

# Style-Based Refinement of Dynamic Software Architectures <sup>\*</sup>

Luciano Baresi<sup>(+)</sup>, Reiko Heckel<sup>(\*)</sup>, Sebastian Thöne<sup>(#)</sup>, and Dániel Varró<sup>(§)</sup>

<sup>(+)</sup> Politecnico di Milano  
Dipartimento di Elettronica e Informazione  
Italy – baresi@elet.polimi.it

<sup>(#)</sup> University of Paderborn  
International Graduate School  
Germany – seb@upb.de

<sup>(\*)</sup> University of Paderborn  
Department of Computer Science  
Germany – reiko@upb.de

<sup>(§)</sup> Budapest University of Technology and Economics  
Department of Measurement and Information Systems  
Hungary – varro@mit.bme.hu

## Abstract

*In this paper, we address the correct refinement of abstract architectural models into more platform-specific representations. We consider the challenging case of dynamic architectures which can perform run-time reconfigurations. For this purpose, the underlying platform has to provide the necessary reconfiguration mechanisms. To conceptually model such platforms including provided reconfiguration mechanisms, we use architectural styles formalized by graph transformation rules. Based on formal refinement relations between abstract and platform-specific styles, we can then investigate how to realize business-specific scenarios on a certain platform by automatically deriving refined, platform-specific reconfiguration scenarios.*

## 1. Introduction

Software architectures play an important role in software development [25]. As abstract models of the run-time structure they help bridging the gap between user requirements and implementation. In the context of e-business, self-healing, or mobile systems, *dynamic architectures* gain more and more importance. They represent systems that do not simply consist of a fixed, static structure, but can react to certain requirements or events by run-time reconfiguration of its components and connections. The availability of those reconfiguration operations depends on the chosen run-time platform which has to support the desired modifications.

The development of such dynamic architectures is a complex task which is usually driven by a step-wise refinement approach. The software architect derives a first abstract model of the architecture from the user requirements. This model mainly covers the functional aspects and business-related components. Later in the design process, more and more non-functional requirements like security concepts and implementation-specific aspects are integrated into the core functionality. This leads to a sequence of refined architectures down to the real system design for implementation.

A recent example of this general modeling principle is the *Model-Driven Architecture (MDA)* [22] put forward by the OMG. Here, platform-specific details are initially ignored at the model-level to allow for maximum portability. Then, these platform-independent models are refined by adding details required to map to a given target platform. Thus, at each refinement level, one imposes more assumptions on the resources, constraints, and services of the chosen platform.

The goal of this paper is to define a notion of refinement which preserves both semantic correctness and platform consistency. This means that a concrete architecture must satisfy the same requirements as the abstract architecture, and that it must be consistent with constraints and mechanisms imposed by the chosen target platform.

In software architecture research, *architectural styles* are used to describe families of architectures by common resource types, configuration patterns and constraints [2]. We already proposed in [5] to consider the restrictions imposed by a certain choice of platform as an architectural style. Moreover, to account for component interactions and platforms that support dynamic reconfigurations, we extend the classical notion of ar-

---

<sup>\*</sup> Research partially supported by the European Research Training Network *SegraVis* (on *Syntactic and Semantic Integration of Visual Modelling Techniques*)

architectural style, which is restricted to structural constraints, by also describing platform-specific communication and reconfiguration mechanisms.

As described in [5], the architectural styles are formalized as graph transformation systems including architectural types, constraints, and graph transformation rules. Based on that, we define refinement relations between abstract and concrete styles in this paper. They enable us to check for correct refinement of two given architectures. We do not only consider *structural* refinements of fixed configurations but also *behavioral* refinement, which means refining abstract scenarios of component interactions and reconfigurations into platform-specific scenarios.

Since refinements are often tedious and error-prone, a further goal is to (semi-)automate the construction of desired refinements. Indeed, the maximum gain of reusing platform-independent models is achieved if the mapping to various target platforms can be automated. For this purpose, we propose a formulation of the behavioral refinement problem as a reachability problem which can be solved by classical graph transformation and model checking tools. This allows, within the usual limitations, an automated refinement of architectures.

The rest of this paper is organized as follows. We survey related work in Sect. 2. In Sect. 3, we introduce a short running example. Then, we review the basics of modeling architectural styles based on graph theory (Sect. 4) and describe the analysis of reachability properties as a basis for behavioral refinement (Sect. 5). In Sect. 6 we use this formal framework to define our concepts for structural and especially behavioral refinement. Sect. 7 concludes the paper.

## 2. Related work

Our work mainly interferes with three different research directions: architecture description languages, graph transformations, and architectural refinement.

There are many proposals for Architecture Description Languages (ADLs) like Rapide [19], Wright [3], or Darwin [20]. To model dynamic architectures, several approaches apply graph transformation [16, 17, 18, 26, 28] using this formal framework to reason about the consistency of reconfiguration operations with structural constraints and component interaction. Our paper is in this tradition, but it combines the formal approach with the notion of style-based refinement.

Le Métayer [18] describes architectures by graphs and the valid graphs of an architectural style by a graph grammar. Reconfiguration is described by conditional graph rewriting rules. By static type checking, the rewriting rules are proved to be consistent with

the respective style. In comparison to our work, his graphs represent computational entities but no connectors, specifications, or other resources. And, instead of a graph grammar, we use a declarative type graph to define the valid graphs of the architectural style.

