Graph Transformations for MDE, Adaptation, and Models at Runtime

Holger Giese, Leen Lambers, Basil Becker, Stephan Hildebrandt, Stefan Neumann, Thomas Vogel, and Sebastian Wätzoldt

System Analysis and Modeling Group
Hasso Plattner Institute at the University of Potsdam, Germany
[prename.surname]@hpi.uni-potsdam.de

Abstract. Software evolution and the resulting need to continuously adapt the software is one of the main challenges for software engineering. The model-driven development movement therefore aims at improving the longevity of software by keeping the development artifacts more consistent and better changeable by employing models and to a certain degree automated model operations. Another trend are systems that tackle the challenge at runtime by being able to adapt their structure and behavior to be more flexible and operate in more dynamic environments (e.g., context-aware software, autonomic computing, self-adaptive software). Finally, models at runtime, where the benefits of model-driven development are employed at runtime to support adaptation capabilities, today lead towards a unification of both ideas.
In this paper, we present *graph transformations* and show that they can be employed to engineer solutions for all three outlined cases. Furthermore, we will even be able to demonstrate that graph transformation based technology has the potential to also unify all three cases in a single scenario where models at runtime and runtime adaptation is linked with classical MDE. Therefore, we at first provide an introduction in graph transformations, then present the related techniques of Story Pattern and Triple Graph Grammars, and demonstrate how with the help of both techniques model transformations, adaptation behavior and runtime model framework work. In addition, we show that due to the formal underpinning analysis becomes possible and report about a number of successful examples.

1 Introduction

Software code does in principle not decay as hardware does and thus, it could be employed forever when the underlying hardware is timely replaced on a regular basis. However, Lehman [1,2] observed and documented in his *laws of software evolution* that unless continuously being adapted, the typical software becomes less and less useful over time. Parnas [3] referred to this phenomena as *software aging* and identified two sources of the problem. (1) *lack of movement* when a software is not changed according to changing needs and (2) *ignorant surgery* which is caused by improper changes that are made to the software. Therefore,

This is a post-peer-review, pre-copyedit version of an article published in Lecture Notes in Computer Science Vol. 7320. The final authenticated version is available online at: https://doi.org/10.1007/978-3-642-30982-3_5

a steady deterioration of the value and quality of the software can be observed unless special action is taken and nowadays software is continuously adapted, which is referred to as *software evolution* [4].

Today, the majority of the costs for software are resulting from *adaptation steps*[1] that happen after the software has been first shipped. In the related *maintenance* [5] effort, versions of the shipped software are adapted in a *construction environment* in parallel to deploying the software in potentially many *runtime environments*. In addition to standard maintenance activities, oftentimes *reengineering* including *reverse engineering* [6] to recover necessary higher level information and *redesign* to improve the inner structure of the software (a popular approach for that is *refactoring* [7]) are employed to counteract the aging. A today often highly automated *distribution* activity then transports the adaptation developed and tested in the construction environment to the different runtime environments.

In addition to the code, software systems today also include configuration data and require a dedicated deployment capturing the mapping of the software components on the available hardware and software platforms in the *runtime environment*. Here, an even stronger demand for continuous adaptation has been observed. As the required adaptation steps have to be handled by the administrators of each individual runtime environment, it does not seem economically feasible in the long run to realize all the required adaptation steps manually and the *autonomic computing* initiative therefore advocates their automation (cf. [8]).

Furthermore, for an increasingly important class of software holds that the required adaptation steps have to happen for each runtime environment and according to the individual context that is only known at runtime. Therefore, the required adaptation steps have to be done in the runtime environment and can only be at most pre-planned in the construction environment (cf. *context-aware computing* [9]). In addition, today's software has to operate in more dynamic organizations and contexts and is often expected to be more versatile, flexible, and resilient. Also it is often envisioned that the software is dependable, robust, continuously available, energy-efficient, recoverable, customizable, self-healing, configurable, or self-optimizing by adapting itself in response to changing requirements and contexts. In all these cases, adaptation steps have to be supported for the runtime environment and have to be initiated by the software itself.

Besides this trend towards context-aware and more versatile software, also the integration of beforehand isolated software islands into extremely complex systems-of-systems, so-called *ultra-large scale systems* [10], leads to a situation where due to their size and complexity such systems are no longer managed by a single central authority. Moreover, for such systems, the structure resp. architecture is subject to changes at runtime and they have to be highly context-

[1] We use *adaptation* here in the broad sense such that it also includes corrective changes such as fixing faults and adding new or modifying existing features and not only making changes in existing software to accommodate it to a changing platform.

aware and to adjust themselves accordingly. Furthermore, the adaptation steps that are necessary for such systems can hardly be developed in the construction environment manually upfront but they have to be derived automatically at runtime.

To address this need for support of adaptation steps in the runtime environment, several approaches where the software itself takes care of the adaptation steps [11,12,13,14,15] have been proposed, which all can be united under the term *self-adaptive software* [16,17,18]. In general, self-adaptive software can be built by following the *internal approach* or the *external approach* [18]. The internal approach realizes self-adaptation capabilities by intertwining the adaptation logic and the application logic at the level of programming languages. Therefore, often programming languages features, like reflection [19], are employed. In contrast, the external approach separates the adaptation logic from the application logic by having a dedicated *adaptation engine* that controls the *core function* within the application. Most approaches for software engineering of self-adaptive systems today support the external approach (cf. survey [18]) and operate with a separation at the architectural level with well-defined interfaces between the adaptation engine and the core function. We refer to adaptation due to development or maintenance activities as *classical adaptation* in the following in order to clearly distinguish it form self-adaptation or in general adaptation.

As also emphasized in autonomic computing [8], not only the self-adaptation steps within the software but the complete feedback loops determining such self-adaptation steps in the form of monitor, analyze, plan and execute steps that happen within the runtime environment have to be taken into account when engineering self-adaptive software [20]. Studying the feedback is easier in case of the external approach. However, oftentimes today the feedback loops are not very visible in the architectures, but rather hidden (cf. [20]). Moreover, self-adaptive software often supports more than a single feedback loop. As an example, the reference architecture suggested in [21] distinguishes a *component layer* where the core functionality resides, a *change management layer* on top of that which manages the changes of the component layer, and a *goal management layer* that is responsible for the long term self-adaptation. Each of the two upper layers employs a feedback loop that steers the directly underlying layer.

Any solution that explicitly captures and analyzes the software and its context at a certain level of abstraction has to use runtime representations of them and thus uses runtime models. Otherwise, it can only consist of a simple case by case treatment in form of adaptation rules that immediately react to observed sensor inputs. While several approaches, like [22,23], employ runtime representations based on architecture description languages, a next step is to leverage the benefits of MDE for such runtime representations by means of *models at runtime* (M@RT) that are built on MDE principles as argued in [24].

Thus, we can conclude that in order to address the evolution challenge, a solution is required that supports adaptation steps initiated both in the construction environment (classical adaptation) as well as in the runtime environment (e.g., context-aware software, autonomic computing, self-adaptive software). As

pointed out in [25] the clear boundary between both cases already starts to disappear. Furthermore, as we pointed out already earlier in [26], a solution is required where adaptation steps in the construction environment and in the runtime environment happen in an integrated manner.

To address the adaptation challenge thus an approach is required that is able to cover *model-driven engineering* (MDE) that supports adaptation steps in the construction environment but also has to lay the foundation for later adaptation steps in the runtime environment, *modeling structural dynamics* that is the foundation for advanced adaptation steps in the runtime environment that goes beyond parameter adaptation, and *models at runtime* that support advanced adaptation steps in the runtime environment by providing a means to represent and handle complex information about the context as well as the system itself as a basis for adaptation decisions.

In this paper, we present *graph transformations*. We show that they can be employed to engineer the required class of systems with adaptation in the construction environment and runtime environment. We will be able to demonstrate that graph transformation based technology has the potential to also cover all three areas with a single formalism such that models at runtime and runtime adaptation can be linked straight forward with classical MDE. Furthermore, we show that due to the formal foundation of graph transformation sound analysis techniques such as conflict detection, invariant checking, and model checking can be applied.

In contrast to [27] introducing graph transformation from a more general software engineering perspective and in contrast to [28] that emphasizes the general benefits of graph transformations compared to other formalisms, we focus in this paper on the particular needs when approaching evolution by supporting MDE, modeling structural dynamics, and models at runtime.

Besides graph transformations, we will in particular present the related techniques of Story Patterns and Triple Graph Grammars, and how with the help of both model transformations, adaptation behavior and runtime model framework work. In addition, we show that due to the formal underpinning analysis becomes possible and report about a number of successful examples.

To exemplify the benefits of graph transformations for MDE and modeling adaptation, we will use the following two running examples.

Example 1 (RailCab). RailCab is a research project at the University of Paderborn, Germany addressing autonomously driving shuttles on regular railway tracks. The shuttles operate like cabs on request and not according to timetables. An important feature is the creation of convoys where the shuttles are not mechanically coupled but drive only with a short distance to each other. This reduces drag and thus permits to saves energy (cf. [29]). Networking and software should further ensure the safe operation and high system efficiency. A small test track has been setup to show the existing prototypes.[2]

Example 2 (SDL). The Specification and Description Language *(SDL) [30] is a specification language targeted the specification and description of reactive and distributed systems. We restrict our attention here to a simplified version for the block diagrams*

[2] `http://nbp-www.upb.de`

of SDL covering mostly structure and communication. A system consists of a number of blocks. Blocks communicate with each other using channels. A block further consists of processes that are the carrier of behavior. The later in this context considered model transformation is a simplified version of a transformation used in the industrial case study on flexible production control systems [31] from SDL block diagrams to UML class diagrams.

The paper explores how graph transformation fulfills the needs of engineering MDE solutions, engineering solutions with adaptation and even engineering solutions that combine both in form of solutions that adapt with the help of models at runtime as follows: At first we introduce graph transformation in Section 2. This introduction intuitively defines how the different forms of graph transformation such as graph transformation systems and graph grammars work together with a definition of their semantics based on set theory. Then, we introduce the concrete graph transformation based languages of Story Patterns, Triple Graph Grammars and a Runtime Model Framework in Section 3. Besides defining their syntax and semantics of the languages based on the beforehand introduced graph transformations, this also includes detailed examples. However, the introduced graph transformations are not only a means for specification and execution. As outlined in Section 4, we can take benefit of available analysis techniques. Finally, we discuss the state-of-the-art for MDE solutions, engineering solutions with adaptation and even engineering solutions that combine both in form of solutions that adapt with the help of models and the benefits graph transformation based techniques offer in Section 5. Afterwards, the paper closes with some final conclusions.

2 Graph Transformations

There are plenty examples where *annotated graphs* are a natural representation of the states of a system. Let us for instance consider the RailCab system of Example 1.

Example 3 (RailCab - Topology). A core element of the RailCab system is its track topology which resides in a 2-dimensional space but is most appropriately represented as a graph that abstracts from the geometric details. Also the shuttles are distributed over a 2-dimensional space, but what again matters is how their position is relative to the track topology. When shuttle build convoys they build new structures which again are best represented at an abstract level using graphs. Fig. 1 summarizes this analogy between a complex RailCab system and graphs and graph transformation systems. As depicted in Fig. 2 we can also further equip the graphs with attributes to store additional information about the available energy in the batteries of the shuttles.

We will see in the following that this analogy does not only hold for the state, but that also the behavior can accordingly be captured using graph transformations that describe which changes to the state represented as a graph will or can happen.

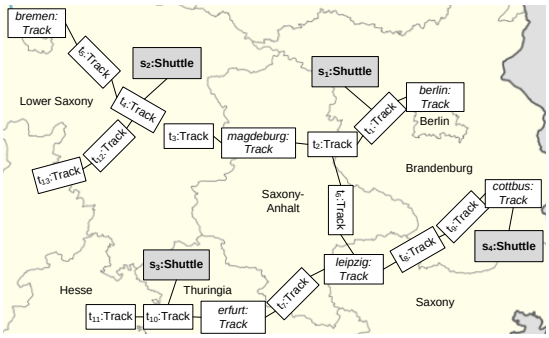


Fig. 1. A simple graph capturing a RailCab topology and the distribution of the shuttles on a map

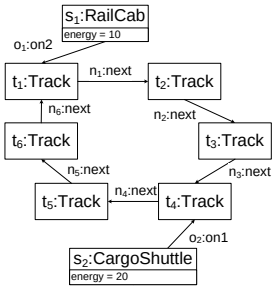


Fig. 2. An attributed graph for the RailCab topology and shuttles

Graphs, Type Graphs, and Graph Morphisms As outlined *graphs* can be used to represent a particular state of a system in a formal way. Also the abstract syntax of (visual) models can be captured by graphs. Thereby, graphs occur at two levels: the type level and the instance level. A fixed *type graph* TG serves as a representation of the combination of node types and edge types that may occur in graphs at the instance level. In particular, instance graphs of a type graph are equipped with a structure-preserving mapping (i.e. a *graph morphism*) to the type graph. First, we introduce graphs and graph morphisms with different useful properties in a formal way. Then, we introduce the notion of typed graphs formally.

Definition 1 (graph and graph morphism). *A* graph $G = (G_V, G_E, s, t)$ *consists of a set* G_V *of vertices, a set* G_E *of edges and two total mappings* $s, t : G_E \to G_V$*, assigning to each edge* $e \in G_E$ *a source* $s(e) \in G_V$ *and target* $t(e) \in G_V$*. A* graph morphism $f : G_1 \to G_2$ *between two graphs* $G_i = (G_{i,V}, G_{i,E}, s_i, t_i)$*,* $(i = 1, 2)$ *is a pair* $f = (f_V : G_{V,1} \to G_{V,2}, f_E : G_{E,1} \to G_{E,2})$ *of total mappings, such that* $f_V \circ s_1 = s_2 \circ f_E$ *and* $f_V \circ t_1 = t_2 \circ f_E$*.*

Graph morphisms may satisfy different useful properties. A graph morphism that does not map two nodes or two edges to the same node or edge, respectively, satisfies the so-called *injectivity* property. A graph morphism defining a preimage for each node and edge of the target graph satisfies the *surjectivity* property. Two graph morphisms having the same target graph defining for each node and edge of the target graph a preimage in at least one of both source graphs are called *jointly surjective*. In this case, we also say that the target graph is an *overlapping* of both source graphs. A graph morphism being both injective and surjective is also called a *graph isomorphism*. It uniquely maps all nodes and edges of source and target graphs to each other. Consequently, trivially speaking, isomorphic graphs are copies of each other, whereas an injective graph morphism finds a copy of the source graph somewhere in the target graph.

Definition 2 (injective, (jointly) surjective morphisms, graph isomorphism). *A graph morphism $m : G_1 \to G_2$ is* injective (resp. surjective) *if m_V and m_E are injective (resp. surjective) mappings. Two graph morphisms $m_1 : L_1 \to G$ and $m_2 : L_2 \to G$ are* jointly surjective *if $m_{1,V}(L_{1,V}) \cup m_{2,V}(L_{2,V}) = G_V$ and $m_{1,E}(L_{1,E}) \cup m_{2,E}(L_{2,E}) = G_E$. A pair of jointly surjective morphisms (m_1, m_2) is also called an* overlapping *of L_1 and L_2. A graph morphism m which is injective and surjective is called a* graph isomorphism. *Two graphs G_1 and G_2 are* isomorphic *if there exists a graph isomorphism $m : G_1 \to G_2$.*

Definition 3 (typed graph). *A* type graph *is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$. V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively. A tuple $(G, type)$ of a graph G together with a graph morphism $type : G \to TG$ is a* graph typed over TG *or* instance graph *of TG.*

Example 4 (RailCab - Typed Graph). Fig. 3 depicts a graph G typed over the type graph TG via the typing morphism $type : G \to TG$. G consists of a set of nodes $G_V = \{s_1, t_1, t_2, t_3\}$ and a set of edges $G_E = \{o_1, n_1, n_2\}$. The source and target mappings s_G and t_G map these edges to the respective source and target nodes as depicted. For example, $s_G(n_1) = t_1$ and $t_G(n_1) = t_2$. The typing morphism $type : G \to TG$ is visualized using dashed arrows. Analogously, TG consists of a set of nodes $TG_V = \{\mathsf{Shuttle}, \mathsf{Track}\}$ and a set of edges $TG_E = \{\mathsf{on}, \mathsf{next}\}$ where s_{TG} and t_{TG} map these edges to the respective source and target nodes as depicted. In particular, type is a graph morphism since it is structure-preserving. This means, for example, for edge o_1 that $s_{TG}(type_E(o_1)) = Shuttle = type_V(s_G(o_1))$. Note that this typing morphism type is surjective, since each node and edge in TG has a preimage in G. However type is not injective, since, for example, the nodes t_1, t_2 and t_3 are mapped to the same node $Track$. Fig. 4 depicts a short-hand notation for a typed graph that we will use in the rest of the paper. The typing morphism between graph and type graph is not explicitly depicted anymore. Instead, each node and edge name is followed by ":" and then the type name of the node type or edge type, the typing morphism assigns the node or edge to.

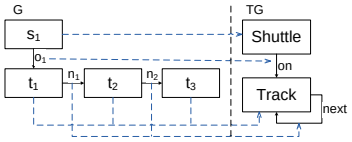


Fig. 3. Typed Graph

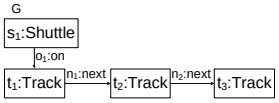


Fig. 4. Typed Graph (shorthand notation)

Typed graph morphisms formalize the concept of structure-preserving mappings compatible with typing. Therefore, they are a formal means to ensure type correctness later on when performing graph transformations.

Definition 4 (typed graph morphism). *Consider typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2, type_2)$, a* typed graph morphism *$f : G_1^T \to G_2^T$ is a graph mor-*

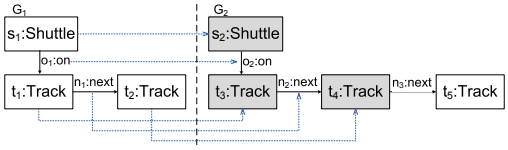


Fig. 5. The example graph morphism f

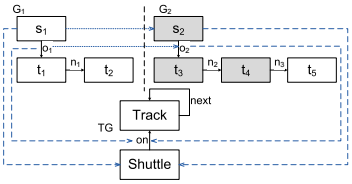
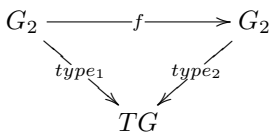


Fig. 6. Type-compatibility of f

phism $f : G_1 \to G_2$ *such that* $type_2 \circ f = type_1$.



