutions, the dynamic meta-model provides graph transformation rules. These are executable specifications that can be used to define local transformations on graphs. Since graphs represent system states, the transformation rules specify, e.g., possible computation, communication, or reconfiguration operations which can be applied to individual states yielding transitions to new states. Based on individual transformation steps, we can explain, simulate, and analyze the behavioral semantics of dynamic models. In particular, we can generate a state transition system that reflects all reachable states of the system with transitions defined by possible transformation steps.

We provide different meta-models for different levels of abstraction. Thus, for refining an abstract model into a more concrete one, we build on a refinement relationship between the meta-models involved. Formally, this relationship is defined by means of an *abstraction function*, as explained in Section 2. Abstraction is a mapping associating with each concrete model a corresponding abstract model, usually by some kind of projection. Based on this, we can check if a concrete model preserves the *structure* of an abstract model.

In Section 3, we provide conditions for what it means to also preserve the *behavior* of an abstract model. We require that the behavior of the abstract model can be simulated at the concrete level, and we discuss how this property can be checked by model checking at the concrete level. For this purpose, we introduce a translation function, contravariant to abstraction, which maps abstract model properties to the concrete level.

2 Structural refinement

A dynamic meta-model is represented as a typed graph transformation system $\mathcal{G} = \langle TG, C, R \rangle$ consisting of a type graph TG, a set of structural constraints C over TG, and a set R of graph transformation rules $r : L \Rightarrow R$ over TG. The set of valid instance graphs typed over TG is called $\mathbf{Graph}_{TG}$.

Like in a previous paper [1], we exemplify this technique by defining architectural styles as meta-models for software architectures: Graph-based models of a software architecture have to conform to a meta-model representing the underlying architectural style. We consider architectural styles as conceptual models of platforms that systems are implemented on. Graph transformation rules specifying the dynamics of a style capture the reconfiguration and communication mechanisms which allow an architecture to evolve at run-time, supported by the respective platform. We will come back to the dynamic aspect in Section 3.

For now, consider a model-driven development process starting with an abstract, business requirements-driven architecture model of a software system which shall be refined into a concrete, platform-specific one. In our simplified example, we assume a component-based architectural style for the platform-independent level where components interact through ports that can only be connected if the provided and required interfaces match.

For the platform-specific level, we assume a style that represents *service-oriented architectures (SOA)*. In SOA, the functionality of components is published as *services* to *service requesters*. Special third-party components, called *discovery agencies*, realize service discovery at run-time, i.e., service provider and requester do not need to know each other in advance. For this purpose, the service-providing component has to publish a description of the provided interface to the discovery agency. A service-requesting component can then use the lookup mechanisms of the discovery agency to find suitable service descriptions for its own requirements.

We do not present the type graphs of these two architectural styles; they can be found in [2]. Basically, they define node and edge types for the architectural concepts summarized above. Instance graphs of these type graphs are used to represent platform-independent or service-oriented architectures respectively. For the sake of readability, we use a UML 2.0-like concrete syntax as shown in Fig. 1. The example describes the architecture of an electronic travel agency application. It requests airline systems to book flights for journeys its clients want to purchase.

Given an abstract transformation system $\mathcal{G} = \langle TG, C, R \rangle$ like the platform-independent architectural style and a concrete transformation system $\mathcal{G}' = \langle TG', C', R' \rangle$ like the service-oriented architectural style, structural refinement establishes a relation between abstract instance graphs $G \in \mathbf{Graph}_{TG}$ and concrete instance graphs $G' \in \mathbf{Graph}_{TG'}$: We require that, in order to be a valid refinement of abstract G, concrete G' has to preserve the structure of the abstract graph.

Since the two instance graphs which shall be compared are expressed over different type graphs, this condition is expressed modulo an abstraction function $abs : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$ that is assumed to be given together with the type graphs, formally: G' is a structural refinement of G if $G \subseteq abs(G')$.

Figure 1 exemplifies the abstraction function applied to the concrete, SOA-specific model of the travel agency system (bottom) yielding the abstract, platform-independent model (top). The abstraction removes all platform-specific elements like the discovery component and the service description and requirements documents. Moreover, the platform-specific stereotype ≪service≫ is adapted to the platform-independent vocabulary ≪component≫.