Wermelinger and Fiadeiro [28] provide an algebraic framework based on Category theory where architectures are represented as graphs of CommUnity programs and superpositions. The allowed ways to apply connectors to components is restricted by an architectural style, given as a type graph. Dynamic reconfiguration is specified by graph transformation rules over architecture instances. Both, styles and rules are used for modeling domain-specific restrictions rather than the underlying platform as we do. Consequently, they do not deal with refinement relationships between different levels of platform abstraction.

In his Ph.D. thesis [16], Hirsch uses hypergraphs to represent architectures and hyperedge replacement grammars to define the valid architectures of an architectural style. Furthermore, he uses graph transformation rules to specify runtime interactions among components, reconfiguration, and mobility. Hypergraphs and rules are textually represented using the concept of syntactic judgements which enables formal type checking proofs. Similar to the other approaches, refinement relationships are not discussed.

The use of graph transformation techniques to capture dynamic semantics of models has also been inspired by work proposed by Engels et al. in [13] under the name of *dynamic meta modeling*. That approach extends meta-models defining the abstract syntax of a modeling language like UML by graph transformation rules for describing changes to object graphs representing the states of a model.

There are different notions of *software refinement*. For instance, Batory et. al. [6] consider *feature refinement* which is modifying models, code, and other artifacts in order to integrate additional features. For every new artifact type, they require a special refinement definition in order to compose software by generators. In our case, we concentrate on the *refinement of architectural models* and derive platform-specific models from abstract ones without adding any extra-functionality.

Such refinement of architectures has first been discussed by Moriconi et al. in [21]. Building on a formalization in first-order logic, the authors describe a general approach of rule-based refinement replacing a structural pattern in the more abstract style by its realization in the concrete style. The approach is complementary to ours because it focuses on refinement of structure rather than behavior and does not capture reconfiguration. The general idea of rule-based refine-

ment, however, could be applicable in our context, too.

Garlan [14] stresses the fact that it is more powerful to have rules operating on styles rather than on style instances. He formalizes refinements as abstraction functions from the concrete to the abstract style. We use a similar approach to define the refinement relations (see Sect. 6). Also, he argues that no single definition of refinement can be provided, but that one should state what properties are preserved. In our case, we concentrate on the preservation of the dynamic semantics of reconfiguration and communication scenarios.

Other proposals on architecture refinement like [1, 8, 10] concentrate on structural refinements only, which is complementary to our work. The only formal approach we are aware of that considers refinement of dynamic reconfiguration can be found in [7]. But, the paper provides only a sketch of the ideas without any concrete definition. Moreover, the approach is targeted on the translation from one ADL to another rather than on the refinement between architectural styles that represent different levels of platform abstraction.

### 3. Example

This paper uses a simple room reservation system as running example. The system involves three different participants: *clients* submitting inquiries about accommodations, a *travel agency* serving such requests by contacting hotels, and *hotel systems* managing the available rooms of a hotel.

System requirements include certain scenarios of component interaction and reconfiguration. For instance, after a client’s request, the travel agency has to find appropriate hotels at the client’s destination and to connect to their reservation systems. Then, the business-related interactions can take place before the connection is removed again.

If we move to design, such requirements lead to a platform-independent architecture, which can be informally depicted by UML communication diagrams as shown in Fig. 1. Reconfigurations are indicated by constraints attached to affected elements, e.g., **{new}** for connections to be created and **{transient}** for connections to be created and removed again later. Interactions are depicted as ordered and directed messages along the connections.

Let’s assume that the development team considers different platforms to implement the travel application on and eventually selects a *service-oriented architecture (SOA)* platform, e.g., Web Services. In SOA, service providers like the hotels and the travel agency expose their software functionality as *services* over a network to their clients. In order to enable dynamic service

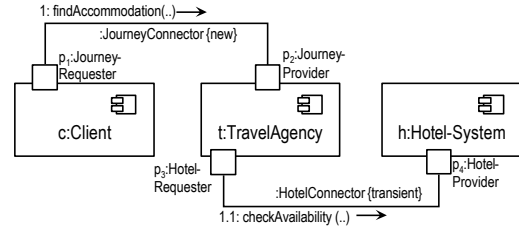


Figure 1. Room reservation system and scenario

discovery, each provider publishes a *service description* via third-party *discovery agencies* which deliver them in response to queries from service requesters. As soon as the requester retrieves a description that meets its requirements, it can start to interact with the service.

In order to integrate SOA-specific features like service discovery into the system design, the domain-specific architecture has to be refined into a SOA-specific architecture. This does not only involve structural refinements like introducing discovery services and service descriptions but also behavioral refinement of the reconfiguration scenarios. For instance, the creation of new connections might require service discovery operations beforehand.

In the following sections, we show how our approach applies to this sample refinement problem. We explain the use of architectural styles as conceptual platform models and exemplify our notion of behavioral refinement for two sample styles: a generic one for the platform-independent architecture and a SOA style for the platform-specific architecture.

### 4. Architectural styles

In this section, we revisit our approach to use architectural styles as conceptual platform models [5]. We formalize the styles as *typed graph transformation systems*, which helps to solve reachability problems (Sect. 5) and to automate and prove correct refinements (Sect. 6).

Informally, a graph transformation system consists of (1) a *type graph* to define the architectural elements, (2) a set of *constraints* to further restrict the valid models, and (3) a set of *graph transformation rules*. A system architecture that conforms to a given style is represented as an instance graph of the type graph. The transformation rules represent both communication and reconfiguration mechanisms of the considered platform. For this reason, communication-related information is structurally included in our graphs, e.g., by special nodes modeling messages with edges to their

sender and receiver components. Then, a certain re-configuration and communication scenario is modeled as a sequence of transformation rules which is applied to the initial instance graph.