Example 5 (RailCab - Typed Graph Morphism). Fig. 5 depicts an injective typed graph morphism f from $(G_1, type_1)$ *to* $(G_2, type_2)$. *The pointed edges visualize f. Note that the graph morphism f is indeed type-compatible because each node or edge of a specific type is mapped to a node or edge of the same type, respectively. Fig. 6 depicts an extract of the same morphism f and an extract of the typing morphisms* $type_1$ *and* $type_2$ *illustrating more formally that according to Def. 4* $type_2 \circ f = type_1$.

Assumption: For the rest of this paper we work with typed graphs and morphisms, although not always explicitly mentioned. This means also that we denote a typed graph $(G, type)$ also simply as G. Moreover, if the edge mapping of graph morphisms is clear from the respective source and target node mappings, then we do not always visualize them completely in the corresponding figures.

Graph Patterns and Graph Properties Graph patterns describe sample graphs for which matches may exist for given instance graphs. We present a simple pattern concept, which is used and supported in most of our graph transformation tools. It consists of a graph P and a set of graphs N_i containing P (with identical typing). We say that a match for this pattern in graph G exists if a copy of P can be found in G, but at the same time no copy for any of the graphs N_i can be found in G.

Definition 5 (graph pattern). *A graph pattern* $\Pi = (P, \{N_i, i \in I\})$ *consists of a graph P and a finite set of graphs* N_i *containing P as a subgraph. As shorthand notation for the graph pattern* $(P, \emptyset)$ *we simply write P.*

Definition 6 (match). *Given a graph pattern* $\Pi = (P, \{N_i, i \in I\})$ *and a graph G, then each injective morphism* $m : P \to G$ *such that there does not exist an injective morphism* $q : N_i \to G$ *with q being identical to m on P, is called a* match *of the graph pattern* Π *in G.*

To visualize N_i we use crossed out dashed boxes and edges. We draw a dashed box around all nodes of $N_i \setminus P$. Also all edges which source and target nodes

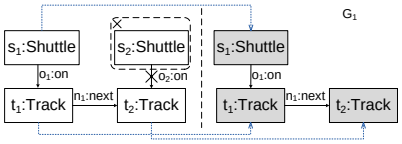


Fig. 7. Pattern Π matches G_1

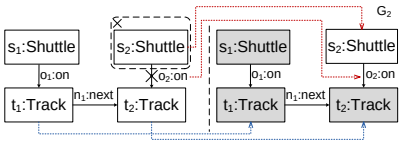


Fig. 8. Pattern Π does not match G_2

are in $N_i \setminus P$ are contained in the box. All edges connecting P and $N \setminus P$ are not contained in the box and in addition crossed out. In the special case that N_i equals a single edge, it is also only crossed out.

Example 6 (RailCab - Graph Pattern). Fig. 7 depicts the pattern $\Pi = (P, \{N\})$. The graph P consists of the nodes s_1, t_1, t_2 and edges o_1, n_1. The graph N consists of P together with the crossed out node s_2 and edge o_2. There exists a match m for Π in G_1, since an injective graph morphism (visualized by pointed lines) exists between P and G_1 and no injective graph morphism exists between N and G_1, since a second node of type Shuttle is not available.

Fig. 8 depicts the same pattern $\Pi = (P, \{N\})$, but a different graph G_2 such that no match exists for Π in G_2. There exists one injective graph morphism from P to G_2, but this graph morphism can be completed to an injective graph morphism from N to G_2, which is not allowed according to Def. 6.

We use graph patterns as basic constructs to define graph properties (also called graph constraints or graph conditions [32,33]). As explained in [32] graph properties may reach the expressiveness of first-order logic, which is not the case here, since we have a more restricted property language.

Definition 7 (graph property, forbidden and required pattern). *A graph pattern $\Pi = (P, \{N_i, i \in I\})$ is a* graph property*, any combination of two graph properties p and q of the form $p \wedge q$, $p \vee q$, and $\neg q$ is also a graph property. We define* satisfaction *of graph properties p by a graph G (written $G \models p$), recursively, as follows:*

- *If $p = \Pi$ with $\Pi = (P, \{N_i, i \in I\})$ a graph pattern, then p is satisfied if there exists a match for the graph pattern Π in G,*
- *if $p = p_1 \wedge p_2$, then p is satisfied if $G \models p_1$ and $G \models p_2$,*
- *if $p = p_1 \vee p_2$, then p is satisfied if $G \models p_1$ or $G \models p_2$,*
- *if $p = \neg p_1$, then p is satisfied if $G \not\models p_1$.*

Given a graph property $p = \Pi = (P, \{N_i, i \in I\})$ we say that Π occurs as a required graph pattern*. For a graph property $p = \neg\Pi = \neg(P, \{N_i, i \in I\})$ we further say that Π occurs as a* forbidden graph pattern*.*

Example 7 (RailCab - Graph Properties). Given the property $p = \Pi = (P, \{N\})$ with Π the pattern depicted in Fig. 7, then this pattern occurs as a required graph pattern. $G_1 \models p$ since there exists a match for the required graph pattern Π in G. In Fig. 8, a graph G_2 is depicted which does not satisfy p since a match for the required graph pattern Π does not exist.

Given the property $p' = \neg \Pi$, then Π occurs as a forbidden pattern and $G_1 \not\models p'$ since $G_1 \models p$ and $G_2 \models p'$ since $G_2 \not\models p$. Consequently, G_1 and G_2 both satisfy $p \vee p'$, but not $p \wedge p'$.

Graph Transformation Rules We can model the modification of graphs by introducing the *graph transformation* approach. It is a rule-based approach, meaning that the way in which a graph can potentially be modified is described by a set of graph transformation rules. By applying these rules to a particular graph, this graph can be transformed. We present a compact, set-theoretical description of graph transformation here and refer to [34,35] for a more comprehensive description with category-theoretical background.

We start with defining the notion of *graph transformation rules*. A rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ consists of a left-hand side (LHS) pattern Π_{LHS} describing the pre-condition, and a right-hand side (RHS) pattern Π_{RHS} describing the post-condition of the rule. In simple rules, the patterns Π_{LHS} and Π_{RHS} are just graphs, L and R, denoting required patterns before and after rule application, respectively. As a consequence, before applying the rule to a graph G, at least L should be present in G, which is replaced by R via the rule's application. In particular, the graph part $L \setminus (L \cap R)$ is to be deleted, and the graph part $R \setminus (L \cap R)$ is to be created when applying the rule. Finally, $L \cap R$ describes which part is to be preserved, when applying the rule. Note that the *graph intersection* $L \cap R$ should form a *well-defined* typed graph again. To this extent the source and target mappings in L and R must be identical on edges belonging to $L \cap R$ such that source and target mappings for $L \cap R$ can be inherited from L and R. Moreover, the type mappings for L and R must be identical on nodes and edges in $L \cap R$ such that the type mapping in $L \cap R$ can be inherited from L and R. The LHS pattern of a rule Π_{LHS} can be also a pattern of the form $(L, \{N_i, i \in I\})$ instead of the simple pattern L. Thus the pattern $(L, \{N_i, i \in I\})$ instead of L is required before rule application. In this context, we say that N_i are the *negative application conditions* (NACs) of the rule r, since the rule can only be applied if a copy of L, but no copies of N_i can be found before rule application.

Definition 8 (rule). *A graph transformation* rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *consists of a rule name r and two patterns $\Pi_{LHS} = (L, \{N_i, i \in I\})$ and $\Pi_{RHS} = R$ with L and R graphs such that the intersection $L \cap R$ of L and R is well-defined. The patterns Π_{LHS} and Π_{RHS} are called the left-hand side (LHS), and the right-hand side (RHS) of r, respectively. We say that $del(r) = L \setminus (L \cap R)$ is the graph part to be deleted and $cre(r) = R \setminus (L \cap R)$ is the graph part to be created by the rule r.*

There are two main *different ways to define rule application* of a rule r to a graph G as soon as a match for the LHS pattern of r in G has been found. One of both rule application approaches can be chosen to perform graph transformation depending on if implicit side-effects are desired or not.

The first main approach *accepts implicit side-effects* such as the deletion of dangling edges. It deletes dangling edges during rule application although this

is not explicitly specified within the rule. This approach has been called the single-pushout (SPO) approach for historical reasons. In particular, a rule application (also called direct graph transformation or graph transformation step) can be formalized in a categorical way by a so-called pushout – a categorical concept generalizing the idea of graph gluing constructions – in the category of graphs with partial graph morphisms [34]. Here we reintroduce this rule application approach in a constructive, set-theoretical way and propose to call it the *dangling-edge-collecting approach*.

The second main approach does not put up with implicit side-effects. It simply does not apply a rule – even if a match has been found – if it is not possible to apply the rule without the implicit side-effects that dangling edges are removed. This is ensured by the fact that a match in this approach needs to satisfy in addition the so-called *dangling edge condition* – expressing that nodes marked for deletion by the rule are matched in such a way that all incident edges are marked for deletion by the rule as well. Like this no dangling edges arise during rule application. This more conservative approach to rule application has been called the double-pushout (DPO) approach for historical reasons. In particular, a rule application can be described formally in a categorical way by a construction consisting of two pushouts in the category of graphs with total graph morphisms [35].[3] Here we reintroduce this rule application approach in a constructive, set-theoretical way and propose to call it the *conservative approach* since no implicit side-effect during rule application is allowed. [4]

Definition 9 (dangling edges, dangling edge condition). *Given a rule* $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *and match* $g : L \to G$ *for the pattern* $\Pi_{LHS} = (L, \{N_i, i \in I\})$ *in* G*, then* $dan(g, r) = \{e | e \in G_E, s(e) \vee t(e) \in g(del(r)), e \notin g(del(r))\}$ *is the* set of dangling edges *in* G *for match* g *and rule* r*. The match* g *fulfills the* dangling edge condition *for rule* r *if* $dan(g, r)$ *is empty.*

Definition 10 (rule applicability). *A rule* $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *with* $\Pi_{LHS} = (L, \{N_i, i \in I\})$ *and* $\Pi_{RHS} = R$ *is* applicable *to a graph* G *in the* conservative approach *if there exists a match* $g : L \to G$ *for* $(L, \{N_i, i \in I\})$ *in* G *fulfilling the dangling edge condition.*

A rule $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *with* $\Pi_{LHS} = (L, \{N_i, i \in I\})$ *and* $\Pi_{RHS} = R$ *is* applicable *to a graph* G *in the* dangling-edge-collecting approach *if there exists a match* $g : L \to G$ *for* $(L, \{N_i, i \in I\})$ *in* G*.*

After having found a match g for the LHS rule pattern of rule r in graph G making the rule applicable, we can define a rule application via rule r to G by

[3] The left pushout of a rule application describes the deletion of graph parts, and the right pushout describes the addition of graph parts, marked accordingly by the corresponding rule.

[4] Note that a match of a LHS rule pattern does not have to be, in general, an injective graph morphism. In some application fields, it makes sense to allow non-injective graph morphisms as matches. In this case however, rule application becomes more difficult because a conflict arises when a match maps two graph elements in L, one marked for deletion and the other one marked for creation by the rule, to the same element in G.

a two-step construction such that in the application result the RHS rule pattern is fulfilled: First, the elements in $del(r)$ are deleted from G together with the implicit deletion of possible dangling edges $dan(g, r)$ obtaining an intermediate result D. Secondly, a copy of the RHS pattern graph R is unified with D such that exactly the elements in $cre(r)$ are indeed created. Thereby nodes and edges in $L \cap R$ to be preserved are glued with the already corresponding elements in D matched via g. Therefore this construction is often also called gluing construction.

Definition 11 (rule application). *A* rule application $G \overset{r,g}{\Rightarrow} H$ *from* G *to* H *via an applicable rule* $r : \langle \Pi_{LHS}, \Pi_{RHS} \rangle$ *with* $\Pi_{LHS} = (L, \{N_i, i \in I\})$ *and* $\Pi_{RHS} = R$ *and match* $g : L \to G$ *is constructed as follows:*

1. $D = G \setminus (g(del(r)) \cup dan(g, r))$ *(delete nodes and edges to be deleted together with possible dangling edges)*
2. $H = D \cup i(R)$ *with* $i : R \to i(R)$ *a graph isomorphism identical to* g *on elements of* $L \cap R$ *and disjoint with* D *on elements in* $cre(r)$ *(create nodes and edges to be created).*

Each graph H' *isomorphic to* H *is a valid result of this rule application too.*

Note that a rule which is only applicable in the conservative approach will be applied without implicit side-effects, since in this case the set of dangling edges is empty because each match fulfills the dangling edge condition. Moreover, note that because of the fact that dangling edges are deleted, D is a well-defined graph again since source and target mappings can be inherited from G. The application result H is a graph again as well, since source and target mappings in D or $i(R)$ are identical on edges belonging to $D \cap i(R)$. This is because the graph morphisms g and i are identical on $L \cap R$.

We omit r and/or g in $G \overset{r,g}{\Rightarrow} H$ if not relevant. As a last remark, note that the typing of H can be inherited from the typing of elements stemming from G (i.e. being left in D) and the typing of created elements in rule r because of type compatibility of g,i and rule r. This means that by construction rule application ensures *type correctness*.

Example 8 (RailCab - Graph Transformation Rule and Rule Application). Fig. 9 depicts the application of rule r_1 *to a graph* G_1*. It is a simple rule, since the LHS pattern consists of a single graph* L*. Rule* r_1 *is applicable in the conservative as well as the dangling-edge-collecting approach, since a match* $g : L \to G_1$ *can be found, depicted with pointed lines. The rule can be applied in the conservative as well as in the dangling-edge-collecting approach, since the depicted match fulfills the dangling edge condition. In particular, this holds already because no node is deleted. The result of the rule application is therefore the same in both approaches. First,* $g(del(r_1))$ *consisting of* o_3 *as image of* o_1 *in* G_1 *is deleted from* G_1 *leading to a graph* D*. A copy of the RHS graph is then unified in a suitable way with* D*. This means that the elements* s_1, t_1, t_2, n_1 *in* $L \cap R$ *are mapped by an isomorphism* i *identical to* g *inducing the gluing of* $i(R)$ *with* D *in the elements* s_2, t_3, t_4, n_2*. Moreover, a copy* $o_4 = i(o_2)$ *of* o_2*, belonging to*

$cre(r_1)$ is indeed created, since it is added disjointly to D[5] and glued with source node $i(s_1) = g(s_1) = s_2$ and target node $i(t_2) = g(t_2) = t_4$.

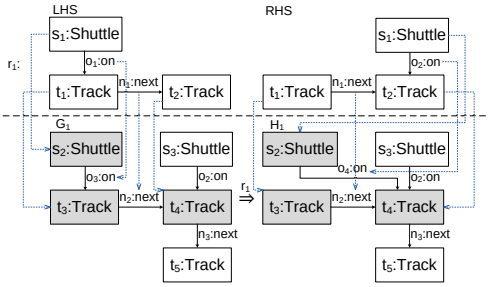


Fig. 9. SimpleMove rule and its application

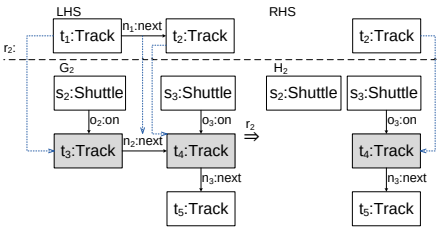


Fig. 10. DeleteTrack rule and its application with unwanted dangling edge deletion

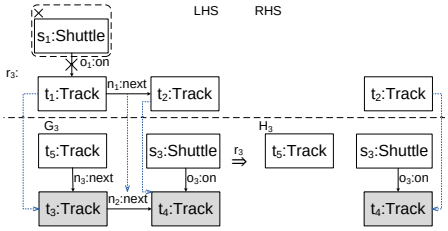


Fig. 11. Corrected DeleteTrack rule and its proper application

Fig. 10 depicts the application of the DeleteTrack rule to a graph G. The rule can only be applied in the dangling-edge-collecting approach, since the depicted match $g : L \to G_2$ does not fulfill the dangling edge condition. This is because $dan(g, r_2) = \{o_2\}$, since o_2 is an edge which is not matched by g, but its target node t_3 is matched by g and identified as a node to be deleted. When applying this rule in the dangling-edge-collecting approach, this means that o_2 is implicitly deleted together with t_3 and n_2. This has as a consequence that the node s_2 of type Shuttle would not be on a track anymore and thus decoupled of the modeled track system. Forbidding the deletion of a Track if some Shuttle is still on a track would make more sense. To this extent, it is possible to add a negative application condition to the LHS rule pattern expressing that it can be applied only if no Shuttle is on the Track. In this case, only edges of type next are implicitly deleted during rule application as can be seen in Fig. 11.

Besides a single application we are also interested in the effect of multiple rule applications. Therefore, we define graph transformation as the reflexive and transitive closure of separate rule applications.

[5] Since another edge called o_2 is present in D, this renaming via $i(o_2) = o_4$ is indeed necessary in this example.

Definition 12 (graph transformation). *A* graph transformation*, denoted as* $G_0 \overset{*}{\Rightarrow} G_n$*, is a sequence* $G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n$ *of* $n \geq 0$ *rule applications. A rule application of length* 0 *is defined as a graph isomorphism* $G_0 \cong G'_0$ *because the result of rule application is only unique up to isomorphism.*

Attributed Graph Transformation Since often besides the structure also attributes contained by the elements are relevant for modeling, we need a way to include attributes in graphs for the formal description of models. We do not introduce attributed graph transformation in a formal way here, but give a short overview on available formal approaches and describe the basic concepts needed to define *attributed graphs and attributed graph transformation.*

There are different approaches to define attributed graphs and graph transformation. In [36] attributed graphs are seen as algebras. In particular, the graph part of an attributed graph is encoded as an algebra, extending the given data algebra. In [35] an attributed graph is basically a pair (G, D) consisting of a graph G and a data algebra D, whose values are nodes in G. [37] is based on the use of labeled graphs to represent attributed graphs, and of rule schemata to define graph transformations involving computations on the labels. That approach has some similarities with the so-called symbolic graph transformation approach [38], including the simplicity provided by the separation of the algebra and the graph part of attributed graphs.

The basic concepts needed to define attributes on the type level and on the instance level are described as follows. For each node type (sometimes also edge types) in the type graph TG a *number of attributes* of a certain data type is defined leading to an *attributed type graph* ATG. Each node (or edge) in a graph on the instance level may have the same number of *attributes*. These have *attribute assignments* mapping each attribute to a concrete value of a data type compatible with the attribute definition of the corresponding node type (or edge type) in the attributed type graph ATG. Each graph in a graph pattern may be equipped with an *attribute condition* Φ over attribute labels in this graph constraining the range of possible values for these attributes when matching the pattern to some instance graph. Moreover, attribute assignment mappings in L of a LHS rule pattern may define assignments to variables that are reused within a computation instruction in an attribute assignment mapping for some attribute a of the RHS rule pattern. Matching the LHS pattern leads to a concrete value assignment of such a variable (respecting the attribute conditions) and this value is reused to compute the attribute value of a according to the computation instruction.[6] The attribute condition and assignment mappings need to be compatible with the data types defined in the attributed type graph for each attribute.