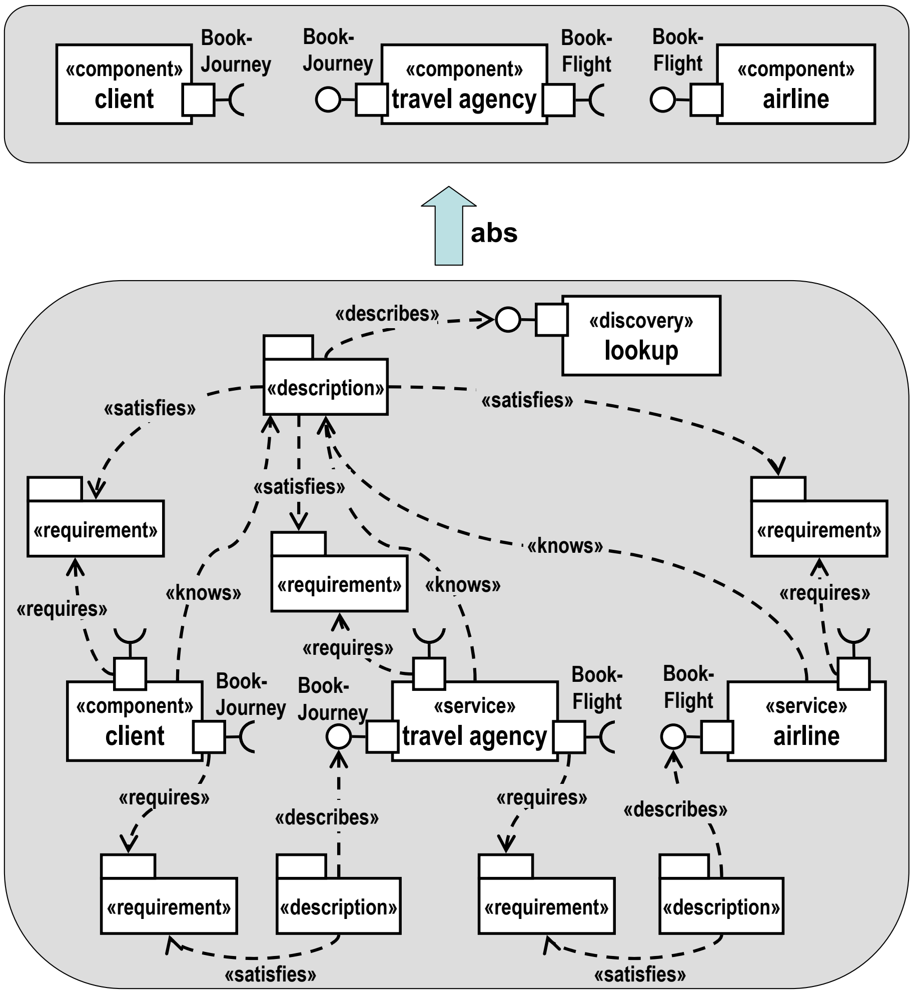

Fig. 1. Abstraction from service-oriented to platform-independent style

There is a range of possibilities for the definition of abstraction functions, from a simple mapping between the two type graphs which can be lifted to instance graphs by renaming the types of the graph elements (cf. [2]) to complex mappings defined by transformation rules, e.g., in order to detect design patterns in reverse engineering. Rather than fixing one concrete way of definition, in this paper we will axiomatize the relevant properties of such mappings.

3 Behavioral refinement

The behavioral part of a dynamic model is defined by the graph transformation rules of its meta-model. For instance, for the abstract, component-based architectural style, we assume that components can dynamically bind to provided interfaces at run-time. This can be realized by appropriate reconfiguration operations for interface binding and unbinding as shown in Fig. 2. In this case, the two transformation rules that define the desired bind and unbind operations are symmetric.