Since a complete architectural model has to comprise both application-specific component *types* as well as the run-time configuration of their *instances*, a type graph contains (meta-)types for both component types and instances (similar to the meta-model of UML). This allows the reconfiguration rules of the style to operate on both levels at the same time, which is necessary for, e.g., adding a new type or checking type compatibility before creating new instances.

#### 4.1. A generic architectural style

At first, we define a generic architectural style which can be used to model platform-independent architectures (for example, the first architectural representation of our reservation system). It distinguishes between components and connectors as first-class architectural elements. Components have ports that are typed by provided and required interfaces. Communication through connections is based on message exchange, and creation and removal of connections are the only supported reconfiguration operations.

The structural elements are defined in a *type graph* as depicted in Fig. 2. According to [9], for a fixed type graph  $TG$  each valid *instance graph*  $G \in \mathbf{Graph}_{TG}$  is equipped with a structure-preserving mapping to the type graph formally expressed as a *graph homomorphism*  $tp_G : G \rightarrow TG$ .

As usual in object-oriented modeling, we use *class diagrams* to represent type graphs and *object diagrams* to represent instance graphs. Example instance graphs for the depicted type graph are presented in Fig. 4.

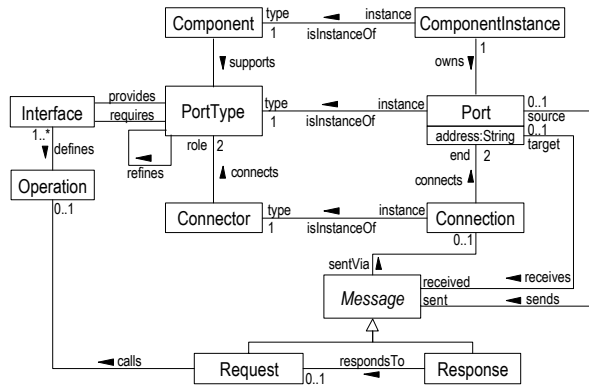


Figure 2. Type graph of the generic style

Along with the type graph comes a set  $C$  of *constraints* that further restrict the set of valid instance graphs. Simple constraints, already included in the class diagrams, are cardinalities that restrict the multiplicity of links between the elements. Omitted cardinality means 0..n by default. More complex restrictions can be defined by OCL constraints. Interested readers can refer to [4] for the full specification.

Available communication and reconfiguration mechanisms of a style are defined by *graph transformation rules*. As an example, consider the reconfiguration rule *connect* depicted in Fig. 3. According to the left-hand side, the rule can be applied if the component instances are not yet connected (*negative application condition*) and if they own ports whose types can be connected by a connector. According to the right-hand side, the result of the rule application is the creation of a new connection between the two ports.

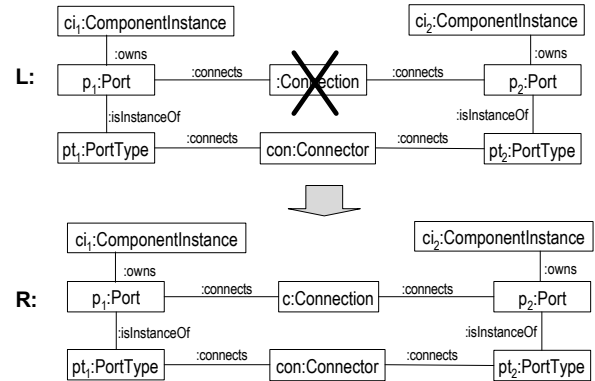


Figure 3. Reconfiguration rule connect

Formally, a graph transformation rule  $r : L \Rightarrow R$  consists of a pair of  $TG$ -typed instance graphs  $L, R$  such that the intersection  $L \cap R$  is well-defined (this means that, e.g., edges which appear in both  $L$  and  $R$  are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.). The left-hand side  $L$  represents the pre-conditions of the rule while the right-hand side  $R$  describes the post-conditions.

The application of a graph transformation rule is performed in three steps. First, we find an occurrence  $o_L$  of the left-hand side  $L$  in the current object graph  $G$ . Second, we remove all the vertices and edges from  $G$  which are matched by  $L \setminus R$ . We must also be sure that the remaining structure  $D := G \setminus o_L(L \setminus R)$  is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. In this case, the *dangling condition* [12] is violated and

the application of the rule is prohibited. Third, we glue  $D$  with a copy of  $R \setminus L$  to obtain the derived graph  $H$ . We assume that all newly created objects, links, and attributes get fresh identities, so that  $G \cap H$  is well-defined and equal to the intermediate graph  $D$ .

Figure 4 demonstrates the application of the rule `connect` creating a connection between the client and the travel agency component of our room reservation scenario. The source configuration  $G$  consists of two component instances named  $c$  of type `client` and  $t$  of type `travelAgency`. Also, a connector is available to connect the corresponding ports. Thus, we can apply the rule `connect` known from Fig. 3, which leads to the target configuration  $H$ . The complete set of about 8 rules can be found in [4].