[6] In [38], assignments and attribute conditions in rule patterns are summarized into one attribute formula over the attribute labels in both rule patterns that needs to evaluate to true during rule application.

Graph Transformation with Inheritance Another concept often used in modeling is inheritance that leads to generalization upwards in the inheritance relation and specialization downwards leading to attributed graphs and attributed graph transformations *with inheritance* [39,35,40]. Again, we do not introduce this formally here, but give a short informal idea of the basic concepts needed to define attributed graphs and attributed graph transformation with inheritance.

The concept of generalization, specialization and inheritance can be described in a type graph TG by introducing an *inheritance relation* between nodes in the type graph, visualized by special edges from each type node to its super type node, which we label with $\mathsf{is\_a}$, and marking specific type nodes as abstract. Patterns typed over such a type graph with inheritance $ATGI$ consist of graphs that may use these abstract nodes. Moreover, source and target mappings are compatible with the inheritance relation. The created elements in the RHS pattern of a rule should not be abstract because when a rule is applied it should be clear which type of node is to be created on the instance level. Now there are two possibilities to define attributed graph transformation with inheritance according to such a type graph with inheritance $ATGI$ and rules and patterns typed as described briefly above over $ATGI$. (1) The type graph with inheritance $ATGI$ is flattened in a suitable way to an equivalent type graph TG without inheritance relation and abstract nodes. Moreover, the rules and patterns typed over $ATGI$ as described briefly above are *flattened* to an equivalent set of rules and patterns typed over TG. Using these flattened rules and patterns regular typed attributed graph transformation can be applied. (2) The *match notion* for patterns is *extended* to patterns typed over $ATGI$ such that the derived notions of rule application and property satisfaction are equivalent to flattened regular rule application or property satisfaction.

For analysis we usually apply variant (1) and work with flattened properties and rules, since most analysis techniques do not explicitly deal with inheritance yet. For rule application and graph property checking at runtime we usually apply the more efficient variant (2).

Graph Transformation with Priorities Non-determinism due to several applicable rules can be explicitly reduced by *priorities* over these rules. Given a rule set $\mathcal{R}$ with priorities specified by a function $\mathsf{prio} : \mathcal{R} \to \mathbb{N}$ assigning priorities to the rules in $\mathcal{R}$, the notion of rule applicability of Def. 10 defined for a separate rule becomes more severe and has to be defined relative to the complete rule set. We say that the rule is *applicable with priority* if for two rules $r, r' \in \mathcal{R}$ that are both applicable to the same graph if considered separately holds that if they have different priorities only the rule with the highest priority is applicable. Thus applicability with priority requires besides a match and the dangling edge condition in case of the conservative approach also that no rule with a higher priority is applicable as a separate rule. Given a set of rules $\mathcal{R}$ with priority function prio, we write $G \overset{r,g}{\Rightarrow}_{\mathcal{R},\mathsf{prio}} G'$ if for rule $r \in \mathcal{R}$ a match g for G exists, r is applicable with priority, and $G \overset{r,g}{\Rightarrow} G'$. For the reflexive and transitive closure we write $\overset{*}{\Rightarrow}_{\mathcal{R},\mathsf{prio}}$.

Assumption: For the rest of this paper we work with attributed typed graphs with inheritance and rules with priorities where necessary, although not explicitly mentioning it each time. Consequently, we sometimes write $G \overset{r,g}{\Rightarrow}_{\mathcal{R}} G'$ instead of $G \overset{r,g}{\Rightarrow}_{\mathcal{R},\mathsf{prio}} G'$ and $\overset{*}{\Rightarrow}_{\mathcal{R}}$ instead of $\overset{*}{\Rightarrow}_{\mathcal{R},\mathsf{prio}}$.

Example 9 (RailCab - Attributes, Inheritance & Priorities). As depicted in Fig. 12 we may equip the type graph of our running example with an attribute energy *of data type* Int *for the node type* Shuttle*. Moreover we can make the node type* Shuttle *abstract and insert two subtypes* RailCab *and* CargoShuttle *into the inheritance relation, respectively. An assignment in an attributed instance graph as depicted in Fig. 2 defines a concrete integer value for the attribute energy in nodes of node type* RailCab *or* CargoShuttle*. The type graph with inheritance in Fig. 12 can be flattened into a regular type graph as depicted on the right in Fig. 12.*

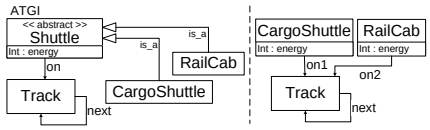


Fig. 12. Attributed Type Graph with Inheritance and Flattening

Fig. 13 depicts a graph pattern with inheritance P_I that can be flattened into four patterns without inheritance on the right. Note that the patterns P_1 and P_2 are isomorphic, so it is sufficient to keep one of these patterns after flattening.

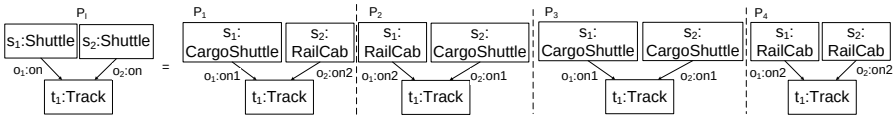


Fig. 13. A graph pattern and the related flattened graph patterns

Now our running example rule, moving a Shuttle *from one* Track *to another (see Fig. 9), can be flattened to two different rules by flattening the corresponding LHS and RHS rule patterns. In Fig. 14, the first rule is depicted and we have added an operation on the previously introduced attribute* energy*. The attribute value of the attribute* energy *is constrained such that in the instance graph to which the pattern can be matched to, a value bigger than or equal to* 2 *should appear. After rule application this attribute value is diminished by* 2*. In Fig. 15, a similar rule is depicted modeling the movement of a* Railcab *which is less expensive in the sense that the attribute value of* energy *is diminished only by* 1*, when moving the* Railcab *from one* Track *to another* Track*.*

In the example so far we do not need priorities. However, let us assume that the rule of Fig. 15 refers to the general case of a Shuttle *rather than a* RailCab *and thus defines that all shuttles by default require one energy point to move along one* Track*. Then, the rules of Fig. 14 and Fig. 15 could both be applied for* CargoShuttles *with* energy *attribute value higher than 1. To ensure that in case of a* CargoShuttle *always*

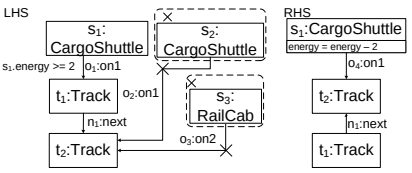


Fig. 14. MoveCargoShuttle rule with attribute condition and side effect

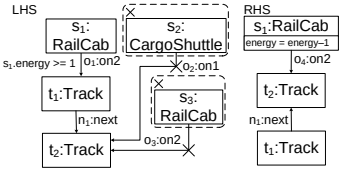


Fig. 15. MoveRailCab rule with attribute condition and side effect

only the more specific rule of Fig. 14 and not the generic one of Fig. 15 is applied, we can then assign the former rule of Fig. 14 a higher priority.

Graph Transformation Systems A dynamic system can be specified by a so-called graph transformation system. It consists of a set of graph transformation rules describing the dynamics in the system. Each system state is described by a graph and state transitions correspond to rule applications. Initial states of a dynamic system can be described by an initial graph or a set of initial graphs.

Definition 13 (graph transformation system). *A* graph transformation system *(GTS)* $S = (\mathcal{R}, TG)$ *consists of a set of rules* $\mathcal{R}$ *typed over a type graph* TG*. A graph transformation system may be equipped with an initial graph* G_0 *or a set of initial graphs* I *being graphs typed over* TG*.*

Note that the definition is analogous whatever type of type graph with inheritance and attributes or without is employed. Also, the rule set $\mathcal{R}$ may support priorities, which we do not always explicitly mention as described in the previous assumption.

The set of reachable graphs of a graph transformation system models the set of reachable states of a dynamic system from its initial states. A graph is reachable if a graph transformation via the system rules exists from some initial graph, describing some initial system state, to this graph. Since often in praxis it does not make sense to distinguish isomorphic graphs, we also define a minimal set of reachable graphs, where exactly one representative of the isomorphism class of each reachable graph is contained.

Definition 14 (set of reachable graphs). *For a GTS* $S = (\mathcal{R}, TG)$ *and a set of initial graphs* I *the* set of reachable graphs $\mathsf{REACH}(S, I)$ *is defined as* $\{G | G_0 \overset{*}{\Rightarrow}_{\mathcal{R}} G, G_0 \in I\}$ *consisting of all graphs* G *such that there exists a graph transformation via rules in* $\mathcal{R}$ *from some initial graph* G_0 *to* G *of arbitrary length. We say that* $\mathcal{G} \subseteq \mathsf{REACH}(S, I)$ *is a* complete *set of reachable graphs up to isomorphism for a GTS* S *and* I *if it contains at least one representative graph of each isomorphism class of graphs in* $\mathsf{REACH}(S, I)$ *and that it is a* minimal *set of reachable graphs up to isomorphism if it contains exactly one representative graph of each isomorphism class of graphs in* $\mathsf{REACH}(S, I)$*.*

Often, it is not only desired to analyze which system states can be reached, but also how they can be reached. The transition system generated by a graph

transformation system and its initial graphs therefore describes the state space of a dynamic system. If a distinction between isomorphic graphs (or states) is not desired, then it is possible to consider a minimal transition system, describing rule applications between the corresponding minimal set of reachable graphs.

Definition 15 (labeled transition system). *Given a GTS $S = (\mathcal{R}, TG)$, a set of initial graphs I, and a set of graphs $\mathcal{G} \subseteq \mathsf{REACH}(S, I)$ that is complete up to isomorphism for S and I, the implied* labeled transition system *$LTS = (\mathcal{G}, I, \mathcal{R} \times \mathcal{M}, \Rightarrow_{\mathcal{R}})$ with $\mathcal{G}$ the set of states, I the set of initial states, $\mathcal{R} \times \mathcal{M}$ the* label alphabet *with $\mathcal{M}$ the set of injective morphisms, and $\Rightarrow_{\mathcal{R}} \subseteq \mathcal{G} \times (\mathcal{R} \times \mathcal{M}) \times \mathcal{G}$ the* transition relation *defined as $\{(G, (r, g), H) | G, H \in \mathcal{G} \wedge \exists H' \in \mathsf{REACH}(S, I) : G \overset{r,g}{\Rightarrow}_{\mathcal{R}} H' \wedge H' \cong H\}$. LTS is* minimal *if its set of states $\mathcal{G}$ is a minimal set of reachable graphs up to isomorphism for the GTS S with initial graphs I.*

Example 10 (RailCab - GTS). The rules depicted in Fig. 14 and Fig. 15 typed over the flattened type graph ATG as depicted in Fig. 12 constitute a GTS modeling the structural dynamics and energy consumption of the shuttle system. Given also the attributed graph in Fig. 2 as initial graph, we can consider the corresponding set of reachable graphs and the corresponding transition system. They will have a finite minimal set of reachable graphs and minimal transition system, respectively. Since Shuttle *movement goes along with diminishing the energy attribute values of s_1 and s_2, this leads to a terminating system. Moreover, each reachable graph satisfies the property $p = \neg P_i$ with P_i one of the graph patterns depicted in Fig. 13. This property can be checked statically with the invariant checker as explained in Section 4.2 or dynamically by analyzing the state transition system via model checking as explained in Section 4.3.*

Graph Grammars A modeling language L, where the abstract syntax of models is described by graphs, can be specified in a constructive way by an attributed graph grammar. A graph grammar consists of a set of creating[7] attributed graph transformation rules and an attributed start graph. The graph transformation rules describe how valid instances of the modeling language at the level of the abstract syntax can be generated.

Definition 16 (graph grammar, graph language). *A* graph grammar *(GG) $GR = (\mathcal{P}, S, TG)$ consists of a set of non-deleting rules $\mathcal{P}$ and a start graph S typed over TG. The* graph language *$\mathcal{L}(GR)$ is defined as $\{G | S \overset{*}{\Rightarrow} G\}$ consisting of all graphs G such that there exists a graph transformation from S to G of arbitrary length.*

Example 11 (SDL - Graph Grammar). As an example for a simple graph grammar we consider the generation of all valid SDL block diagrams. At first, we have to define a related type graph. In this case, we make use of generalization and assume a GTS formalism that is able to cope with it. In Fig. 16, the related type graph with generalization

[7] Note that in graph transformation standard literature the rules of a graph grammar are in general not required to be creating or are not restricted to generate a language, but we restrict them here accordingly to be consistent with more widely used notion of grammars.

can be seen. Note that flattening the type graph would require adding the name attribute to all nodes that are a specialization of type Element as well as the addition of related edge types for all specializations of Connectable. The start graph and rules (productions) of the grammar are described in Fig. 17. The start graph creates a BlockDiagram with a new and unique name (described by newName()). The first rule creates a SystemBlock as an element of a BlockDiagram node to which also a new and unique name is assigned. A Block as an element of a Block node to which also a new and unique name is assigned is created by rule 2. The third rule creates a Process as an element of a Block node to which also a new and unique name is assigned. In contrast to a process, the blocks created by rule 2 may have contained connectable elements. Finally, rule 4 describes that between any two Connectable elements that are contained by the same Connectable a Connection with a new and unique name might be created. An example of how the particular instance graph may be derived by subsequent application of the rules (productions) of the graph grammar starting with the start graph is presented in Fig. 18.

3 Languages & Execution

There are several tools that support languages that have been established on top of graph transformations.[8] Examples are Fujaba[9], AGG[10] [42], Henshin[11] [43], PROGRESS [44], AToM3[12], and MDElab[13].

We will in the paper and this section in particular look on the languages supported by MDElab. We will focus on the direct integration of meta-models resp. class diagrams as type graphs presented in Section 3.1, Story Pattern outlined in Section 3.2, Triple Graph Grammars introduced in Section 3.3, and a Runtime Model Framework introduced in Section 3.4. Additionally concepts supported by MDElab omitted for space reasons are Story Diagrams [45] that extend Story Patterns with control flow constructs and Mega Models for model management and traceability in scenarios with multiple models [46].

3.1 Type and Instance Graphs

In the last section we could already observe that type graphs and instance graphs seem quite similar to class diagrams and object diagrams. Another similarity to meta-models and models also became apparent. We will in the following study both relation in more detail using two concrete examples.

Modeling: Structure with Class Diagrams and Object Diagrams Concerning the similarity between type graphs and instance graphs on the one hand

[8] For an updated view on more available tools we refer to the Transformation Tool Contest[41] initiative.
[9] `http://www.fujaba.de`
[10] `http://tfs.cs.tu-berlin.de/agg`
[11] `http://www.eclipse.org/modeling/emft/henshin/`
[12] `http://atom3.cs.mcgill.ca/`
[13] `http://www.mdelab.org`

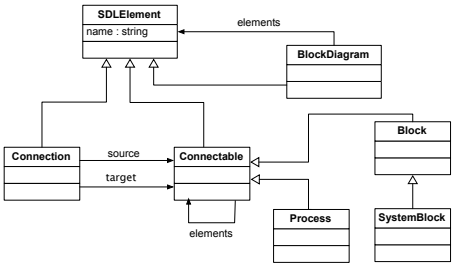


Fig. 16. Type graph for SDL instance graphs

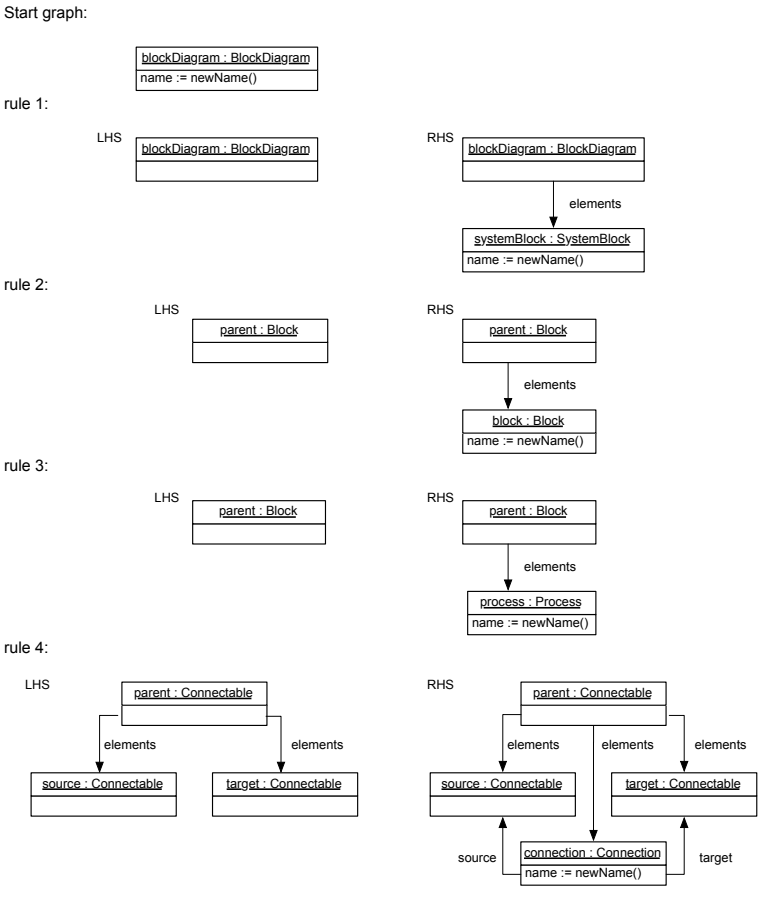


Fig. 17. Start graph and rules for a simple SDL block diagram graph grammar

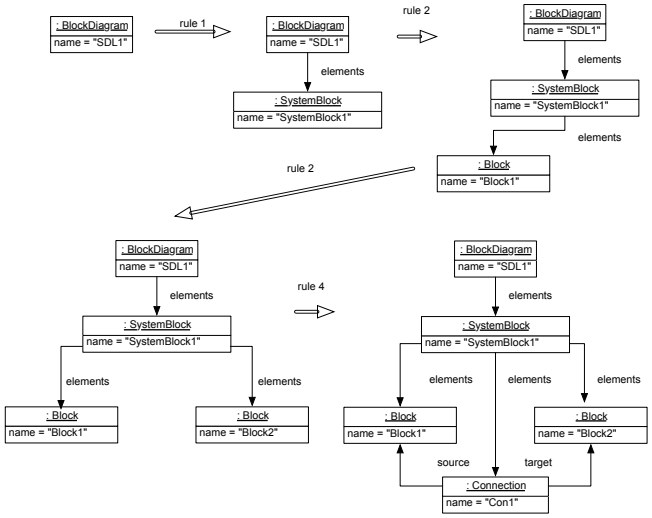
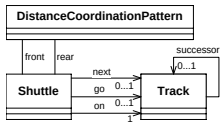


Fig. 18. Derivation of the instance graph example

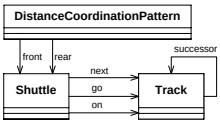
and class diagrams and object diagrams on the other hand holds that node types and their defined attributes relate directly to class definitions and their attributes. Furthermore, simple associations relate to edge types. Undirected associations have to be mapped to directed edges. Thus, the core concepts of class diagrams can be directly mapped. Some other concepts such as association attributes, cardinality constraints, or OCL constraints have to be mapped to additional node types that do not represent classes and sufficiently expressive graph property specification techniques. Analogously, an object diagram is related to an instance graph. It is to be noted that here less differences exist. The common case of binary links can be represented directly in an instance graph and only non binary links require a indirect encoding. An example for such a mapping only for the class diagram is explained in the following Example 12.