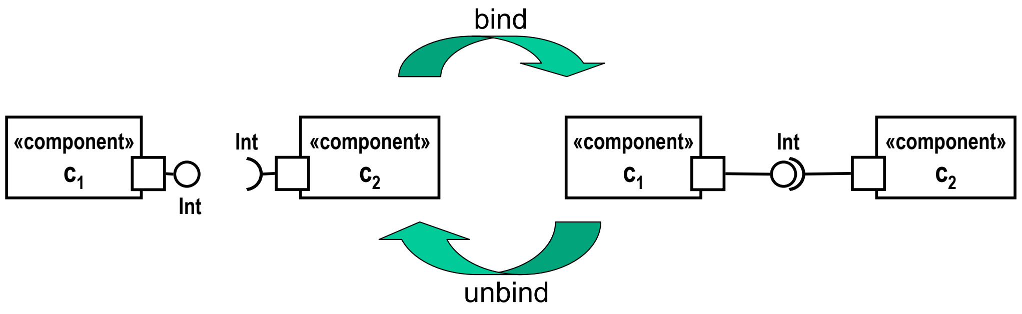

Fig. 2. Reconfiguration rules of abstract architectural style

Formally, the transformation rules are expressed by pairs of instance graphs over the underlying type graph. However, for space reasons and for the sake of better readability, we present them in a UML-like syntax, similar to the instance graphs in the previous section.

Behavior is represented by transitions between instance graphs. The space of possible behaviors is thus given by a *transition system* whose states are the reachable graphs and whose transitions are generated by rule applications. Given the initial state of the model as a start graph, one can generate and explore the transition system by continuously applying transformation rules to previously generated states. To give an example, Figure 3 shows the transition system for the travel agency system in the abstract architectural style. The transitions are labeled by the names of the applied transformation rules. Recently, the automated generation of transition systems from graph transformation system is supported by tools like GROOVE [7] or CheckVML [8].

Similar to the platform-independent style, there are also graph transformation rules in the service-oriented architectural style. However, they have to account for platform-specific restrictions. In the SOA case, for instance, it is required to know the description of a service before it is possible to access it. Therefore, the corresponding reconfiguration rule bind, shown in Fig. 4, includes this additional precondition on its left-hand side. Thus, the bind-operation can only be applied if the service description is known to the component playing the role of the service requester. This is represented by the UML dependency with sterotype ≪knows≫.

The service-oriented style contains further platform-specific transformation rules publish and find, which enable dynamic service discovery by publishing

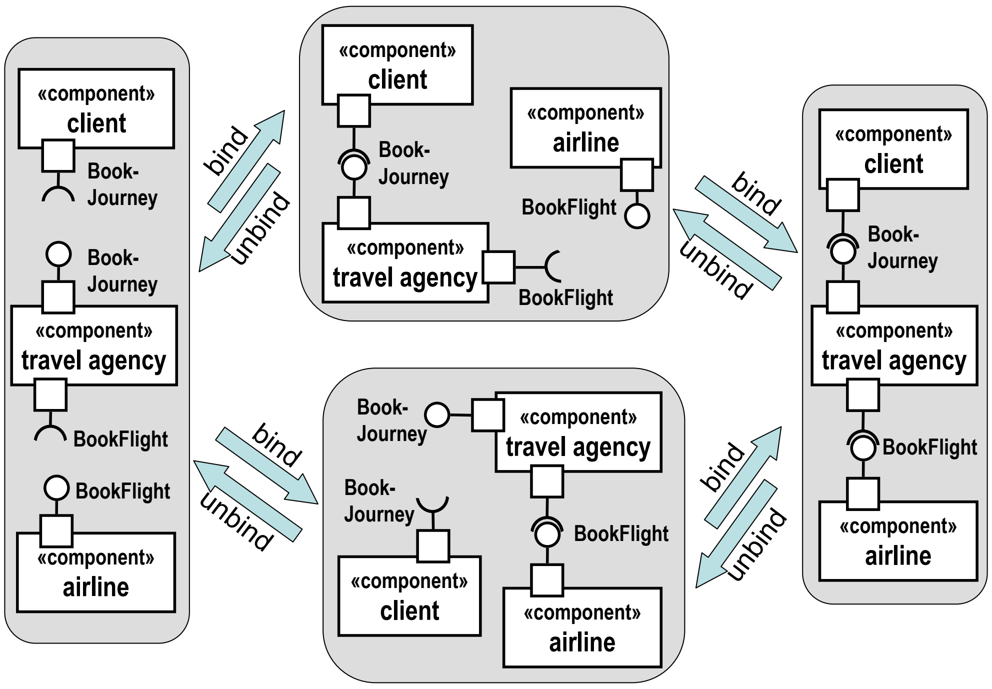

Fig. 3. Transition system for the abstract travel agency architecture

service descriptions to discovery agencies and by querying suitable descriptions that satisfy certain requirements. These operations might be required before a bind-operation can be performed. Due to space limitations, the rules presented in Fig. 4 form a simplified version of the SOA style presented in [1,2].