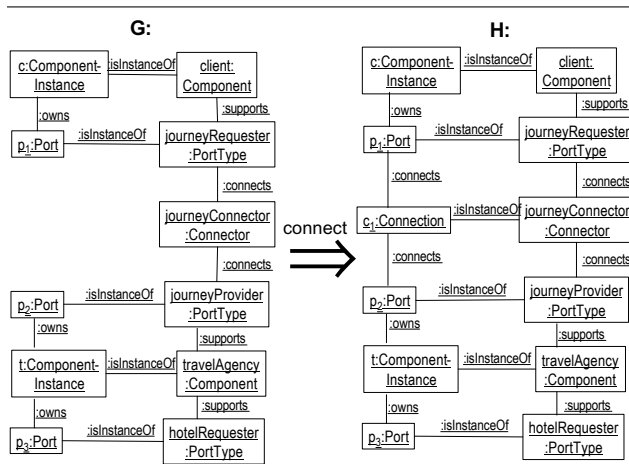


Figure 4. Application of the rule `connect`

Putting the pieces together, a *typed graph transformation system*  $\mathcal{G} = \langle TG, C, R \rangle$  consists of a type graph  $TG$ , a set of structural constraints  $C$  over  $TG$ , and a set  $R$  of rules  $r : L \Rightarrow R$  over  $TG$ .

A transformation sequence  $s = (G_0 \xrightarrow{r_1(o_1)} \dots \xrightarrow{r_n(o_n)} G_n)$  in  $\mathcal{G}$ , briefly  $G_0 \Rightarrow_{\mathcal{G}}^* G_n$ , is a sequence of consecutive transformations using the rules of  $\mathcal{G}$  such that all graphs  $G_0, \dots, G_n$  satisfy the constraints  $C$ . As above, we assume that fresh names are given to newly created elements, i.e., ones that have not been used before in the transformation sequence. In this case, for any  $i < j \leq n$  the intersection  $G_i \cap G_j$  is well-defined and represents that part of the structure which has been preserved in the transformation from  $G_i$  to  $G_j$ .

## 4.2. A SOA-specific architectural style

In this section, we define a style specific to service-oriented architectures as introduced in Sect. 3. Figure 5

shows a part of the type graph of the SOA style. It contains similar elements to those of the generic type graph but adds SOA-specific entities like `Service`, `DiscoveryService`, `ServiceDescription`, and port types used for SOA operations like publication and retrieval of service descriptions. For the complete type graph the interested reader is referred to [4].

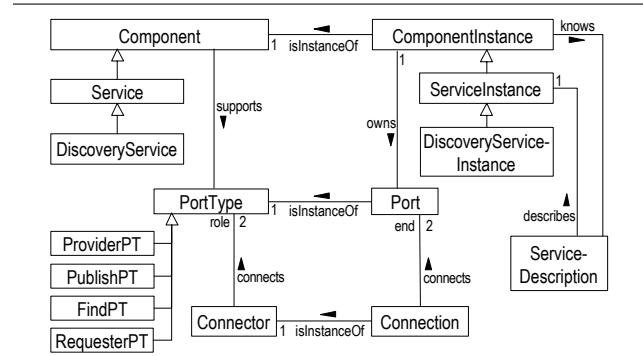


Figure 5. SOA types for structural elements

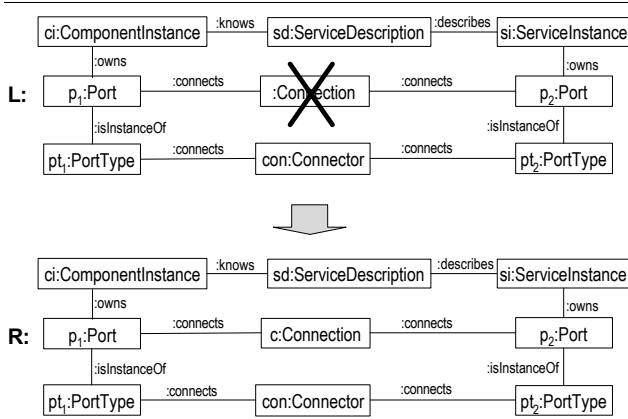
The set of transformation rules specific to SOA contains some new reconfiguration rules like for publishing and querying service descriptions, and some of the generic rules are adapted to SOA-specific platforms.

As an example, consider the SOA variant of the rule for creating connections, which is shown in Fig. 6. In comparison to its equivalent from the generic style (see Fig. 3), the rule contains a stronger precondition: Before a service requester component can be connected to the desired service, it has to know a `ServiceDescription`. Thus, other actions might be required to establish the `knows` relationship to the `ServiceDescription` before the rule can be applied. These actions refer to other rule applications and include the publication of the description to a discovery service and a lookup of the description by the service requester afterwards.

Altogether, the SOA style contains about 15 transformation rules which also cover publication and query of service descriptions, communication with a connected service, and disconnection of services (cf. [4]).

## 5. Analysis of reachability properties

The operational semantics allows us to execute our architectural specifications (simulation) and to analyze them against various kinds of properties (verification) as already discussed in a previous paper [5]. In this paper, we concentrate on *reachability properties* that are one specific class of properties.



**Figure 6. SOA-rule: connect to service**

A reachability property holds for a given graph transformation system  $\mathcal{G} = \langle TG, C, R \rangle$  and start graph  $G_0$  if an instance graph that contains a certain target pattern is reachable by applying available transformation rules. This means that an architecture can evolve from the given start configuration to the desired target configuration by performing style-specific operations.

Formally, a reachability property is expressed by a rule  $r : S \Rightarrow T$  from a *source pattern*  $S$  to a *target pattern*  $T$ . As an example, consider the statement: "Any two services of the start configuration may eventually be connected". The source pattern  $S$  of this reachability property contains place holders for the two services with their associated ports. In the target pattern we have, in addition, the required connection.