Example 12 (RailCab - Class Diagram). A class diagram used for modeling the collision avoidance for the RailCab Example 1 is shown in Fig. 19(a). The class diagram defines the classes Shuttle*,* Track *and* DistanceCoordinationPattern *which are connected through associations. A* Track *may have one successor* Track*, the annotation "*$0 \dots 1$*" expresses the multiplicity of the* successor *association. A* Shuttle *is always located at exactly one* Track *(association* one*) and can mark further* Tracks *through the associations* next *and* go*. A* Track *is marked through the* go *association if the* Shuttle *is about to go to this* Track*, the* next *association models the* Shuttle*'s intent for the following move operation. To avoid collisions,* Shuttles *can instantiate a* DistanceCoordinationPattern *collaboration between them. The* DistanceCoordinationPattern *collaboration employs two roles* front *and* rear *which are both modeled through associations.*

The similarity to the corresponding type graph can be seen in Figure 19(b), which only differs from Figure 19(a) in the absence of the cardinality constraints, which have



(a) Class diagram



(b) Type graph

Fig. 19. Class diagram and type graph for the collision avoidance model

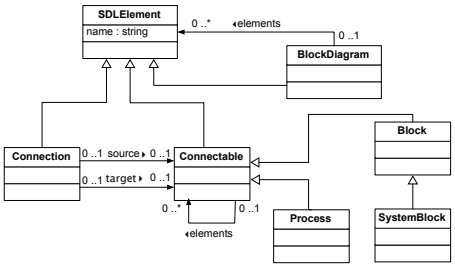


Fig. 20. A simplified meta-model for SDL block diagrams

to be specified by appropriate graph properties (cf. Definition 7), and undirected associations that are mapped to directed edge types.

Due to the explained mapping of class diagrams on type graphs and object diagrams on graphs, we have a sound foundation and semantics based on typed graphs with attributes and inheritance. This will be exploited later when the complete model of the RailCab example that besides the class diagram also includes a number of Story Patterns and simple graph properties are analyzed in Section 4.2 and 4.3 .

MDE: Meta-Model and Model The relation observed for the class diagrams and object diagrams also holds for type graphs and instance graphs on the one hand and meta-models and models on the other hand. Node types and their defined attributes relate directly to class definitions in the meta-model and their attributes. We also explain the mapping by the following Example 13. For the syntax we use in the following as usual the notation of UML class diagrams to depict EMF meta-models.

Example 13 (SDL - Meta-Model). The simplified meta-model used in the following for our consideration of the Example 2 is depicted in Fig. 20. It introduces the main concepts Connection *and* Connectable *that are linked via the associations* source *and* target*. Furthermore, the concept* Connectable *can be refined to be a* Process *or* Block*, where a* Block *can be further be refined to be a* SystemBlock*. The grammar of Example 11 defines in addition that* SystemBlock*s may only contain* Block*s,* Block*s may only contain* Block*s or* Process*es and that* Process*es cannot contain anything. These restrictions are not encoded in this meta-model and additional OCL constraints would have to be added to declaratively exclude all unwanted forms of containment.*

Thus, we have seen that the core concepts of meta-models can also be mapped to typed graphs with attributes and inheritance such that we have also a sound foundation and semantics for them. This will be a foundation for the analysis of model transformations later in Section 4.1 and 4.2.

3.2 Story Pattern

We introduce in this section *Story Patterns* (SPs) [47] that are a compact visual notation for graph transformation rules and graph patterns. SPs have been introduced in the context of Story Diagrams. They take advantage of the similarity between UML object diagrams and graph patterns. As in object diagrams, objects have a name and a classifier, separated by a colon. SPs represent a graph transformation rule such that both sides of the rule are combined. Regular elements belong to both sides, elements with a ++ have to be created and belong only to the RHS, elements with a −− belong only to the LHS and have to be removed.

Example 14 (RailCab - SP(1/3)). The SP in Fig. 21 related to the class diagram in Fig. 19(a) deletes the associations of type go *and* on *between* Shuttle *s1 and* Tracks *t2 and t1, respectively. Further, the SP creates an association of type* on *between s1 and t2. The SP can only be applied to an instance situation if no* Shuttle *is located at* Track *t2.*

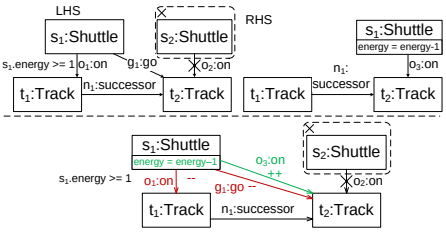


Fig. 21. moveSingle: SP for moving a shuttle to an empty track.

The semantics of SP is given via a mapping on GTS rules (cf. Def. 8) assuming a proper mapping from class diagrams (meta-models) to type graphs with attributes and inheritance as described in Section 3.1. For the translations of SPs into GTS rules we have split up SPs into a graph pattern for the LHS and RHS as follows: All elements that have no annotation or a −− become nodes and edges in the graph pattern for the LHS. Note that in particular the NACs are not allowed to carry annotations and thus always become part of the LHS. Given the case that the SP contains NACs, they are directly mapped to NACs in the graph pattern $(L, \{N_i | 1 \leq i \leq I\})$ (cf. Def. 5) with I being the number of NACs. Elements that have no −− attached and are not part of a NAC become nodes and edges of the RHS. The types of the nodes and edges are set according to the mapping to the type graph. All elements only occurring in the RHS but not in the LHS are obviously those annotated with a ++ and all elements besides the NACs only occurring in the LHS but not in the RHS are obviously those annotated with a −−. A SP is called *side-effect free* if no elements are annotated with ++ or −− and can be used to describe basic graph properties.

If the SPs do not delete nodes but at most edges, the conservative and dangling-edge-collecting approach are identical. However, if also nodes are deleted, one of the two options has to be chosen.

Example 15 (RailCab - SP(2/3)). To exemplify the mapping from a SP to a GTS rule, we consider here again the simple rule for the Example 12 of the RailCab system that describes the Shuttle's move operation. The SP for this rule is given in Fig. 21 (lower part) and the corresponding GTS rule is given in Fig. 21 (lower part). The correspondence between nodes and edges in the SP and the GTS rule is indicated through the names. Note that this an improved version of the GTS rule in Fig. 9 (upper part) that excludes collisions by checking that no other Shuttle is located on the Track the Shuttle moves to.

We restrict our discussion here to the main features of SP and refer to [48] for a more complete coverage of features. The Story Diagrams language integrates SPs as basic building blocks and in addition offers the typical control flow concepts of an UML activity diagram to steer when which SP should be applied. An additional activity node foreach in these Story Diagrams permit to also apply a SP to all matches in the considered object graph. More on Story Diagrams can be found also in [48].

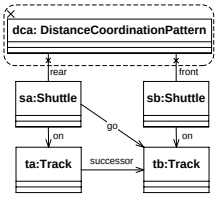
Modeling: Structural Dynamics SPs can be employed in combination with class diagrams to describe the structural dynamics and other behavior of dynamic systems. To achieve this, one has to provide a suitable class diagram describing all possible states of the system under development, a set of SPs that specify the system's behavior, and a set of side-effect free SPs that specify required system properties.

Example 16 (RailCab - SP(3/3)). The behavior of a model of the RailCab system of Example 1 to study the collision avoidance is defined by a set of SPs. The class diagram of the model is depicted in Fig. 19(a) of Example 12. Based on this class diagram the SPs shown in Fig. 21, 22(b), 22(c), and 22(d) describe how a shuttle may move. Fig. 21, 22(c), and 22(d) specify the movement of Shuttles under different conditions – i.e. succeeding Track is empty, Shuttle has the DistanceCoordinationPattern protocol established – and Fig. 22(b) specifies the instantiation of the DistanceCoordinationPattern protocol. The operational rules are equipped with priorities ensuring that rules specifying the Shuttles' movement without an established DistanceCoordinationPattern protocol are preempted by rules requiring the DistanceCoordinationPattern protocol. The instantiation of the DistanceCoordinationPattern protocol has the highest priority and it's removal the lowest.

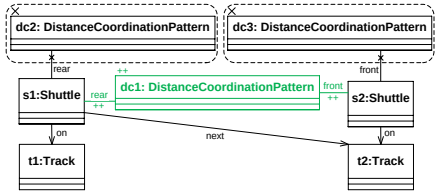
Besides operational rules, the model also consists of forbidden patterns that identify system states, which are considered unsafe or may lead to an unsafe situation. For the RailCab system these forbidden patterns are depicted in Figure 22(a) and Figure 22(e), which are SP without side-effects and describe situations where the DistanceCoordinationPattern protocol is not established for two Shuttles located at succeeding Tracks and a collision – i.e. two Shuttles at the same Track – respectively.

Overall, the complete RailCab system is specified through six rules and 19 forbidden patterns (see [49]). Most of the forbidden patterns are required to encode cardinality constraints. We use the conservative approach for the rule execution for this example,

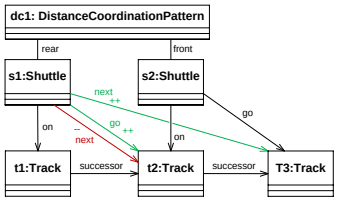
as edges represent meaningful real world concepts that should not be implicitly deleted. Anyway, the rules only delete DistanceCoordinationPattern nodes together with its two links to the connected shuttles. Thus, in this case there will be no valid graph where the behavior would differ if the dangling-edge-collecting approach would have been chosen.



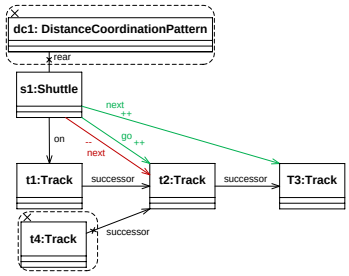
(a) Invariant: No uncoordinated movement of Shuttles in close proximity, which would constitute a hazard



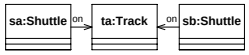
(b) Instantiation rule: creating a DistanceCoordinationPattern



(c) Behavioral rule: Coordinated movement



(d) Behavioral rule: unrestricted movement for a solitary Shuttle



(e) Invariant: No collision accident

Fig. 22. SPs specifying the structural dynamics of the RailCab model

MDE: Refactoring As outlined in the following example, SPs can be also used in the context of MDE. A first example is the specification and execution of a *refactoring* [50,7]. Based on the meta-model of a source model (in this case the SDL block diagram meta-model), the required refactorings are described by SPs. An in-place transformation of a source model then results in a refactored model.

Example 17 (SDL - Refactoring). We consider here again SDL block diagrams as in Example 11.[14] *Assume that we want to develop a refactoring that change improper*

[14] It is to be noted that the case considered here is not covered by the GG example and later TGG examples where for space reasons the rule for connections across the hierarchy are omitted.

connections across Block *boundaries. In case two* Blocks *are embedded into different* Blocks *but are directly connected, this single* Connection *has to be replaced by three* Connections *with the same* name*. One* Connection *between the outer blocks and two additional* Connections *linking the inner bocks to their outer block (see Fig. 23). As in case of refactoring deleting all edges to removed elements is rather cumbersome, here the dangling-edge-collecting approach is used.*

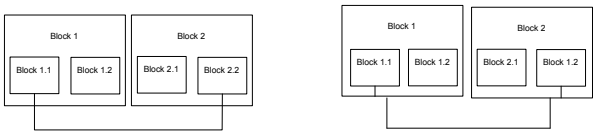


Fig. 23. Required refactoring at the concrete syntax level

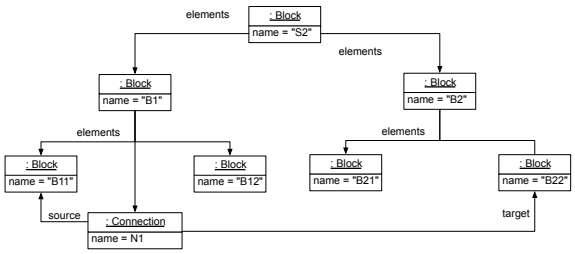


Fig. 24. The abstract syntax of the example model before the refactoring

To capture the SDL block diagrams, we at first need a meta-model as depicted in Fig. 20. Based on this meta-model the example of Fig. 23 can be depicted at the level of the abstract syntax in Fig. 24. With a single SP we can then describe how to manipulate the models by means of in-place model transformations. The required changes for the refactoring are depicted in SP of Fig. 25. The direct Connection *is removed and instead three new* Connections *with the same name as the removed* Connection *are created that ensure that the* Connections *are always respect the block hierarchy. In Fig. 26 we can see the expected result of refactoring the model of Fig. 24 according to Fig. 25.*

It is to be noted that the SPs can also be used to define complete model transformations in an operational style. However, either we simply identify corresponding elements in the other model using names or complex additional structures have to be maintained explicitly or explicit control structures as supported by Story Diagrams would be required. In the next section, we will instead discuss how the same kind of problem can be addressed with a graph grammar based approach in a more elegant and effective manner by specifying the relation between source and target model declaratively and derive related operational solutions for the model transformation.

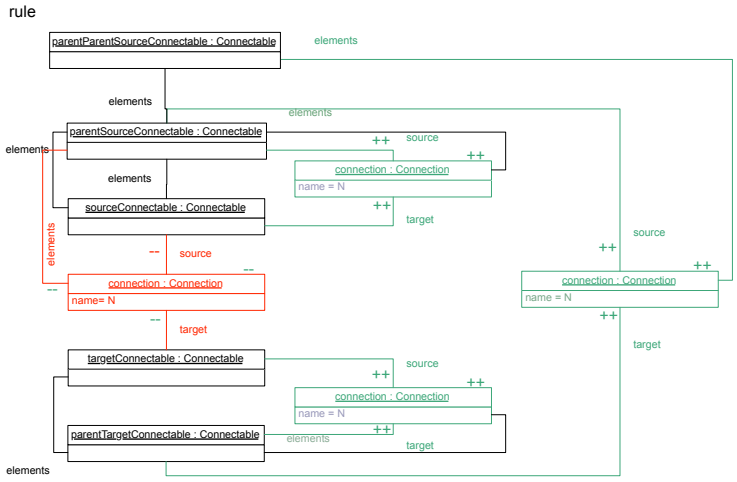


Fig. 25. SP for the refactoring that corrects connections across hierarchy

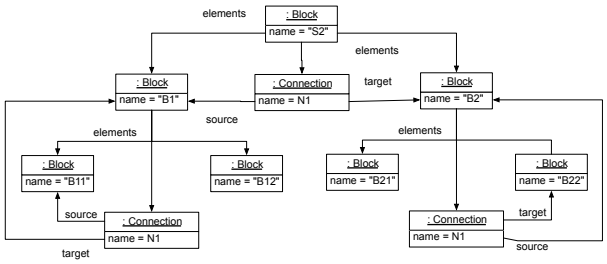


Fig. 26. Result of refactoring the model of Fig. 24 with the SP of Fig. 25

Code Generation and Interpreter Story diagrams can be executed by generating code, which is the approach used in Fujaba and former versions of our tool, and by interpreting them directly [45].

The former code generation required that all conditions are specified as Java conditions such that they can be simply embedded in the generated code. The generated code has a very good performance, but was not very flexible. First, it did not support OCL. Second, the search for a match happened according to fixed order for the nodes of an SP set at compile-time. Third, changes of the SPs and Story Diagrams at runtime were not possible due to the generated code.

To overcome these limitations, an interpreter was developed that supports OCL conditions, adjusts the matching order to the instance graph to decrease the worst-case execution times, and permits to modify the SPs and Story Diagrams at runtime (higher-order transformations). In addition, the SP matcher can start the matching with any initial bindings such that also incremental matching of

SPs based on change events could be realized with the interpreter. The tool set is completed by a debugger at modeling level (see [51]).

Note also that SPs and Story Diagrams have already be employed for industrial strength case study such as the MATE project [52] for the enhanced model validation and model transformation of Simulink/Stateflow models. The current and older versions of the SP and Story Diagram Interpreter have been realized based on Eclipse and the Eclipse Modeling Framework. It can be downloaded from our Eclipse update site `http://www.mdelab.org/update-site`.

3.3 Triple Graph Grammars

In this section, we present *Triple Graph Grammars* (TGGs) [53] that allows specifying model transformations in a rule-based and relational way. In particular, graph grammars as introduced in Section 2 are the formal basis for this model transformation specification language.

In order to properly specify the triple graph transformations, we require a meta-model for the source model, for an additionally supported correspondence model, which stores traceability information that allows finding elements of one model that correspond to an element of the other model, and for the target model. TGG rules are accordingly divided into three domains: The source model domain (left), target model domain (right), and the correspondence model domain (middle).

A TGG consists of an axiom (the grammar's start graph) and several TGG rules that describes how consistent triples of source, correspondence and target models can be generated. TGGs permit to derive three kinds of model transformation directions: Forward, backward, and correspondence transformations. A forward (backward) transformation takes a source (target) model as input and creates the correspondence and target (source) model. A correspondence transformation[15] requires a source and target model and creates only the correspondence model. In addition, also forward or backward model synchronization is possible where only changes are propagated. As in case of TGGs and related operational SPs only bookkeeping edges are deleted, the chosen approach whether conservative or dangling-edge-collecting does not matter.