Like at the platform-independent level, we can now apply the SOA rules to the SOA-specific variant of the travel agency architecture (see Fig. 1), yielding another transition system which represents the platform-specific behavior.

For checking the behavioral refinement of two models (architectures), we now have to take into account the transition systems that can be generated within the underlying dynamic meta-model (architectural style). Formally, we consider again an instance graph G of an abstract system $\mathcal{G} = \langle TG, C, R\rangle$ and an instance graph G' of a concrete system $\mathcal{G}' = \langle TG', C', R'\rangle$. We assume that G' represents a *structural* refinement of G. In order to be a *behavioral* refinement, the behavior of G' must refine the behavior of G. This is the case if every path $G \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ in the abstract transition system has a correspondent path $G' \overset{*}{\Longrightarrow} G'_1 \overset{*}{\Longrightarrow} \ldots \overset{*}{\Longrightarrow} G'_n$ in the concrete transition system with G'_i refining G_i (that is, $G_i \subseteq abs(G'_i)$) for all $i = 1 \ldots n$.

Each step in the abstract system can be matched by a sequence of steps in the concrete system. A single transformation *step* $G_i \Rightarrow G_{i+1}$ of the abstract path is refined by a transformation *sequence* $G'_i \overset{*}{\Longrightarrow} G'_{i+1}$ at the concrete level because it might be necessary to perform a set of consecutive concrete steps in order to realize the abstract one (for example, additional publish and find operations in the SOA case).