In order to verify this property for all pairs of services in the initial graph  $G_0$ , we have to consider all occurrences of  $S$  in  $G_0$  expressed via a homomorphism  $o_S : S \rightarrow G_0$ . For every such occurrence  $o_S$ , we have to find a transformation sequence from  $G_0$  to a graph  $G_n$  which contains a connection between the two services originally identified by  $o_S$ .

Formally, the reachability property  $r$  is *valid* in  $G_0$  if for each occurrence  $o_S : S \rightarrow G_0$  of  $S$  there exists a transformation sequence  $G_0 \Rightarrow_{\mathcal{G}}^* G_n$  that realizes all the effects required by  $r$ . This is the case if there exists an occurrence  $o : S \cup T \rightarrow G_0 \cup G_n$  such that

- $o|_S = o_S$ , i.e., the restriction of  $o$  to  $S$  yields the occurrence  $o_S$  of the source pattern  $S$  in  $G_0$ ,
- $o(T) \subseteq G_n$ , i.e., the target pattern is embedded into the last state of the sequence, and
- $o(S \setminus T) \subseteq G_0 \setminus G_n$  and  $o(T \setminus S) \subseteq G_n \setminus G_0$ , i.e., at least that part of  $G_0$  is deleted which is matched by elements of  $S$  not belonging to  $T$  and, symmetrically, at least that part of  $G_n$  is newly added which is matched by elements new in  $T$ .

Although reachability properties are quantified over all possible occurrences of the source pattern  $S$  in the initial graph  $G_0$ , we sometimes want to check the reachability for a specific, predefined instantiation of  $S$  only, e.g., in order to verify that "Service A may eventually be connected to service B". In this case, the graph homomorphism  $o_S : S \rightarrow G_0$  is already fixed. Since this is a subproblem of the general one, the general solution also covers this case.

The analysis of reachability properties helps to decide if a required business-related scenario, e.g., the reservation process of Fig. 1, is realizable on a certain platform. In this case, we check whether the different stages of the scenario are reachable from the initial configuration. Usually we are not merely interested in a yes-or-no answer, but we also want to know by which sequence of platform-specific actions the scenario can be realized. We call the problem to retrieve such sequence of rule applications from the start to the end configuration a *reachability problem*.

Note that, according to the definitions above, the transformation sequence from  $G_0$  to  $G_n$  may have arbitrary "side effects", besides the generation of the desired connection. Since we have to avoid such irrelevant changes when solving reachability problems for architectural refinement (see Sect. 6.2), we employ the concept of *open types* [15], i.e., a specified subset of the types of the type graph, such that only instances of these types can have side effects in the above sense. For all instance of the other (*closed*) types, such side effects are forbidden.

For every chosen or predefined occurrence of the source pattern  $S$ , the search problem of reaching  $G_n$  from  $G_0$  can be dealt with by both model checking and simulation. If it is desired, such analysis can be repeated for several or all instantiations of  $S$  in  $G_0$ .

## 5.1. Model checking

Model checking graph transformation systems has already been investigated by Varró in [27]. There, the main idea is to derive a transition system from a given graph transformation system and initial graph. The states of the transition system represent all graphs that are *reachable* from the initial graph by the transformation rules. Thus, this approach is inherently suitable to analyze reachability problems.

In order to optimize the efficiency of the analysis, Varró proposes a sophisticated encoding of graph transformation rules [27] and a prototype tool (Check-VML) [23] to automate the translation into Promela, the language of the SPIN model checker. The desired target pattern of a reachability property has to be en-

coded as an LTL formula that is satisfied as soon as a suitable state is reached.

A drawback of the approach is that model checking has to be a priori restricted to a finite state space. Therefore, one has to fix an upper bound for the number of nodes that can be created by the transformation rules. If the analysis is not successful, one can increase the bound and try again (within certain limits).

## 5.2. Simulation

We can also use classical graph transformation tools to solve reachability problems by simulating the transformations. The PROGRES [24] tool is especially suitable since it supports depth-first search and backtracking. In PROGRES, we can define a type graph, the set of transformation rules, and the given start graph. A so-called test graph models the target pattern, e.g., the two services with a connection in between.

The interpreter simulates the execution of the transformation rules by non-deterministically choosing applicable rules. If the system runs into a dead end, backtracking is used to roll back the current state. As soon as an occurrence of the test graph is found in the current host graph, the search successfully terminates.

Since the tool performs depth-first search in an infinite state space, it might run into an infinite path. For this reason, one has to program cuts in PROGRES that interrupt the backtracking at certain points and guarantee termination by limiting the search depth.

## 6. Refinement of dynamic architectures

Our notion of refinement is *style-based*, i.e., it is based on a reusable refinement relation between an abstract architectural style  $\mathcal{G} = \langle TG, C, R \rangle$  and a concrete style  $\mathcal{G}' = \langle TG', C', R' \rangle$ , where the latter has more fine-grained elements and transformation rules.

The refinement relation is based on a type mapping  $t : TG' \rightarrow TG$ , formally a *partial* surjective graph homomorphism, which maps elements of the concrete type graph  $TG'$  to the elements of the abstract type graph  $TG$  (partially shown in Fig. 7 for the two sample styles).

The definition of  $t$  is driven by semantic correspondences between the elements of the two styles. If the concrete type graph  $TG'$  is an extension of the abstract one, it should contain equivalent elements for all elements of  $TG$ . In our case,  $TG$  is even a subgraph of  $TG'$ . Thus the abstraction mapping easily becomes surjective; e.g.,  $t$  maps **Component** to **Component**.