Example 18 (SDL - TGG Specification). For a transformation from SDL block diagrams to UML class diagrams we require a meta-model for SDL block diagrams (as already depicted in Fig. 20), and a meta-model for UML class diagrams as presented in Fig. 27. There is also a correspondence meta-model as depicted in Fig. 28.

The axiom in Fig. 29 relates the root elements of the source and target models with the axiom correspondence node. The attribute assignments, defined through OCL expressions in our tool environment, state that the names of the block and class diagrams must be equal. Rule 1 creates a Block *and a corresponding* UMLClass*. The* BlockDiagram *and* ClassDiagram *must already exist. Rule 2 creates a* Block *in the block diagram domain and connects it to an already existing parent* Block*. In the class diagram domain, a class is created and connected to the parent* Block*'s* UMLClass *with an* Association*.*

[15] The correspondence transformation is also known as mapping or model integration.

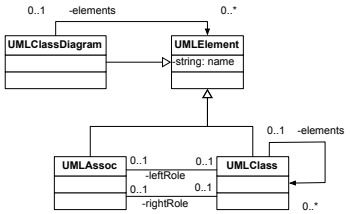


Fig. 27. Simplified meta-model for UML class diagrams

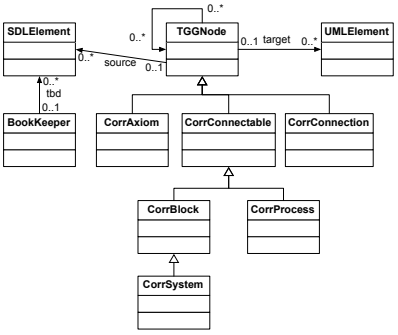


Fig. 28. Correspondence meta-model with extra concept for bookkeeping

Rule 3 is analogous to Rule 2, but covers the creation Process in the block diagram domain. Rule 4 creates a Connection and a corresponding UMLAssoc between already corresponding Connectables in the block diagram domain and UMLClasses in the class diagram domain.

Triple Generation The TGG itself can be used to build the three models in parallel by applying TGG rules successively to extend the axiom. In the resulting graphs, the source and target components (i.e. the source and target models) are consistent to each other according to the TGG. We employ this triple generation, for example, to generate test cases for model transformation implementations that need to adhere to the TGG. Since the TGG is a specific graph grammar (see Section 2), it defines a language of consistent source and target models.

Example 19 (SDL - TGG - Triple Generation). For the SDL block diagram to UML class diagram transformation of Example 18 the triples can be generated by starting with the axiom and then applying the rules directly as if they were simply SPs.

Forward & Backward Transformation However, to perform model transformations in practice it would be too cumbersome to generate all triples of related size to determine what the output of a transformation should be. Instead, under some well-formedness conditions for the TGG rules an efficient operationalization can be generated, which create target model elements for given source model elements, so that both are consistent to each other. These well-formedness conditions are described in more detail in [54,55] and range from simple syntactical checks to more expensive checks (as discussed, for example, in Section 4.1) that can still be performed at design time.

For each of the aforementioned transformation directions, separate operational rules are derived from the TGG rules. In particular, the elements with ++ in the source domain become regular elements by removing the ++. The parts

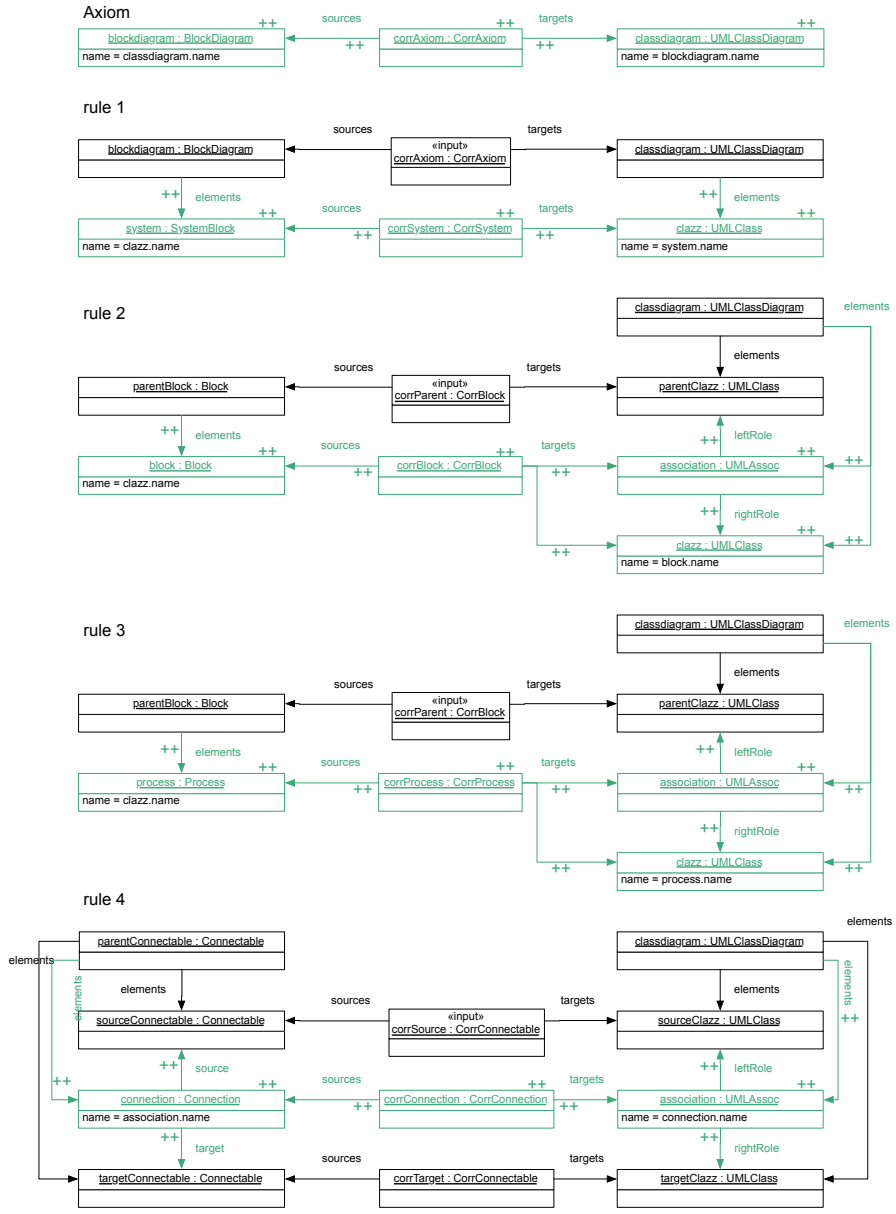


Fig. 29. TGG rules to transform SDL block diagrams into UML class diagrams

with ++ of the correspondence domain and target domain remain as they are. The operational rules also have to make sure that a given source model element is only transformed once. This requires a bookkeeping mechanism, which keeps track of those elements that were already transformed, and those that still have to be transformed. Accordingly, an initially set link to a special bookkeeping object is removed when a source element has been translated and its non-existence is tested for all context objects as they should have been translated already.

Example 20 (SDL - TGG - Forward Transformation). For Example 18 the SPs derived from the TGG rules to transform SDL block diagrams into UML class diagrams are depicted in Fig. 30. While the elements of the source model become additional pre-conditions, the new elements of the TGG rule in the correspondence model and target model are generated. In addition, it is checked if a link to a bookkeeping object is available. It ensures that the translated source elements have not yet been processed (required edge) and that all context elements of the source model have been processed (forbidden edges). The links to the translated elements are deleted by the rules such that subsequent rule applications will not consider the covered elements of the source model.

The steps of a forward transformation with TGGs are depicted in Fig. 31. Dashed lines separate the elements covered by each step for the source model and the generated elements for the correspondence and target model.

Consistency Transformation TGGs can also be used to derive the correspondence model for a give source and target model. In that case, in each TGG rule all elements of the source and target domain become part of the pre-condition of the related SP and only the parts of the correspondence domain to be generated become part of the post-condition. In addition, bookkeeping must ensure that only those elements of the source and target model are considered as match for the SP.

Forward & Backward Synchronization In case of model synchronization, the target and correspondence model are also input for the processing. Next links leading from all referenced correspondence nodes to the newly created correspondence nodes (also created by the model transformation, but omitted there for space reasons) capture the dependencies between different rule applications related to the correspondence nodes. The goal of the forward synchronization is then to propagate only the changes that occur in the source model to the correspondence model and target model but regenerate only the necessary minimum.

Example 21 (SDL - TGG - Model Synchronization). We consider here how to synchronize a SDL block diagram with a UML class diagram, which is related to the transformation considered in Example 18 and 20.

The considered change in a SDL block diagram is depicted in Fig. 32. A block Block3 and a contained block Block4 are moved from the embedding block Block1 into another block Block2. Using model synchronization only the changes are propagated.

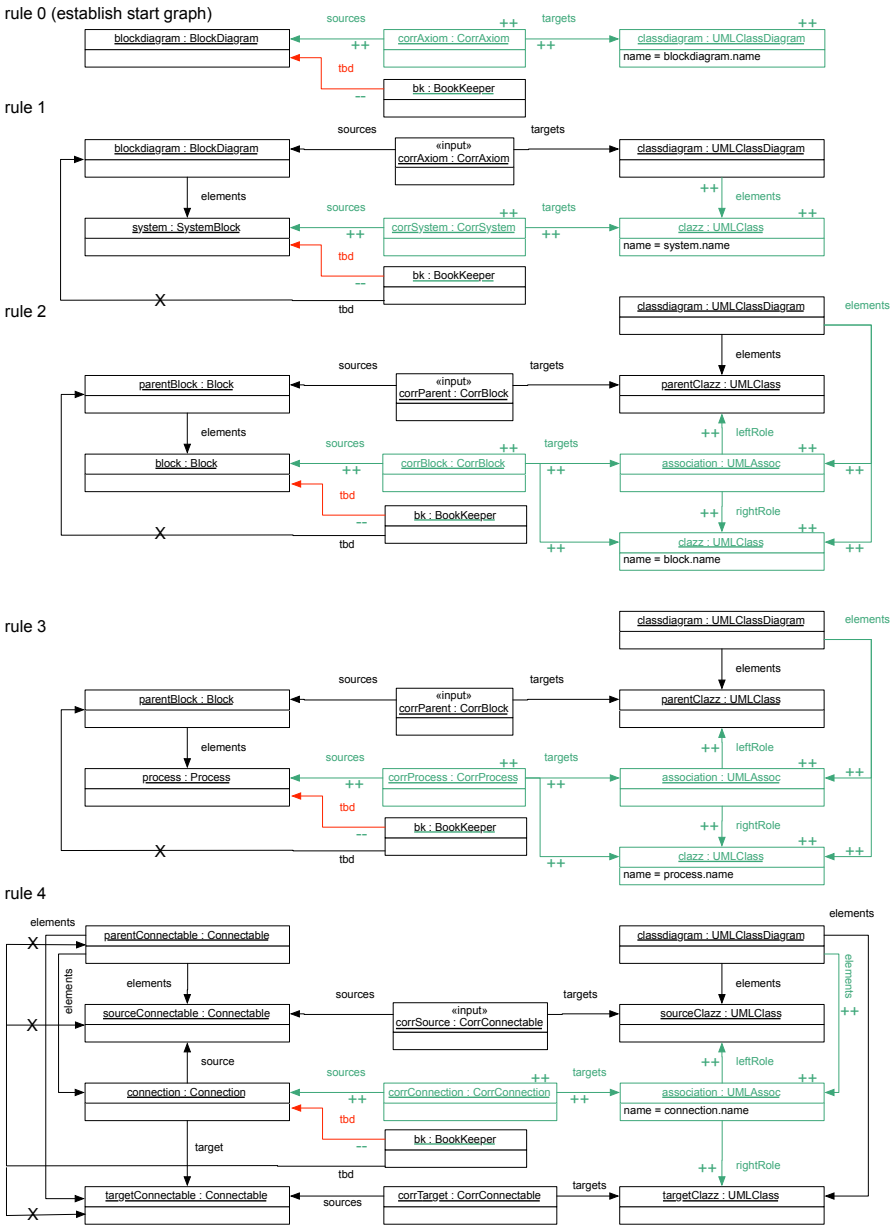


Fig. 30. Derived SPs to transform SDL block diagrams into UML class diagrams

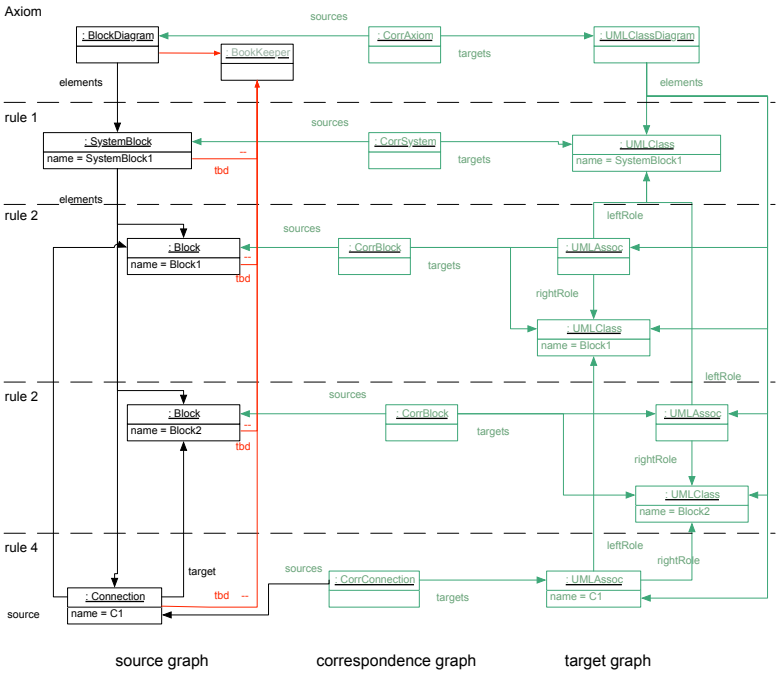


Fig. 31. Derivation of a forward transformation with TGGs

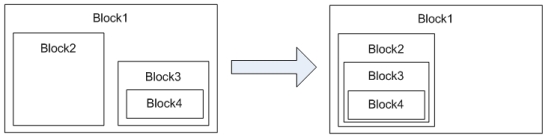
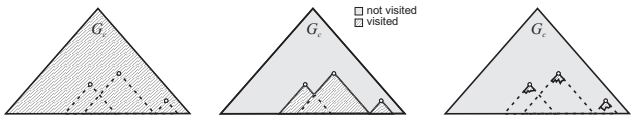


Fig. 32. Considered change in a SDL block diagram

Model Transformation and Synchronization Engine Our implementation of the *model transformation* takes advantage of the knowledge which correspondence nodes can be a trigger for a TGG rule. It manages a queue of the created correspondence nodes and then only triggers the necessary rules for those nodes. In addition, the bookkeeping is used as an additional side-condition to limit the search space as newly matched elements are always still connected with the bookkeeping object. Both tricks permit to avoid any global search for matches and considerably speedup the transformation.

In case of *model synchronization*, the correspondence model and target model are also input for the processing. In addition, we remember the dependencies between rule applications in the correspondence model in form of additional next links as defined in Fig. 28 between the newly created correspondence nodes and those in the LHS when transforming as well as synchronizing the models.



(a) batch (b) average case incremental (c) worst-case incremental

Fig. 33. Parts of the correspondence model G_c affected by the different algorithms

Therefore, the correspondence model with the next links has further the form of a directed acyclic graph (visualized in Fig. 33 as a tree) and we can exploit this acyclic structure in a number of improvements for the efficient incremental processing of changes for the synchronization (similar to ideas for incremental parsing as outlined in [56]).

In [57,58], we have achieved that in most cases a single change can be processed in the average case with only logarithmic effort concerning the size of the models involved. In [59], we further improved the solution in such a way that even in the case of multiple changes, we can ensure only a slow increase of the efforts and that the effort always remains below or equal to the batch algorithm. Further improvements ensure that for a restricted class of TGG rule sets, also in the worse-case [60] an incremental processing can be observed.

In the latest version [60], we additionally use the information available in the declarative TGG rules to also derive additional checks to repair structural changes by adjusting links and avoiding retransformation of elements. This also ensures that the effects of changes are only propagated when necessary (cf. Fig. 33(c)). Therefore, the new algorithm drastically improves our former results [57,58,59] and we can show that it is even optimal if the overall algorithm and not the rule matching dominates the complexity.

Example 22 (SDL - TGG - Efficient Model Synchronization). In case of a complete transformation the whole source model would have been traversed following the scheme in Fig. 33(a). If we in contrast follow the scheme of Fig. 33(b), only the complete source model below the change would be retransformed. Therefore, the synchronization would work as if Block3 *and a contained block* Block4 *embedded into the block* Block1 *are first deleted and a new* Block3 *and a contained block* Block4 *are created located under the block* Block2*. The improved version of Fig. 33(c)) will instead consider the right scenario and take into account that block* Block3 *and the contained block* Block4 *are moved from the embedding block* Block1 *to block* Block2*.*

Finally, in Fig. 34 the resulting effects of the model synchronization following the scheme of Fig. 33(c) are presented. In the source model, only a link is deleted and block Block3 *is added a new element of block* Block2*. Due to the improved handling the resulting synchronization effects requires that in the correspondence model and the target model the related links are corrected as the algorithm is able to reuse the old correspondence model and target model related to block* Block3 *and block* Block4*.*

It should be noted that in the considered case the absolute improvement is only moderate while the relative improvement is already considerable. However, if the part of the correspondence model located under the correspondence directly affected by the a

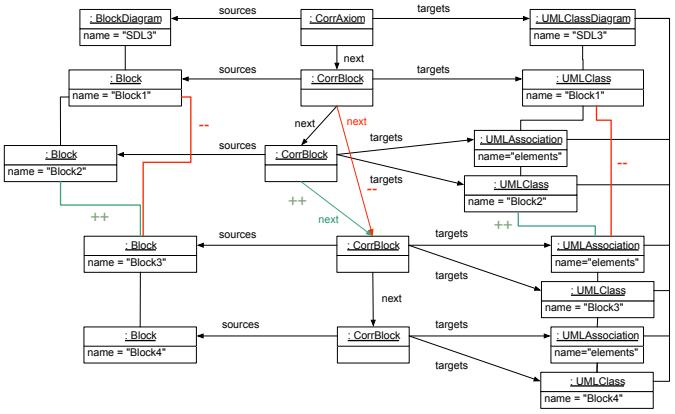


Fig. 34. Effects of the model synchronization following the scheme of Fig. 33(c)

change is large, the effort without repair can become as large as transforming the model anew while only a few local changes are required for our synchronization algorithm.