Building on the model checking approaches for graph transformation sys-

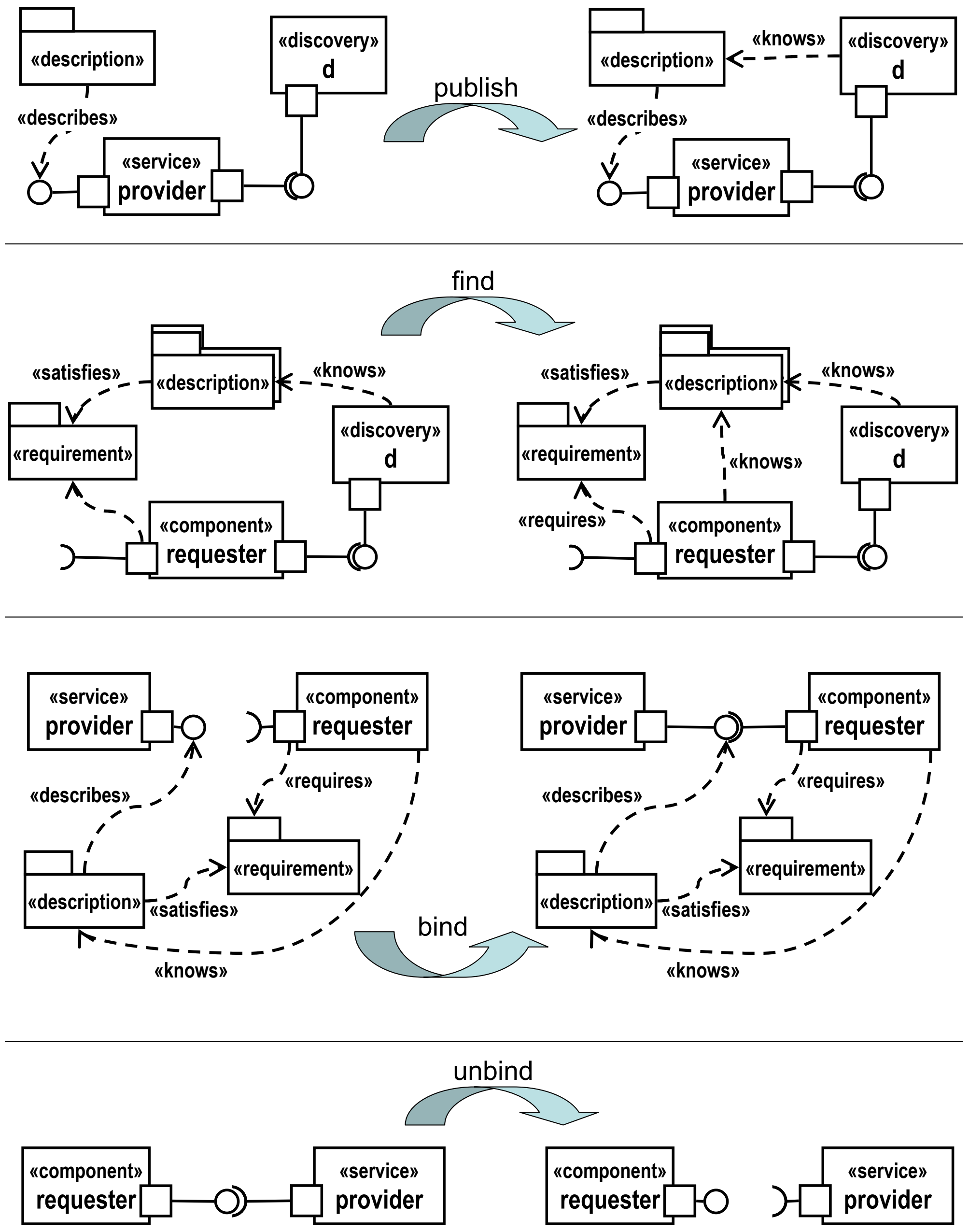

Fig. 4. Reconfiguration rules of the service-oriented architectural style

tems mentioned above, we would like to formulate the refinement of an abstract path as a reachability problem in the concrete transition system. However, the condition for behavior refinement includes the structural refinement of G_i by G'_i which, in general, requires to project the concrete graph to the abstract level in order to verify the desired inclusion.

In order to express the same property solely at the level of the concrete system, we must assume a second mapping $trans : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$, contravariant to abstraction. It translates an abstract instance graph into a concrete one representing the reformulation of the abstract state over the concrete type system. Note that the concrete graph does not necessarily represent a complete state of the concrete model, but rather a minimal *pattern* which has to be present in order for the requirements of the abstract graph to be fulfilled. Thus, we consider a concrete instance graph as a valid refinement of an abstract one if it contains this pattern as a subgraph, formally $trans(G) \subseteq G'$.

For example, Fig. 5 shows how the platform-independent model of the travel system is translated into a pattern for the service-oriented style with services instead of components where desired. According to the definition above, a valid service-oriented architecture containing this pattern, e.g., the SOA model at the bottom of Fig. 1, is a refinement of the abstract model.

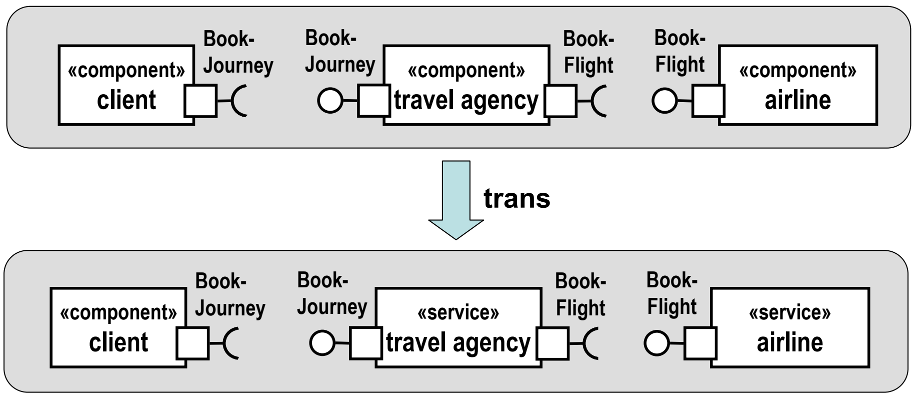

Fig. 5. Translation from platform-independent to service-oriented style

To make sure that the above condition is equivalent to the original one for structural refinement, we have to ensure the mutual consistency of the two contravariant mappings. This is formally expressed as a *satisfaction condition*, reminiscent of similar conditions in algebraic specification or logics, i.e.,

$$trans(G) \subseteq G' \text{ iff } G \subseteq abs(G').$$

In this case, we say that the two mappings are compatible.

Under this assumption, refinement can be formulated as follows. Concrete graph G' *refines* abstract graph G if

- $trans(G) \subseteq G'$
- for every transformation step $G \Rightarrow H$ in the abstract system there exists a transformation *sequence* $G' \overset{*}{\Longrightarrow} H'$ such that H' refines H.