Style-specific variants of abstract elements are defined as subtypes in  $TG'$ . The type mapping  $t$  maps

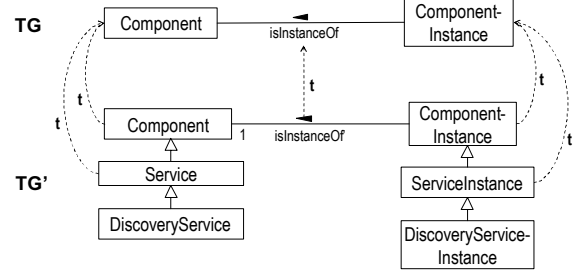


Figure 7. Part of the type graph mapping  $t$

these subtypes to the images of their supertypes in the abstract type graph. Consider, for instance, the mapping of the subtypes **Service** and **ServiceInstance** in Fig. 7. Other entity types of the concrete style, which represent entirely new, platform-specific concepts and thus do not have any equivalent in the abstract style are not mapped to  $TG$ , e.g., **DiscoveryService**.

Based on the type mapping, we now define correctness criteria for the refinement of instance graphs and transformation sequences.

### 6.1. Refinement of instance graphs

From the type mapping  $t$ , we can derive an abstraction function  $abs_t : \mathbf{Graph}_{TG'} \rightarrow \mathbf{Graph}_{TG}$  that abstracts instance graphs typed over  $TG'$  to those typed over  $TG$ . In our example, this abstraction informally consists of (1) deleting all objects (with adjacent edges) and links which, due to the partiality of  $t$ , have a type in  $TG'$  but not in  $TG$ , (2) renaming the types of the remaining elements according to  $t$ , and (3) extracting the maximal subgraph that satisfies all constraints  $C$ .

Figure 8 illustrates this by a small instance graph of the **TravelAgency** service in the SOA style. First the **ProviderPT** and **ServiceDescription** nodes are deleted including adjacent edges, because they have no mapping to  $TG$  under  $t$ . Next, the **Service** and **ServiceInstance** nodes are retyped as **Component** and **ComponentInstance**, according to the type mapping  $t$ . The intermediate result after step (2) is not consistent with the cardinality constraints of the abstract style because  $TG$  demands exactly one associated **PortType** for each **Port** (see Fig. 2). Since this is violated for node  $p_2$ , we have to delete further nodes in step (3), which formally extracts the maximal subgraph that satisfies the constraints in  $C$ .

A concrete instance graph  $G'$  is called a *refinement* of an abstract graph  $G$ , if its abstraction into the abstract style reflects exactly the elements of the abstract graph, i.e., if  $abs_t(G') = G$ . As an example, consider the instance graph in the upper left of Fig. 8. Obvi-

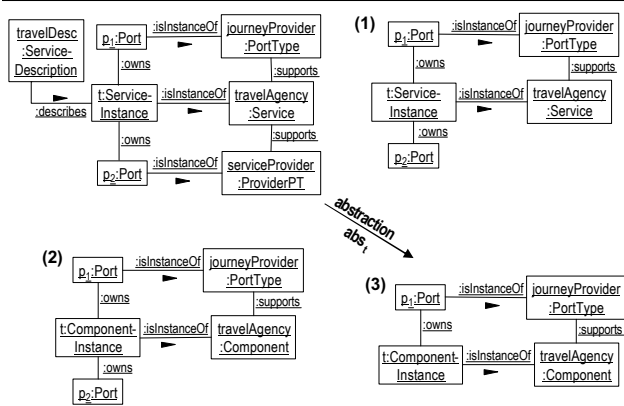


Figure 8. Abstraction of an instance graph

ously, this is a correct refinement of the abstract graph in the lower right of the figure. Thus, the abstraction function provides a criterion for the correctness of refinements.

Though this criterion helps to *check* for correct refinements, we still need a technique to *construct* the refined graphs. For this problem, we refer to existing work on structural refinements such as [1, 21]. Since our focus is more on behavioral refinement, we here assume that we can use a technique for deriving concrete graphs from abstract graphs which conforms to the above correctness criterion, e.g., by rewriting rules.

We could then apply this technique to refine the configurations that occur in our room reservation scenario of Fig. 1 by, e.g., declaring components as services and adding a discovery service. Nevertheless, plain structural refinement is not sufficient to refine the behavioral aspects of the scenario.

## 6.2. Refinement of transformations

As described in Sect. 4.1, a reconfiguration scenario is represented as a transformation sequence in the architectural style. For this reason, we extend the correctness criterion for the refinement of instance graphs to the refinement of transformation *steps* and further on to the refinement of transformation *sequences*.

For a transformation step  $s = (G \Rightarrow H)$  in the abstract transformation system  $\mathcal{G}$ , the transformation sequence  $s' = (G' \Rightarrow_{\mathcal{G}'}^* H')$  in the concrete transformation system  $\mathcal{G}'$  is a correct refinement, if  $G'$  refines  $G$  and  $H'$  refines  $H$  ( $abs_t(G') = G \wedge abs_t(H') = H$ ).

The refinement  $s'$  is a transformation *sequence* rather than a single *step* because, at the platform-specific level, it might be necessary to perform a set of consecutive steps to realize the abstract step.