TGGs have already be employed for an industrial strength case study in the context of an automotive tool chain. The task was to transform elements of SysML system models, which refer to the software, into an AUTOSAR software architecture model. Additionally, both models had to be kept synchronized after transformation [61]. The current and older versions of the presented TGG Engines have been realized based on Eclipse and the Eclipse Modeling Framework. The current version can be downloaded from our Eclipse update site `http://www.mdelab.org/update-site`.

3.4 Runtime Model Framework

In the following, we discuss a framework leveraging runtime models for self-adaptive systems [62,63,64,65]. Having explicit models that represent the running system, MDE techniques based on graph transformations (cf. Section 2) can be applied. The generic architecture of the framework, which extends the control loop concept proposed in [8], is depicted in Fig. 35.

A *Managed System* provides *Sensors* and *Effectors* that are used to observe and change the running system, respectively. These sensors and effectors provide the so-called *Source Model*, which is a runtime representation of the system. This model is *causally connected* to the system, which generally means that any change in the running system is reflected in the source model, and any change in the source model is reflected in the system. Therefore, this model can be directly used by *Autonomic Managers* to perform the feedback loop activities that comprise the monitoring and analysis of the running system, and if changes are required, the planning and execution of adaptation to the system.

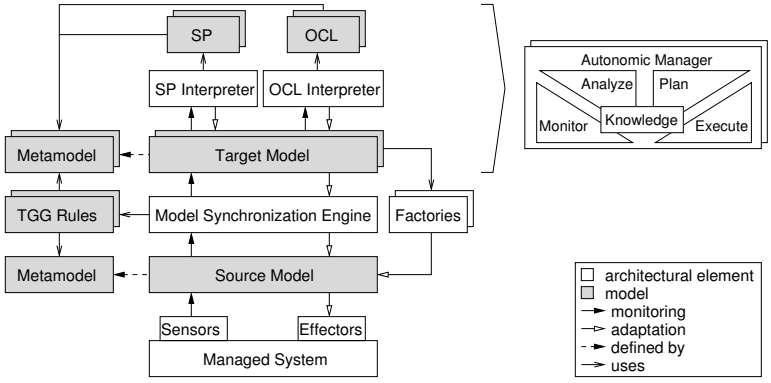


Fig. 35. Generic Architecture for the Runtime Model Framework (cf. [64])

However, a source model represents all functionalities and concerns of the sensors and effectors. Therefore, it is usually complex and related to the solution space and platform of a managed system. Thus, a source model provides a view on a system at a low level of abstraction, which could make it laborious to use it as a basis for the feedback loop activities performed by managers.

Therefore, several *Target Models* are derived from a source model at runtime. Each target model abstracts from the source model and it provides a specific view on a managed system required for a certain self-management capability. As an example, a target model might represent the performance state or failures of a system to address self-optimization or self-healing, respectively. A manager concerned with self-optimization will use only the target models relevant for optimizing a system, but not necessarily consider target models addressing other capabilities like self-healing. This and appropriate abstractions of models, reduce the complexity for individual managers in coping with runtime models and performing their activities.

Thus, target models tend to provide views related to problem spaces of different self-management capabilities and to abstract from the underlying system platform. This supports the reusability of managers that focus on problem spaces shared by different managed systems. Furthermore, as target models can be platform-independent, the kinds of target models used in our approach are primarily defined with the needs of the autonomic managers in mind rather than focusing on the underlying infrastructure.

Therefore, managers preferably use target models than a complex source model to perform the feedback loop activities. This requires that a target model is causally connected to the source model. Thus, changes in the source model are reflected in target models for monitoring, and vice versa for adaptation. To maintain different target models at runtime and to realize causal connections between the models, we use our *Model Synchronization Engine* based on TGGs that incrementally synchronizes models with each other (cf. Section 3.3). To use the engine, source and target models have to be defined by meta-models that

are the basis to define *TGG Rules* (cf. Fig. 35). These rules define how a pair of source and target models are synchronized with each other.

However, all concepts in one model need not to be represented in the other model. Especially, concepts in a source model may not be reflected in the target model since target models are at a higher level of abstraction than source models. Hence, synchronizing source model changes reflecting changes in the managed system to a target model for monitoring is not problematic. During synchronization, concepts that are represented in a source model but not in a target model are simply discarded, which causes the intended abstraction. Therefore, changes can be propagated from source to target models without any difficulty. However for adaptation, the opposite direction of propagating target model changes to the source model is problematic since these changes have to be refined in order to be reflected properly in the source model. The abstraction gap between source and target models prevents a bidirectional synchronization using the TGG-based transformation engine. Therefore, this abstraction gap is filled by *Factories* (cf. Fig. 35) that are invoked on target models but they operate on the source model where all required information is provided. Hence, the intended changes are performed by factories on the source model and afterwards they are synchronized to target models by the synchronization engine, which makes them visible for managers. Though factories are currently implemented in *Java*, they could also be specified and realized by graph transformation rules, like SPs (cf. Section 3.2), that perform an in-place transformation of the source model. Further issues concerning adaptations based on target models are discussed in [64].

Overall, this approach leverages abstract runtime models and MDE techniques for adaptive systems. In contrast to a complex source model, an abstract target model provides a more appropriate abstraction for autonomic managers and a more specific view for a self-management capability. Both aspects ease the work of managers. Moreover, target models can abstract from a concrete managed system and platform, which supports the reusability and extensibility of managers being able to operate on these models across different systems.

While the synchronization between source and target models with TGGs as discussed above supports the monitoring and the execution of adaptations, the analysis and planning activities of the feedback loop can be tackled as well by graph transformations. In [65], we discuss the applicability of SPs (cf. Section 3.2) working on runtime (target) models. For analysis, SPs perform checks on a runtime (target) model, while for planning adaptations, a runtime (target) model is transformed in-place by SPs. In addition to SPs, OCL expressions can be used by autonomic managers to perform the feedback loop activities. This is outlined on top of Fig. 35 by an implementation example of an autonomic manager. The analysis and planning activites of this manager are specified by SPs and OCL expressions based on the target meta-models. These SPs and OCL expressions are executed by corresponding interpreters and they operate on target models to analyze the managed system and to plan adaptations.

As sketched in Fig. 35, several runtime models, like SPs, OCL, or target models, and several model operations, i.e., tools like the synchronization engine

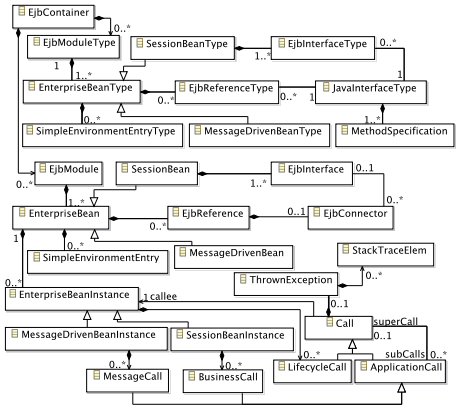


Fig. 36. Simplified source meta-model [64]

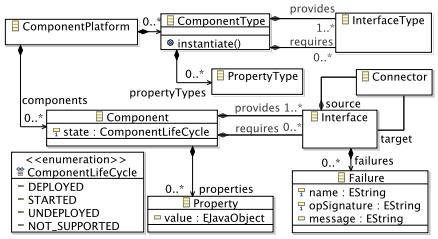


Fig. 37. Simplified target meta-model [64]

and different interpreters, are used to implement and execute a feeback loop with its activities. To explicitly specify the interplay between all these models and operations, so-called *megamodels* can be employed [66]. A megamodel is a model that has models as its elements and that captures the relationships between these models in the form of model operations. Thus, a feedback loop and especially its flow of activities implemented by interacting models and model operations can be specified by a megamodel. Moreover, having an interpreter for megamodels, a megamodel can be kept alive at runtime in order to maintain the different runtime models and operations, and to directly execute a feedback loop. Therefore, besides making feedback loops explicit in the design of a self-adaptive system, a megamodel approach together with an interpreter supports the execution, adaptation, and composition of feedback loops [67].

Example 23 (Runtime Model Framework). As an example, we consider managed systems implemented with Enterprise Java Beans 3.0 (EJB) *technology. Fig. 36 shows the simplified*[16] *meta-model for the source model. Based on this meta-model, EJB-based systems can be described at three different layers. The top layer covers components types that correspond to artifacts from the development phase. These types define the configuration space for a system. Concrete configurations of a system are instances of these types that are deployed in a container (server) and they are considered by the middle layer. Finally, the lower layer addresses bean instances and interactions by means of calls among them.*

Since models conforming to this meta-model are complex, very detailed, and platform-specific, we introduce a meta-model for generic component-based software systems, which is used for target models. A simplified version[17] *of this meta-model is depicted*

[16] The meta-model depicted in Fig. 36 is simplified as it does not show any attributes, operations, and enumerations, and it hides some associations. Moreover, elements for concerns like security, transaction, timers, or quiescence are hidden.

[17] The meta-model is simplified as several attributes and three associations to navigate from a Component, Interface, or Property to their corresponding types are hidden.

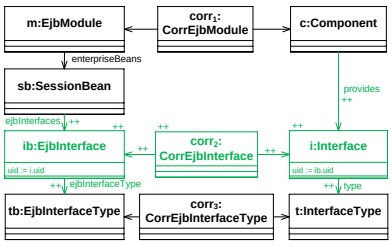


Fig. 38. Example TGG Rule [64]

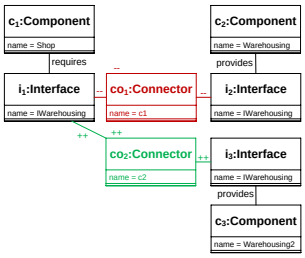


Fig. 39. Example adaptation SP

in Fig. 37. It generally considers component-based systems and it covers failures that have occurred when using a provided interface.

Using this generic meta-model, EJB-based systems can be described in a platform-independent and abstract manner, while highlighting the specific concern of failures occurring in the running system. Hence, a target model as an instance of this generic meta-model has to be synchronized at runtime with the source model conforming to the meta-model depicted in Fig. 36.

Overall, eleven TGG rules were required to specify the synchronization between instances of these specific source and target meta-models. One of these rules is depicted in Fig. 38. This rule transforms and synchronizes an EjbInterface element to an Interface element, or vice versa. Model elements on the left refer to the source model, elements in the middle to the correspondence model, and elements on the right to the target model. Thus, for each EjbInterface provided by a SessionBean that is part of an EjbModule in the source model an Interface is created in the target model and associated as a provided interface to the Component that corresponds to the EjbModule. Moreover, a CorrEjbInterface element as part of the correspondence model is created that stores the mapping between the EjbInterface and the Interface. Finally, the Interface is associated to the InterfaceType that corresponds to the EjbInterfaceType to which the EjbInterface is linked. Likewise, if an Interface is created in the target model, it is transformed or synchronized to an EjbInterface in the source model. This rule also shows how attribute values are synchronized. The uid of an Interface is directly derived from the uid of the EjbInterface, and vice versa.

Moreover, this rule exemplifies that not all concepts in one model need to be represented in the other model. A SessionBean in a source model is not reflected in the target model and therefore no correspondence model element exists that is connected to a SessionBean.

As an example for manipulating a target model, Fig. 39 shows a SP specifying one step within a complex architectural adaptation. This pattern works on target models that conform to the meta-model shown in Fig. 37. Considering a web shop as an example system, it changes the binding between components of the system by removing the connector between the Shop and the Warehousing components, and creating a new connector to bind the Shop component to the Warehousing2 component. This architectural adaptation is motivated by a faulty Warehousing component that causes failures at runtime. This requires that requests from the Shop component are routed to the alternative Warehousing2 component. Similar SPs are used for the other adaptation steps, like checking if failures occur at runtime, to deploy and start the alternative component, and to stop and undeploy the faulty component.

The framework has been employed to academic case studies for self-adaptive software systems. The framework's implementation is continuously enhanced and elaborated, and it is available on request. For further information on the research prototype, please contact us at `contact@mdelab.org`.

4 Analysis

For the introduced SP and TGG languages as well as the Runtime Models Framework a number of analysis techniques available for graph transformations can be employed. The formal foundation of graph transformation permits to analyze them in different ways. At first we can use *static analysis techniques* that only analyze the structure of the GTS rule sets such as static conflict detection [68] or invariant checking [49]. Secondly, there are analysis techniques that explore the state space directly such as *model-based testing* [69,70,71] or *model checking* [72,73]. Moreover, based on the formal foundation of graph transformation, it is possible to apply *theorem proving* to graph transformation [74,75]. In [76], for example, we already verified behavior preservation of a model transformation (see [76]) specified with TGGs using theorem proving. [77] presents another static analysis technique for graph transformation systems based on a translation into so-called Petri graphs, which can be seen as unfoldings of the graph transformation system. Finally, verification techniques for the *correctness of so-called graph programs*, equipping graph transformation rules with basic control structures, have been developed in [78], following Dijkstra's approach to program verification, and [79], where a Hoare calculus for graph programs is presented.

We will look in the following into static conflict detection for model transformations with TGGs in Section 4.1, invariant checking for model refactorings with SPs and systems with structural dynamics with SPs in Section 4.2, and model checking for systems with structural dynamics with SPs in Section 4.3.

4.1 Static Conflict Detection

Conflict detection allows for detecting and visualizing conflicts that may occur between rule applications. Conflicts arise, for example, if one rule deletes an element used by the other rule. This is because after applying the first rule and deleting this used element from the other rule, this other rule cannot be applied anymore. Conflicts between rule applications can be computed at design time by analyzing the corresponding graph transformation rules. To this extent, the so-called theory of critical pairs [35,80] can be applied. A critical pair describes a conflict between two rule applications in a minimal context. AGG is a graph transformation tool [42] able to compute the complete set of critical pairs for a given set of graph transformation rules for the conservative approach.[18] Since this set can be computed from the rules (without executing them and generating

[18] Note that we can verify with the invariant checker discussed in Section 4.2 whether for a given rule set the dangling-edge-collecting approach and the conservative approach result in the same behavior.

a corresponding state space), conflict detection is a so-called static analysis technique. In general, computing the complete set of critical pairs for a given pair of rules is exponential in the number of rule elements in the LHSs of these rules. This is because so-called overlaps (jointly surjective morphisms, see Def. 2) of the rules' preconditions need to be built in order to compute all possible minimal contexts of rule applications.

Example 24 (SDL - TGG - Static Conflict Detection). In [54,55], we perform conflict detection using AGG on the rule-based specification of model transformations in order to find out at design time if each model transformation following this specification can be performed efficiently, i.e. without backtracking at runtime.

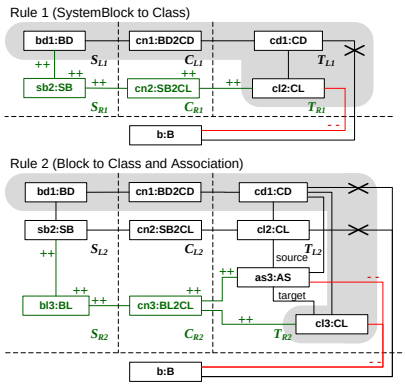


Fig. 40. Backward rules $r1^{BB}$ and $r2^{BB}$ in conflict

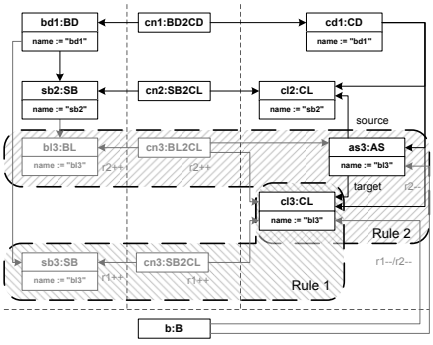


Fig. 41. Operational backward rules $r1^{BB}$ and $r2^{BB}$ competing for translating cl3

For the example transformation rules in Fig. 40, depicting backward transformation rules with bookkeeping from class diagrams to block diagrams derived from a similar TGG as presented earlier in this paper, a conflict arises. The LHS of rule 1 is completely contained in the LHS of rule 2 (shaded background). Therefore, both rules can be applied in the same context and compete for the translation of the same Class, namely cl2 in rule 1 and cl3 in rule 2, respectively. Fig. 41 shows the backward transformation of a class diagram model with both alternatives. In particular, cl3 can be translated by rules 1 and 2 but with different results, which are both shown in the figure.[19] *Rule 1 creates a second SystemBlock in the block diagram model, rule 2 creates a Block. In particular, we have a delete-use-conflict because if the bookkeeping edge to the instance class cl3 is deleted by rule 1, then it cannot be matched anymore by rule 2 and the other way round. In addition, rule 1 leaves as3 untranslated. After applying rule 1 to translate cl3, the bookkeeping edge to as3 still exists. Therefore, the transformation result is not unique and our TGG model transformation implementation can not perform in a safe way the corresponding model transformation efficiently without backtracking.*

[19] Thereby, cl2 of rule 1 as well as cl3 of rule 2 are mapped to the instance Class cl3.

4.2 Invariant Checking

Given a set of SPs describing the behavior of a system and required properties in form of side-effect free SPs being forbidden graph patterns, we present here a static verification technique we developed for analyzing the structure of the underlying GTS rules assuming the conservative approach to determine whether the required properties are inductive invariants.[18] Since it is a static analysis technique, it even works when we have arbitrary many or even infinitely many reachable graphs. We will only review here the basic idea [49] and refer the interested reader to [81] for an extension for timed models. For the collaboration building and its structural dynamism, a fully automatic checker for inductive invariants of graph transformation systems [49] presented in Section 4.2 and an extension supporting timed graph transformation systems [81] and an incremental checker [82] have been developed.

In our approach, a set of SPs describing the behavior relates to a GTS $S = (\mathcal{R}, TG)$ (cf Def. 13), where $\mathcal{R}$ is equipped with a priority function prio, that captures the possible changes of the graphs representing the state of a system. An additional set of side-effect free SPs represent forbidden graph patterns $\mathcal{F} = \{F_1, \ldots, F_n\}$ (cf. Def. 7) representing safety-violations of our system that have to be excluded. The related property $\Phi_{\mathcal{F}}$ is thus a conjunction of the forbidden patterns $(\neg F_1) \wedge \cdots \wedge (\neg F_n)$. We call G a *witness* for the property $\neg\Phi_{\mathcal{F}}$ if G in contrast matches any forbidden graph pattern $F \in \mathcal{F}$.