It follows from the satisfaction condition that the first clause above is equivalent to the original condition $G \subseteq abs(G')$, expressed in terms of abstraction. However, the new condition can be verified solely at the concrete level.

The second clause is effectively a co-inductive definition of a simulation relation. Spelled out in terms of sequences, it says that for every (possibly infinite) path $G \Rightarrow G_1 \Rightarrow G_2 \Rightarrow \ldots$ in the abstract system there exists a path $G' \overset{*}{\Longrightarrow} G'_1 \overset{*}{\Longrightarrow} G'_2 \overset{*}{\Longrightarrow} \ldots$ in the concrete system with $trans(G_i) \subseteq G'_i$.

4 Related work

The use of meta-models for defining graphical languages has become very popular in the context of the *Meta-Object-Facility (MOF)* authored by the OMG. They also define meta-models as type graphs with additional constraints like, e.g., cardinalities. A model is an instance of the meta-model, if it conforms to the type graph.

In our work, we extend the static declaration of the meta-model by graph transformation rules which allow the definition of dynamic model evolutions as a simulation of system evolution. The use of graph transformation techniques to capture dynamic semantics of models has been inspired by Engels et al. in [4]. That approach extends meta-models defining the abstract syntax of a modeling language like UML by graph transformation rules for describing changes to object graphs representing the states of a model.

In [2], we have already considered several levels of platform abstraction that allow an MDA-like refinement from platform-independent architectures to more platform-specific ones. This has brought us to the question of suitable notions for refining graphs and graph transformation behavior: While structural refinement implies a relation between the involved type graphs, the idea for behavioral refinement is to relate the transformation rules of the involved graph transformation systems. In general, one can place these refinement relationships in a continuum from syntactic to semantically defined relations.

Große-Rhode et. al. [5], for instance, propose a refinement relationship between abstract and concrete rules that can be checked syntactically. One of the conditions requires that, e.g., the abstract rule and its refinement must have the same pre- and post-conditions. Based on this restrictive definition they can prove that the application of a concrete rule expression yields the same behavior as the corresponding abstract rule. The draw-back of this approach is that it cannot handle those cases where the refining rule expression should have additional effects on elements of the concrete level that do not occur in the abstract rule. And, the approach does not allow for alternative refinements of the same abstract rule depending on the context of its application.

Similarly, the work by Heckel et. al. [6] is based on a syntactical relationship between two graph transformation systems. Although this approach is less restrictive as it allows additional elements at the concrete level, it is still difficult to apply if there are no direct correspondences between abstract and concrete rules. Moreover, their objective is to project any given concrete transformation behavior to the abstract level, and not vice versa as in our case. Thus, refinement means a restriction of behavior rather than its extension.

In our work, we propose a more flexible, semantically defined notion of refinement. We do not require a fixed relation between transformation rules but only between the structural parts of the graph transformation system. Then, we check whether selected system states in the abstract system are also reachable at the concrete level, no matter by which sequence of transformations. By avoiding the functional mapping between rules, we can also relate transformation systems with completely different behavior, and we are flexible enough to cope with alternative refinements.

5 Conclusion

We have discussed semantic conditions for the refinement of dynamic models expressed as instances of graph transformation systems. Applications of this technique include so far the refinement of architectural models based on corresponding relations between architectural styles.

We are planning to support the approach by a coupling of CASE tools with editors and analysis for graph transformation systems, presently conducting experiments with existing model checkers.

References

[1] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 03*, pages 68–77. ACM Press, 2003.

[2] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proc.* 4^{th} *Working IEEE/IFIP Conference on Software Architecture, WICSA4*, pages 155–164. IEEE, 2004.

[3] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.

[4] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.

[5] M. Große-Rhode, F. Parisi Presicce, and M. Simeoni. Spatial and temporal refinement of typed graph transformation systems. In *Proc. Math. Foundations of Comp. Science 1998*, volume 1450 of *LNCS*, pages 553–561. Springer, 1998.

[6] R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struct. in Computer Science*, 6:613–648, 1996.

[7] A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfalz, and B. Böhlen, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.

[8] D. Varró. Towards symbolic analysis of visual modeling languages. In *Proc. GT-VMT 2002 - Int. Workshop on Graph Transformation and Visual Modeling Techniques*, volume 72 of *ENTCS*, pages 57–70. Elsevier, 2002.