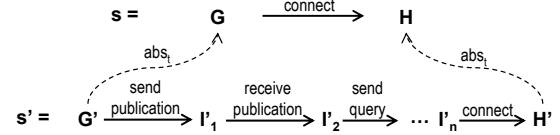


Figure 9. Refinement of a transformation step

As an example, Fig. 9 represents the refinement of the abstract transformation step  $s$  of Fig. 4, which creates a connection between client and travel agency component. At the SOA-specific level, where the travel agency is a *Service*, it would not be possible to apply only the corresponding connect rule (Fig. 6) because its precondition requires an additional *knows*-dependency to the *ServiceDescription*. To satisfy this precondition, other SOA-specific rules have to be applied first in order to publish and find the service description.

The correctness criterion for transformation steps is easily extended to sequences  $s = (G_0 \Rightarrow_{\mathcal{G}}^* G_n)$  of length greater than one: A sequence  $s' = (G'_0 \Rightarrow_{\mathcal{G}'}^* G'_n)$  over the concrete style is a valid refinement of such abstract sequence  $s$ , if  $s'$  can be partitioned into consecutive *sub-sequences* that are valid refinements of the individual transformation *steps* of  $s$ .

To actually construct such refined transformation sequence, we stick to the stepwise view and decompose the abstract sequence  $s$  into its individual steps  $s_i = (G_i \Rightarrow G_{i+1})$ . Each step is then transformed into a reachability problem as known from Sect. 5 and solved with the proposed tools as follows:

As a starting point for the first step  $s_0 = (G_0 \Rightarrow G_1)$ , we assume that we already have a correctly refined start graph  $G'_0$  and an occurrence  $o_0 : G_0 \rightarrow G'_0$ . Then, we consider  $s_0$  as a reachability property in  $\mathcal{G}'$  with source pattern  $G_0$  and target pattern  $G_1$ .<sup>1</sup> For the given start graph  $G'_0$  and the fixed occurrence  $o_0$ , we can use, e.g., a model checker to find a sequence of transformation rules leading to a graph at the concrete level which embeds the target pattern  $G_1$ .

As explained in Sect. 5, we use the concept of open types to avoid undesired side effects. In this case, we declare only those types as open that are specific to the concrete style and not mapped to abstract concepts, i.e.,  $TG' \setminus dom(t)$  such as *DiscoveryService*. Consequently, side effects are forbidden on domain-relevant types, such as *Component* and *Service* (cf. Fig. 7).

From this we can conclude that, after a successful search, the target pattern  $G_1$  has been found without any undesired side effects on business-relevant elements. Together with the refinement criterion for

1  $G_0$  and  $G_1$  are typed over both  $TG$  and  $TG'$ , as  $TG \subset TG'$ .

the start graph ( $abs_t(G'_0) = G_0$ ), we can prove that  $abs_t(G'_1) = G_1$  holds, too. Thus, the transformation sequence returned by the model checker is a valid refinement of the abstract transformation step  $s_0$ .

For all subsequent steps  $s_i = (G_i \Rightarrow G_{i+1})$ , we take the results of the previous step, i.e., the refined graph  $G'_i$  together with the occurrence  $o_i|_{G_i}$  of  $G_i$  in  $G'_i$ , in order to formulate a new reachability property and to compute a path to the desired target graph  $G'_{i+1}$ . If we repeat this procedure for all steps of the abstract transformation sequence and, if successful, concatenate all resulting transformation sequences at the concrete level, we receive a complete refinement of the abstract reconfiguration scenario.

For our example, the resulting SOA-specific transformation sequence of the reservation scenario can be summarized in a UML communication diagram as shown in Fig. 10. In addition to the interactions and reconfigurations already contained in the abstract scenario of Fig. 1, it contains SOA-specific communication with the DiscoveryEngine and corresponding reconfiguration of connections. For instance, the behavioral refinement inserted service publication (1) and query (3) operations before the actual travel service is called (4).

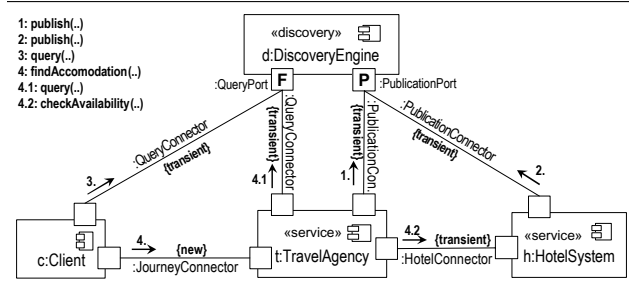


Figure 10. SOA-specific reservation scenario

### 6.3. Generalization of the approach

To illustrate the approach, we have used relatively simple mappings and closely related type graphs in the example above. Nevertheless, the approach also works with more complex mappings and with the abstract type graph not directly contained in the concrete one.

In our example, the concrete type graph simply extends the abstract one. This allowed us to directly consider an abstract transformation step as a reachability property at the concrete level. In the general case, we cannot assume that the concrete style subsumes the vocabulary of the abstract style, but we have to *translate* every abstract step into the vocabulary of the concrete style before we can apply reachability analysis.

To enable such translation, we require another mapping  $ref_t : \mathbf{Graph}_{TG} \rightarrow \mathbf{Graph}_{TG'}$ , which translates an abstract instance graph  $G$  into a reachability pattern at the concrete level by changing the typing from  $TG$  to  $TG'$ .