The graph property $\Phi_{\mathcal{F}}$ is an *operational invariant* of the GTS S if for a given initial graph G^0 and for all $G \in \mathsf{REACH}(S, \{G^0\})$ (cf. Def. 14) holds $G \models \Phi_{\mathcal{F}}$ (cf. [83]). However, checking operational invariants is undecidable as graph transformations with types are Turing-complete. We therefore instead tackle the problem whether the property $\Phi_{\mathcal{F}}$ is an *inductive invariant*. This is the case if for all graphs G typed over TG and for all rules $r \in \mathcal{R}$ holds that $G \models \Phi_{\mathcal{F}} \wedge G \overset{r}{\Rightarrow}_{\mathcal{R}} G'$ implies $G' \models \Phi_{\mathcal{F}}$. If we have an inductive invariant and the initial graph G^0 fulfills the graph property, then $\Phi_{\mathcal{F}}$ is also an *operational invariant* as inductive invariants are stronger than their operational counterparts.

We can reformulate the definition of an *inductive invariant* as follows to have a falsifiable form: a graph property $\Phi_{\mathcal{F}}$ is an inductive invariant of a GTS $S = (\mathcal{R}, TG)$ if and only if there exists no pair (G, r) of a graph G and a rule $r \in \mathcal{R}$ such that $G \models \Phi_{\mathcal{F}}$, $G \overset{r}{\Rightarrow}_{\mathcal{R}} G'$ and $G' \not\models \Phi_{\mathcal{F}}$. Such a pair (G, r) which witnesses the violation of graph property $\Phi_{\mathcal{F}}$ by rule r is then a *counterexample* for the initial hypothesis.

The invariant checker proceeds as follows for verifying statically that the absence of *forbidden patterns*[20] is preserved by a set of graph transformation rules with priorities: it is analyzed statically which kind of graph elements may be produced by a rule and then, it is checked how these created graph elements may be overlapped with the forbidden pattern $F \in \mathcal{F}$. In case that overlappings

[20] We explain the algorithm for patterns of the form $(F, \emptyset)$, denoted also as F. For an explanation of invariant checking for patterns of the form $\Pi = (F, \{N_i, i \in I\})$, we refer to [49].

are present, counterexamples can be constructed (by inverse rule application to the overlapping), expressing that if the rule is applied to a graph holding the remaining part of the forbidden pattern (*source pattern*), then after rule application the complete forbidden pattern F will be present (*target pattern*). Thereby, counterexamples may be rejected because of three reasons: (1) the source pattern comprises the precondition for a rule with a higher priority to be applicable (2) the source pattern comprises forbidden elements of one of the NACs of the rule (3) the source pattern comprises a forbidden pattern. In the first case, the rule with the higher priority ensures that the rule with lower priority under verification would not be applicable anyway. In the second case, similarly, the rule under verification would not be applicable because the source pattern comprises one of its NACs. In the latter case, the rule under verification would lead to a state comprising the forbidden pattern, if it is applied to a state which comprises the forbidden pattern already. If no counterexamples exist, it is ensured that a set of rules with priorities cannot be applied in such a way that they allow for transitions from states holding no forbidden pattern to states holding some forbidden pattern.

Example 25 (SP - Correct Model Refactorings). We have applied invariant checking in the context of in-place model transformations, in particular, refactorings. In this application context, invariant checking is very useful to investigate at design time if a rule-based refactoring specification could lead to inconsistent refactored models at runtime. We briefly review this approach here and we refer to [84] for a detailed description.

Fig. 42. A forbidden pattern (with predicate elements) specifying that no two methods with the same signature are members of the same class

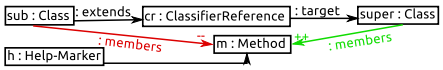


Fig. 43. Refactoring rule for the "Pull Up Method" refactoring

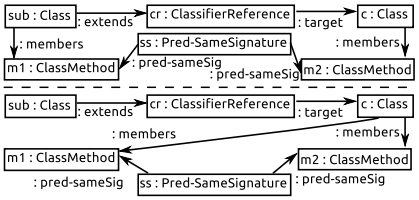


Fig. 44. Counterexample for the "Pull Up Method" refactoring

For example, for the consistency of the refactoring Pull Up Method [50], it is important that afterwards "no two Methods *sharing the same signature are contained in one* Class*". This well-formedness constraint is depicted as a forbidden pattern in Fig. 42. The types* Pred-SameSignature *and* Pred-NotSameSignature *mark that two* Methods *have the same or a different signature, respectively. If we run our Invariant Checker with the well-formedness constraint shown in Fig. 42 and the refactoring rule depicted in Fig. 43 the verification result is likely to be a counterexample as the one shown in Fig. 44. The reason that the refactoring rule is unsafe is that the rule completely ignores the* Pred-SameSignature *nodes. If we change the rule to require the existence of*

a Pred-NotSameSignature and forbid the existence of a Pred-SameSignature node, the rule is safe.

Example 26 (RailCab - Invariant Checking for Structural Dynamics). A further example successfully applying our Invariant Checker is the Railcab system. Obviously, this system is hard to check using other verification techniques, such as model checking, as the system's potential state space would be very large and it is hard to identify a valid initial state.

To ensure that the Railcab system is safe, we have to verify that Shuttles never collide. A collision can be expressed by a forbidden pattern, as shown in Fig. 22(e). An invariant that is implied in this specification of the Railcab system is that a Shuttle will never try to go to a Track occupied by another Shuttle without making sure the other Shuttle is moving (see Fig. 22(a)). Along with several structural constraints restricting cardinalities, these two forbidden patterns form the set $\mathcal{F}$.

The complete set of rules is given through the Story-Pattern shown in Fig. 22(d), 22(b), 21 and 22(c). For a short description of the rules we refer to Example 16. For the RailCab system to be safe it is required that rules for the creation of the DistanceCoordinationPattern protocol (cf. Fig. 22(b)) preempts all move-rules. Therefore this rule has the highest priority. Due to space limitations we have omitted the rule that destroys the DistanceCoordinationPattern protocol, however the complete example including a more detailed explanation of the rules can be found in [49]. Using the rules mentioned above we have verified in[49] that the RailCab system is collision free.

The Invariant Checker has be employed for several variants of the reported case studies for MDE including the rules of the industrial case study [61] and models with structural dynamics. The current implementation of our Invariant Checker tool is constantly improved and is available on request only. For further information on the research prototype, please contact us at `contact@mdelab.org`.

4.3 Model Checking

In contrast to the previously described static analysis methods model checking [85] is a dynamic verification technique that explores the state space of the system under consideration. In case of a graph transformation system S (see Def. 13) with a set of initial graphs I, a related labeled transition system as specified in Def. 15 as state space for model checking. However, model checking can only be efficiently applied if the state space is finite, which is not necessarily the case for graph transformation systems where nodes and edges can be dynamically created. In addition, such a finite state space can only be build when the initial graph or set of initial graphs is known. If a meaningful criterion to limit the explored state space exists, bounded model checking [86] can be used to investigate only the related finite subset of the overall state space. Other approaches use symbolic representations of the state space to overcome this limitation [87].

A desired property is usually expressed as a condition for all reachable states or in form of a sequence property by some form of temporal-logic. An example of such a temporal-logic is the Computation-Tree-Logic (compare [85]). The state space is analyzed and depending on the given property a counterexample is

derived as a witness in the case the property is violated. Accordingly, also for graph transformation systems approaches for model checking exist [72].

In [49] we used the particular tool GROOVE [88,73]. To be able to apply model checking GROOVE requires a GTS according to Def. 13 including an initial graph and supports the dangling-edge-collecting approach[21] (see Def. 10). Moreover, GROOVE allows for generating a minimal labeled transition system in the sense of Def. 15. Atomic properties can be expressed in GROOVE in form of side-effect free rules that are checked for applicability on a given graph state. This conforms to properties as given in Def. 7 consisting of a required pattern, where the pattern consists of the LHS of the side-effect free rule. If the required pattern can be matched (see Def. 6) in a specific graph state, the property represented by the rule is fulfilled for that state. These atomic properties can then be used inside a Computation-Tree-Logic (CTL) formula. GROOVE then allows automatically exploring the reachable states via the transition relation of the given GTS as well as automatically evaluating the given CTL formulae. In case an example respectively counterexample in form of a witness can be found, GROOVE provides an alternating sequence of states and rule applications leading to or directly representing the witness.

The Henshin tool [43] also provides model checking capabilities for graph transformation systems. Henshin is based on typed graphs and supports both the dangling-edge-collecting approach and the conservative approach. State spaces generated by Henshin can be checked for given properties. Model checking is supported using external, third party verification tools, such as mCRL2 and CADP.[22] Similarly to the approach implemented in GROOVE, the specification of atomic properties is based on matched graph patterns.

Example 27 (RailCab - SP - Model Checking for Structural Dynamics). In [49] we used the GTS model checker GROOVE [88] and compared the results of GROOVE with those of the approach described in Section 4.2. We have further investigated the complexity of the different analysis methods. To be able to do so the rules describing the application example of the Railcab system depicted in Fig. 22(d), 22(b) and 22(c) beneath others have been imported into GROOVE. We analyzed our model in GROOVE, using the forbidden pattern collision *depicted in Fig. 22(e). The outcome of the investigation was that models of moderate size can be effectively analyzed and accordingly we have been able to apply model checking in GROOVE on systems with smaller topologies. However, experiments on a Railcab system with more than 15 tracks turned out to be to complex and leading to a large state space for which it was rarely possible to apply model checking using GROOVE in a efficient way. For more details about the used graph rules, the analyzed properties as well as the evaluation results concerning the complexity of the different approaches compare [49].*

[21] Note that given a type graph additional NACs can be derived such that the adjusted rule in the dangling-edge-collecting approach behaves like the original one in the conservative approach. The additional NACs simply ensure that no dangling edges can exist if the rule is applicable.

[22] See `http://www.mcrl2.org` and `http://www.inrialpes.fr/vasy/cadp/` for more information about mCRL2 and CADP.

5 Discussion

In order to discuss the benefits of graph transformations for MDE, the modeling of structure dynamics, and models at runtime, we will at first look at the options that exist for each of the areas and finally look into their combination.

In MDE, the models are not only a byproduct but become the core carrier of the higher-level knowledge about the software. Model transformation to partially generate subsequent models and code generation result in a situation where, if properly done, the code and the models remain consistent. Thus, required classical adaptation steps can take advantage of the up-to-date higher-level knowledge about the software. Therefore, MDE promises to better support the long-term evolution of the software. Today, the principles of MDE are to employ *meta-models* to define the modeling languages and to use related techniques for model operations such as model transformations or consistency checks (e.g., QVT, ATL, or OCL) that take advantage of an underlying *meta-meta-model* and considerably ease to develop the required model operations. We presented in particular graph transformation based techniques such as SPs for model manipulation and checking models in Section 3.2 and on model transformation and incremental model synchronization based on TGGs in Section 3.3.

Besides the evolution of the software, also the *co-adaptation* resp. *language evolution* is a fact that matters for the long-term evolution (cf. [89,90]). Typically, this leads to a need for transformations to adjust the models but also *higher-order transformations* to adjust model operation (e.g., model transformations). Due to the employed interpreter for SPs [45] presented in Section 3.2 that is also used as a basis for executing the derived TGG rules, our techniques support higher order transformations of the transformation models at runtime.

Today, most existing work on (semi-)automatic correctness verification of model operations only permits to prove that a particular result of a model transformation is correct with respect to the input [91,92]. In our own work partially presented in Section 4 we were in contrast able to derive guarantees that hold for all possible results of a model operation with respect to the input. We presented an approach employing a theorem prover for model transformations with TGGs in [76] that show behavioral equivalence and an automated verification technique for refactorings with SPs [84] that permit to guarantee that required properties are preserved by the refactorings. [93] approach the first problem by comparing two proof techniques with respect to chances of successful mechanization. [94] tackles the problem of verifying required properties for model transformations specified with TGGs by proposing a method to derive OCL invariants from TGGs in order to enable their automated verification and analysis.

Structure dynamics is required to realize complex capabilities such as self-healing, self-configuring etc. on top of related basic capabilities such as self-awareness and context-awareness. A proper combination of the higher level capabilities then finally leads to the capability of self-managing or more general self-adaptive software [12,21,18]. Suggestions for the construction of such system include frameworks like the *Rainbow* approach [22] that addresses the construction of self-adaptive software systems by providing reusable elements for

the adaptation engine in order to reduce development efforts. The *MUSIC* approach [95], the context-aware and quality of service aware architectural variability of the core function is specified by models during development. Likewise, in [96] modeling and code generation are employed to simplify the development of self-adaptive software, while any further changes to the generated software requires re-modeling and re-generation steps.

Our own work has resulted in the Mechatronic UML approach [97] for the model-driven development of self-optimizing embedded real-time systems. It employs graph transformation systems and hybrid statecharts to reconfigure hierarchical component-based systems. For the ad hoc formation of collaborations between mechatronic systems (e.g., vehicles that form convoys) or other forms of structural dynamism graph transformation systems are employed [49,81,98] including also first ideas for exchanging models at runtime [99].

A first direction for assurance that is employed for self-adaptive systems is runtime verification [100]. Available techniques for the assurance of self-adaptive systems using model checking either restrict their focus to separate adaptation steps [101,102,103] or assume a decoupling of the adaptation decision from the local functional state [104] in order to achieve scalability. More fundamental work is studying properties of graph transformation systems [105] for characteristics which must hold for self-healing, a special case of self-adaptive behavior.

For our Mechatronic UML approach and and ad hoc real-time collaborations between multiple complex subsystems a compositional verification approach has been developed [106]. For the collaboration building and its structural dynamism, a fully automatic checker for inductive invariants of graph transformation systems [49] presented in Section 4.2 and an extension supporting timed graph transformation [81] has been developed. Finally, the combination of the verification results for inductive invariants for graph transformation models and the compositional verification of the collaboration of multiple roles represented by real-time statecharts has been presented in [98]. These results have also led to studies for self-adaptive software in general and first general results for modeling and verifying them [107]. Furthermore, also an incremental invariant checker [82] has been developed which allows to reduce the effort for performing checks when the behavior has changed at runtime.

There is a lack of work on systematically developing *causal connections* between a runtime model and the running system to reflect changes of the system in the model, and changes of the model in the system. Usually, manually developed solutions are employed, or some work tries to simplify the development by increasing the level of automation for implementing a causal connection [108]. Most approaches focus on having appropriate abstractions (runtime models) of a running system and on connecting a model and a system. Thereby, monitoring the system and effecting adaptations to the system are supported. However, the approaches do not completely work incrementally, like [109,110,111] that entirely compares two models to identify the changes to be executed. However, the available techniques are often very demanding and thus result in a too high overhead while being not responsive enough (cf. need for an incremental handling

at runtime in general as discussed in the sidebar of [24]). In our work [62,63,64], the monitoring [63] and effecting [64] stages of the feedback loop can handle the causal connection including an abstraction step automatically. Furthermore, our own model synchronization techniques [58], that we employ at runtime, works incrementally as outlined in Section 3.2 and thus enable responsive solutions.

Existing approaches address assurances through validation and verification of adaptation mechanisms, like testing conceivable adaptation results at the level of the runtime model before executing it to the running system. First approaches employing certain techniques have been proposed for constraint checking [110,111], simulation [112], and model checking [113]. However, there is a lack of work on assurances for the runtime models themselves as well as for employing incremental MDE techniques working on these models. We think that the in Section 4 presented results provide a solid basis for a more subtantial coverage of this problem.

As we pointed out in [26], a solution is required where adaptation steps in the construction environment and in the runtime environment happen in an integrated manner. Consequently, the integrated co-existence of self-adaptation and classical adaptation including dependencies between them also have to be addressed. In this direction, only few preliminary ideas exist [110] and a more thorough approach towards integrating these ideas is required and we think that due to the in this paper outlined support for the different cases graph transformations are a good candidate as a foundation for such efforts.

6 Conclusion

As outlined in the paper graph transformations provide a solid basis for related techniques such as SP, Story Diagrams and TGGs such that we cannot only address MDE, structural dynamics as well as models at runtime using these techniques but also analyze important properties for the resulting systems. Due to the fact that all the developed techniques share the underlying concepts of graph transformations, they do not only provide basic building blocks for MDE, systems with structural dynamics, and models at runtime, but furthermore provide a basis to integrate these directions into a single coherent approach. Therefore, graph transformations seem to be a good candidate to provide a solid foundation for approaching the evolution challenge.