Since the reachability problem is then solved for  $ref_t(G)$  as target pattern, we require a relationship between the two mappings  $ref_t$  and  $abs_t$  that allows us to conclude the correctness of a refinement from the solution of the modified reachability problem. Formally, this relationship is expressed by the *satisfaction condition*, which states that the refinement of an abstract graph  $G$  can be embedded into a concrete graph  $G'$  ( $G' \models ref_t(G)$ ), if and only if  $G$  can be embedded into the abstraction of the concrete graph ( $abs_t(G') \models G$ ):

$$G' \models ref_t(G) \iff abs_t(G') \models G$$

Thus, we can abstract from the concrete definition of  $abs_t$  and  $ref_t$  as long as they preserve union, intersection, and subgraphs and fulfill the above satisfaction condition. If this is the case, one can prove *independently of the concrete mappings* that the solution of the reachability problem actually yields a correct refinement of the abstract transformation step. Thus, we believe that our approach to behavioral refinement also works with other mappings like, e.g., proposed in [21].

## 7. Conclusions and future work

In this paper, we introduced a formal technique to check and construct refinements of dynamic architectures. We used graph transformation systems to model architectural styles for different levels of platform abstraction and represented reconfiguration scenarios as graph transformation sequences. Style-based abstraction and refinement mappings were introduced to automatically refine reconfiguration scenarios while preserving semantical correctness.

The approach requires two different kinds of human intervention. People who are proficient in both graph transformation and architectural styles can design the graph transformation systems with type graph, constraints, and rules that mimic the platform-specific reconfiguration mechanisms. We need specific rules for each architectural style, but several architectures – based on the same style – can exploit the same set of rules. The *style architect* also defines a mapping between the type graph and parts of the UML meta-model (possibly extended by style-specific stereotypes) which can be used to convert UML diagrams into the graph representations. As soon as rules and UML mapping are defined, *application architects* can model their

architectures using conventional UML diagrams (suitably stereotyped for the chosen style) and validate and refine them by means of our approach.

The main goal of our future work is the development of an integrated CASE environment for the analysis and stepwise refinement of software architectures. We are proficiently conducting experiments with existing graph transformation tools and model checkers in isolation, but the final objective is a toolset that seamlessly integrates the different components. The main problem so far is the need for different formats to feed the different tools. We do not envisage significant problems to deploy a fully automated framework (maybe using some tricks to lighten the use of the model checker) with suitable backward translations of analysis results onto user models.

Needless to say, our future work also includes the application of our approach on significant and real case studies.

## References

- [1] M. Abi-Antoun and N. Medvidovic. Enabling the refinement of a software architecture into a design. In *Proc. UML 99 - The Unified Modeling Language*, volume 1723 of *LNCS*, pages 17–31. Springer, 1999.
- [2] G. D. Abowd, R. Allen, and D. Garlan. Using style to understand descriptions of software architectures. *ACM Software Engineering Notes*, 18(5):9–20, 1993.
- [3] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.
- [4] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Specification of generic and SOA-specific style. [www.upb.de/cs/ag-engels/ag\\_eng1/People/Thoene/MRDSA](http://www.upb.de/cs/ag-engels/ag_eng1/People/Thoene/MRDSA).
- [5] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and validation of service-oriented architectures: Application vs. style. In *Proc. ESEC/FSE 03 European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 68–77. ACM Press, 2003.
- [6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *Proc. ICSE 2003 - Int. Conference on Software Engineering*, pages 187–197. IEEE, 2003.
- [7] T. Bolusset and F. Oquendo. Formal refinement of software architectures based on rewriting logic. In *Proc. RCS 02 Int. Workshop on Refinement of Critical Systems*, 2002. [www-lsr.imag.fr/zb2002/](http://www-lsr.imag.fr/zb2002/).
- [8] C. Canal, E. Pimentel, and J. M. Troya. Specification and refinement of dynamic software architectures. In *Proc. WICSA1, First Working IFIP Conference on Software Architecture*, volume 140 of *IFIP Conference Proceedings*, pages 107–126. Kluwer, 1999.
- [9] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.
- [10] M. Denford, T. O’Neill, and J. Leaney. Architecture-based design of computer based systems. In *Proc. StraW03, Int. Workshop From Software Requirements to Architectures*, 2003. [se.uwaterloo.ca/~straw03/](http://se.uwaterloo.ca/~straw03/).
- [11] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
- [12] H. Ehrig, M. Pfender, and H. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
- [13] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
- [14] D. Garlan. Style-based refinement for software architecture. In *Proc. ISAW-2, 2nd Int. Software Architecture Workshop on SIGSOFT ’96*, pages 72–75. ACM Press, 1996.
- [15] R. Heckel, G. Engels, H. Ehrig, and G. Taentzer. A view-based approach to system modelling based on open graph transformation systems. In [11].
- [16] D. Hirsch. *Graph transformation models for software architecture styles*. PhD thesis, Departamento de Computación, Universidad de Buenos Aires, 2003.
- [17] D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. CONCUR 2001 - Concurrency Theory*, volume 2154 of *LNCS*, pages 121–136. Springer, 2001.
- [18] D. Le Métayer. Software architecture styles as graph grammars. In *Proc. 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23. ACM Press, 1996.
- [19] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
- [20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proc. ESEC 95 - 5th European Software Engineering Conference*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.
- [21] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
- [22] Object Management Group. Model-Driven Architecture. [www.omg.org/mda/](http://www.omg.org/mda/).
- [23] Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 92–95, 2003.
- [24] A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In [11].
- [25] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
- [26] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *Proc. TAGT’98 - Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 179–193. Springer, 2000.
- [27] D. Varró. Towards symbolic analysis of visual modeling languages. In *Proc. GT-VMT 2002 - Int. Workshop on Graph Transformation and Visual Modeling Techniques*, volume 72 of *ENTCS*, pages 57–70. Elsevier, 2002.
- [28] M. Wermelinger and J. L. Fiadero. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.