References

1. Lehman, M.M., Belady, L.A., eds.: Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA (1985)
2. Lehman, M.M.: Laws of Software Evolution Revisited. In Montangero, C., ed.: Software Process Technology, 5th European Workshop, EWSPT'96, Nancy, France, October 9-11, 1996, Proceedings. Volume 1149 of Lecture Notes in Computer Science., Springer (1996) 108–124

3. Parnas, D.L.: Software aging. In: ICSE '94: Proceedings of the 16th international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1994) 279–287
4. Mens, T., Demeyer, S.: Software Evolution. Springer (2008)
5. Martin, R., Osborne, W.: Guidance of Software Maintenance. Technical Report NBS Pub. 500-129, U.S. Nat. Bureau of Standards (December 1983)
6. Chikofsky, E.J., II, J.H.C.: Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software **7**(1) (January 1990) 13–17
7. Mens, T., Tourwe, T.: A survey of software refactoring. Software Engineering, IEEE Transactions on **30**(2) (February 2004) 126 – 139
8. Kephart, J.O., Chess, D.: The Vision of Autonomic Computing. Computer **36**(1) (January 2003) 41–50
9. Brown, P., Bovey, J., Chen, X.: Context-aware applications: from the laboratory to the marketplace. Personal Communications, IEEE **4**(5) (October 1997) 58 –64
10. Northrop, L., Feiler, P.H., Gabriel, R.P., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006)
11. Sztipanovits, J., Karsai, G., Bapty, T.: Self-adaptive software for signal processing. Commun. ACM **41**(5) (1998) 66–73
12. Oreizy, P., Gorlick, M.M., Taylor, R., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An Architecture-Based Approach to Self-Adaptive Software. IEEE Intelligent Systems **14**(3) (June 1999) 54–62
13. Musliner, D.J., Goldman, R.P., Pelican, M.J., Krebsbach, K.D.: Self-Adaptive Software for Hard Real-Time Environments. IEEE Inteligent Systems **14**(4) (July 1999)
14. Robertson, P., Shrobe, H.E., Laddaga, R., eds.: Self-Adaptive Software, First International Workshop, IWSAS 2000, Oxford, UK, April 17-19, 2000, Revised Papers. In Robertson, P., Shrobe, H.E., Laddaga, R., eds.: IWSAS. Volume 1936 of Lecture Notes in Computer Science., Springer (2001)
15. Laddaga, R., Robertson, P., Shrobe, H.E., eds.: Self-Adaptive Software, Second International Workshop, IWSAS 2001, Balatonfüred, Hungary, May 17-19, 2001 Revised Papers. In Laddaga, R., Robertson, P., Shrobe, H.E., eds.: IWSAS. Volume 2614 of Lecture Notes in Computer Science., Springer (2003)
16. Cheng, B., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Cheng, B., Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science., Springer (2009) 1–26
17. Cheng, B., Giese, H., Inverardi, P., Magee, J., Lemos, R.D., eds.: Software Engineering for Self-Adaptive Systems. 1 edn. Volume 5525 of Lecture Notes in Computer Science. Springer, Berlin (2009)
18. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2) (2009) 1–42
19. Maes, P.: Concepts and experiments in computational reflection. In: Conference proceedings on Object-oriented programming systems, languages and applications. OOPSLA '87, New York, NY, USA, ACM (1987) 147–155

20. Brun, Y., Serugendo, G.D.M., Gacek, C., Giese, H.M., Kienle, H.M., Litoiu, M., Müller, H.A., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops. In Cheng, B., Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science., Springer (2009) 48–70
21. Kramer, J., Magee, J.: Self-Managed Systems: an Architectural Challenge. In: FOSE '07: 2007 Future of Software Engineering, Washington, DC, USA, IEEE Computer Society (2007) 259–268
22. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer **37**(10) (2004) 46–54
23. Georgas, J.C., Hoek, A., Taylor, R.N.: Using Architectural Models to Manage and Visualize Runtime Adaptation. Computer **42**(10) (2009) 52–60
24. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. Computer **42**(10) (2009) 22–27
25. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP workshop on Future of software engineering research (FoSER '10), New York, NY, USA, ACM (2010) 17–22
26. Gacek, C., Giese, H., Hadar, E.: Friends or Foes? – A Conceptual Analysis of Self-Adaptation and IT Change Management. In: Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS"08), Leipzig, Germany, ACM Press (May 2008)
27. Baresi, L., Heckel, R.: Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G., eds.: Graph Transformation. Volume 2505 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2002) 402–429
28. Rensink, A.: The Edge of Graph Transformation – Graphs for Behavioural Specification. In Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B., eds.: Graph Transformations and Model-Driven Engineering. Volume 5765 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2010) 6–32
29. Giese, H., Klein, F.: Autonomous Shuttle System Case Study. In Leue, S., Systä, T., eds.: Scenarios: Models, Algorithms and Tools. Volume 3466 of Lecture Notes in Computer Science (LNCS). Springer Verlag (April 2005) 90–94
30. International Telecommunication Union, I.: ITU-T Recommendation Z.100: Specification and Description Language (SDL). (2002)
31. Schäfer, W., Wagner, R., Gausemeier, J., Eckes, R.: An Engineer's Workstation to support Integrated Development of Flexible Production Control Systems. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of Lecture Notes in Computer Science (LNCS). Springer Verlag (2004) 48–68
32. Habel, A., Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19** (2009) 1–52
33. Ehrig, H., Habel, A., Lambers, L., Orejas, F., Golas, U.: Local Confluence for Rules with Nested Application Conditions. In: Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10). Volume 6372. (2010) 330–345
34. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1. World Scientific, Singapore (1999)
35. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)

36. Löwe, M., Korff, M., Wagner, A. In: An algebraic framework for the transformation of attributed graphs. John Wiley and Sons Ltd., Chichester, UK, UK (1993) 185–199
37. Plump, D., Steinert, S.: Towards graph programs for graph algorithms. In: Proceedings of Intern. Conf. on Graph Transformation (ICGT'10), Springer LNCS (2004) 128–143
38. Orejas, F., Lambers, L.: Delaying Constraint Solving in Symbolic Graph Transformation. In: Proceedings of Intern. Conf. on Graph Transformation (ICGT' 10). Volume 6372., Springer (2010) 43–58
39. Bardohl, R., Ehrig, H., de Lara, J., Taentzer, G.: Integrating Meta Modelling with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Wermelinger, M., Margaria-Steffens, T., eds.: Proc. Fundamental Aspects of Software Engineering 2004. Volume 2984 of Lecture Notes in Computer Science., Springer (2004)
40. Golas, U., Lambers, L., Ehrig, H., Orejas, F.: Attributed graph transformation with inheritance: Efficient conflict detection and local confluence analysis using abstract critical pairs. Theoretical Computer Science **424** (2012) 46 – 68
41. Gorp, P.V., Mazanek, S., Rose, L., eds.: Proceedings Fifth Transformation Tool Contest. Volume 74 of EPTCS. (2011)
42. Ermel, C., Rudolf, M., Taentzer, G.: The AGG approach: language and environment. In Ehrig, H., Engels, G., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools. World Scientific Publishing Co., Inc., River Edge, NJ, USA (1999) 551–603
43. Arendt, T., Bierman, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In Petriu, Rouquette, Haugen, eds.: MoDELS'10: Proc. 13th international conference on Model Driven Engineering Languages and Systems. Volume 6394 of Lecture Notes in Computer Science., Springer (2010) 121–135
44. Schürr, A., Winter, A., Zündorf, A.: The PROGRES Approach: Language and Environment. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, volume 2 - Application, Languages and tools. World Scientific, Singapore (1999) 487–546
45. Giese, H., Hildebrandt, S., Seibel, A.: Improved Flexibility and Scalability by Interpreting Story Diagrams. In Magaria, T., Padberg, J., Taentzer, G., eds.: Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009). Volume 18., Electronic Communications of the EASST (2009)
46. Seibel, A., Neumann, S., Giese, H.: Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. Software and System Modeling **9**(4) (2010) 493–528 10.1007/s10270-009-0146-z.
47. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proc. of the 6^th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. LNCS 1764, Springer Verlag (1998) 296–309
48. Zndorf, A.: Rigorous Object Oriented Software Development with Fujaba. (2002) Draft Version 0.3. (available under http://www.se.eecs.uni-kassel.de/se/fileadmin/se/publications/Zuen02.pdf).
49. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the

28^th International Conference on Software Engineering (ICSE), Shanghai, China, ACM Press (2006)
50. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
51. Krasnogolowy, A., Hildebrandt, S., Wätzoldt, S.: Flexible Debugging of Behavior Models. In: Proceedings of 2012 IEEE International Conference on Industrial Technology (ICIT), IEEE (2011)
52. Stürmer, I., Kreuz, I., Schäfer, W., Schürr, A.: Enhanced Simulink/Stateflow Model Transformation: The MATE Approach. Proc. of MathWorks Automotive Conference (MAC 2007), Dearborn (MI), USA (2007)
53. Schafe, S.: Objektorientierte Entwurfsmethoden. Addison-Wesley (1994)
54. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. In: Proceedings of MoDeVVa 2010, Models Workshop on Model-Driven Engineering, Verification and Validation, Oslo, Norway (2010)
55. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars. Technical Report 37, Hasso Plattner Institute at the University of Potsdam (2010)
56. Larchevêque, J.M.: Optimal incremental parsing. ACM Trans. Program. Lang. Syst. **17**(1) (1995) 1–15
57. Giese, H., Wagner, R.: Incremental Model Synchronization with Triple Graph Grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy. Volume 4199 of Lecture Notes in Computer Science (LNCS)., Springer Verlag (October 2006) 543–557
58. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling (SoSyM) **8**(1) (28 March 2009)
59. Giese, H., Hildebrandt, S.: Incremental Model Synchronization for Multiple Updates. In: Proceedings of the 3rd International Workshop on Graph and Model Transformations, May 12, 2008, Leipzig, Germany. Volume Proceedings of GraMoT'08, May 12, 2008, Leipzig, Germany., ACM Press (2008)
60. Giese, H., Hildebrandt, S.: Efficient Model Synchronization of Large-Scale Models. Technical Report 28, Hasso Plattner Institute at the University of Potsdam (2009)
61. Giese, H., Neumann, S., Hildebrandt, S.: Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B., eds.: Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday. Volume 5765 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2010) 555–579
62. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems. In: Proceedings of the 6th IEEE/ACM International Conference on Autonomic Computing and Communications (ICAC 2009), Barcelona, Spain, ACM (2009) 67–68
63. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In Ghosh, S., ed.: Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers. Volume 6002 of Lecture Notes in Computer Science (LNCS). Springer-Verlag (2010) 124–139

64. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 5th Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010) at the 32nd IEEE/ACM International Conference on Software Engineering (ICSE 2010), Cape Town, South Africa, ACM (2010) 39–48
65. Vogel, T., Giese, H.: Requirements and Assessment of Languages and Frameworks for Adaptation Models. In Kienzle, J., ed.: Models in Software Engineering, Workshops and Symposia at MoDELS 2011, Wellington, New Zealand, October 16-21, 2011, Reports and Revised Selected Papers. Volume 7167 of Lecture Notes in Computer Science (LNCS). Springer-Verlag (2012) 166–181
66. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In Dingel, J., Solberg, A., eds.: Models in Software Engineering, Workshops and Symposia at MODELS 2010, Oslo, Norway, October 3-8, 2010, Reports and Revised Selected Papers. Volume 6627 of Lecture Notes in Computer Science (LNCS). Springer-Verlag (2011) 224–238
67. Vogel, T., Giese, H.: A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), IEEE Computer Society (2012)
68. Hausmann, J., Heckel, R., Taentzer, G.: Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In: Proc. of Int. Conference on Software Engineering 2002, Orlando, USA, IEEE Computer Society (2002) 105–115
69. Engels, G., Güldali, B., Lohmann, M.: Towards Model-Driven Unit Testing. In Kühne, T., ed.: Models in Software Engineering. Volume 4364 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2007) 182–192
70. Ehrig, K., Küster, J., Taentzer, G., Winkelmann, J.: Generating Instance Models from Meta Models. In: Formal Methods for Open Object-Based Distributed Systems. Volume 4037 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 156–170
71. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: In FASE 2005, Springer (2005) 34–48
72. Rensink, A., Schmidt, A., Varró, D.: Model checking graph transformations: A comparison of two approaches. In Ehrig, H., Engels, G., Parise-Presicce, F., Rozenberg, G., eds.: International Conference on Graph Transformations (ICGT). Volume 3256 of Lecture Notes in Computer Science., Berlin, Springer Verlag (2004) 226–241
73. Kastenberg, H., Rensink, A.: Model checking dynamic states in groove. In Valmari, A., ed.: Model Checking Software (SPIN), Vienna, Austria. Volume 3925 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (2006) 299–305
74. Strecker, M.: Modeling and Verifying Graph Transformations in Proof Assistants. In Mackie, I., Plump, D., eds.: International Workshop on Computing with Terms and Graphs (TERMGRAPH), Braga/Portugal, 31/03/2007. Volume 203 of Electronic Notes in Theoretical Computer Science., http://www.elsevier.com, Elsevier Science (2008) 135–148
75. Pennemann, K.H.: Resolution-like theorem proving for high-level conditions. In: Graph Transformations (ICGT'08). Volume 5214 of Lecture Notes in Computer Science., Springer-Verlag (2008) 289–304
76. Giese, H., Glesner, S., Leitner, J., Schäfer, W., Wagner, R.: Towards Verified Model Transformations. In Hearnden, D., Süß, J., Baudry, B., Rapin, N., eds.: Proc. of the $3^r d$ International Workshop on Model Development, Validation and Verification (MoDeV^{2}a), Genova, Italy, Le Commissariat à l'Energie Atomique - CEA (October 2006) 78–93

77. Baldan, P., Corradini, A., König, B.: A Static Analysis Technique for Graph Transformation Systems. In: Proc. CONCUR. Volume 2154 of LNCS., Springer (2001) 381–395
78. Habel, A., Pennemann, K.H., Rensink, A.: Weakest Preconditions for High-Level Programs. In: Graph Transformations (ICGT'06). Volume 4178 of Lecture Notes in Computer Science., Springer-Verlag (2006) 445–460
79. Poskitt, C.M., Plump, D.: A hoare calculus for graph programs. In: Proceedings of the 5th international conference on Graph transformations. ICGT'10, Berlin, Heidelberg, Springer-Verlag (2010) 139–154
80. Lambers, L.: Certifying Rule-Based Models using Graph Transformation. PhD thesis, Technische Universität Berlin (2010) Also as book available: Südwestdeutscher Verlag für Hochschulschriften ISBN: 978-3-8381-1650-1.
81. Becker, B., Giese, H.: On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In: In Proc. of 11th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC), IEEE Computer Society Press (5-7 May 2008) 203–210
82. Becker, B., Giese, H.: Incremental Verification of Inductive Invariants for the Run-Time Evolution of Self-Adaptive Software-Intensive Systems. In: Proc. 1st International Workshop on Automated engineeRing of Autonomous and run-tiMe evolvIng Systems (ARAMIS), IEEE Computer Society Press (2008) 33–40
83. Charpentier, M.: Composing Invariants. In: Proc. of International Symposium of Formal Methods Europe. Volume 2805 of Lecture Notes in Computer Science., Springer (2003) 401–421
84. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative Development of Consistency-Preserving Rule-Based Refactorings. In Cabot, J., Visser, E., eds.: Theory and Practice of Model Transformations, Fourth International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings. Volume 6707 of Lecture Notes in Computer Science., Springer / Heidelberg (2011) 123–137
85. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2002)
86. Biere, A., Cimatti, A., Clarke, E., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003)
87. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. **98** (June 1992) 142–170
88. Rensink, A.: Towards Model Checking Graph Grammars. In Leuschel, M., Gruner, S., Presti, S.L., eds.: 3rd Workshop on Automated Verification of Critical Systems (AVoCS). Technical Report DSSE–TR–2003–2, University of Southampton (2003) 150–160
89. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference, Washington, DC, USA, IEEE Computer Society (2008) 222–231
90. Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice. IWMCP '11, New York, NY, USA, ACM (2011) 30–38
91. Narayanan, A., Karsai, G.: Verifying Model Transformations by Structural Correspondence. Electronic Communications of the EASST: Graph Transformation and Visual Modeling Techniques 2008 **10** (2008)
92. Varró, D., Pataricza, A.: Automated Formal Verification of Model Transformations. In Jürjens, J., Rumpe, B., France, R., Fernandez, E.B., eds.: CSDUML

2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop. Number TUM-I0323 in Technical Report, Technische Universitat Munchen (September 2003) 63–78
93. Hülsbusch, M., König, B., Rensink, A., Semenyak, M., Soltenborn, C., Wehrheim, H.: Showing Full Semantics Preservation in Model Transformation - A Comparison of Techniques. In D. Méry, S.M., ed.: IFM 2010. Volume 6396., Springer Verlag Berlin-Heidelberg (2010) 183–198
94. Cabot, J., Clarisó, R., Guerra, E., Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. J. Syst. Softw. **83**(2) (2010) 283–302
95. Rouvoy, R., Barone, P., Ding, Y., Eliassen, F., Hallsteinsen, S., Lorenzo, J., Mamelli, A., Scholz, U.: MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science (LNCS). Springer Berlin / Heidelberg (2009) 164–182
96. Bencomo, N., Blair, G.: Using Architecture Models to Support the Generation and Operation of Component-Based Adaptive Systems. In Cheng, B., de Lemos, R., Giese, H., Inverardi, P., Magee, J., eds.: Software Engineering for Self-Adaptive Systems. Volume 5525 of Lecture Notes in Computer Science (LNCS). Springer Berlin / Heidelberg (2009) 183–200
97. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer (STTT) **10**(3) (June 2008) 207–222
98. Giese, H.: Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In Kordon, F., Sztipanovits, J., eds.: Reliable Systems on Unreliable Networked Platforms - 12th Monterey Workshop 2005 . Laguna Beach, CA, USA, September 22-24,2005 . Revised Selected Papers. Volume 4322 of Lecture Notes in Computer Science., Springer Verlag (2007) 258–280
99. Burmester, S., Giese, H.: Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML. In: Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA, IEEE Computer Society Press (September 2005) 109–116
100. Goldsby, H.J., Cheng, B., Zhang, J.: AMOEBA-RT: Run-Time Verification of Adaptive Software. In: Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers. Springer-Verlag, Berlin, Heidelberg (2008) 212–224
101. Zhang, J., Cheng, B.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software **79**(10) (2006) 1361 – 1369 Architecting Dependable Systems.
102. Zhang, J., Cheng, B.: Model-based development of dynamically adaptive software. In: ICSE '06: Proceeding of the 28th international conference on Software engineering, New York, NY, USA, ACM (2006) 371–380
103. Zhang, J., Goldsby, H.J., Cheng, B.: Modular verification of dynamically adaptive systems. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development, New York, NY, USA, ACM (2009) 161–172
104. Adler, R., Schaefer, I., Trapp, M., Poetzsch-Heffter, A.: Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems.

ACM Transactions on Embedded Computing Systems **10** (January 2011) 20:1–20:39
105. Ehrig, H., Ermel, C., Runge, O., Bucchiarone, A., Pelliccione, P.: Formal Analysis and Verification of Self-Healing Systems. In Rosenblum, D.S., Taentzer, G., eds.: Fundamental Approaches to Software Engineering, 13th International Conference, FASE 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010 Paphos, Cyprus, March 20-28, 2010. Proceedings. Volume 6013 of Lecture Notes in Computer Science., Springer (2010) 139–153
106. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the Compositional Verification of Real-Time UML Designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11), ACM Press (2003) 38–47
107. Becker, B., Giese, H.: Modeling of Correct Self-Adaptive Systems: A Graph Transformation System Based Approach. In: CSTST '08: Proc. 5th Intl. Conference on Soft Computing as Transdisciplinary Science and Technology, ACM Press (2008) 508 – 516
108. Song, H., Xiong, Y., Chauvel, F., Huang, G., Hu, Z., Mei, H.: Generating Synchronization Engines between Running Systems and Their Model-Based Views. In Ghosh, S., ed.: Models in Software Engineering, Workshops and Symposia at MODELS 2009, Denver, CO, USA, October 4-9, 2009, Reports and Revised Selected Papers. Volume 6002 of Lecture Notes in Computer Science (LNCS). Springer-Verlag (2010) 140–154
109. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M., eds.: Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MODELS). Volume 5301 of LNCS., Springer (2008) 782–796
110. Morin, B., Barais, O., Jézéquel, J.M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. Computer **42**(10) (2009) 44–51
111. Morin, B., Barais, O., Nain, G., Jézéquel, J.M.: Taming Dynamically Adaptive Systems using models and aspects. In: ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2009) 122–132
112. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In Schürr, A., Selic, B., eds.: MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. Volume 5795 of LNCS., Berlin, Heidelberg, Springer-Verlag (2009) 606–621
113. Inverardi, P., Mori, M.: Model checking requirements at run-time in adaptive systems. In: Proceedings of the 8th workshop on Assurances for self-adaptive systems. ASAS '11, New York, NY, USA, ACM (2011) 5–9