Behavior-Preserving Refinement Relations between Dynamic Software Architectures

Reiko Heckel[1] and Sebastian Thöne[2]

[1] Department of Computer Science
[2] International Graduate School Dynamic Intelligent Systems
University of Paderborn, Germany
`reiko|seb@upb.de`

Abstract. In this paper, we address the refinement of abstract architectural models into more platform-specific representations. For each level of abstraction, we employ an architectural style covering structural restrictions on component configurations as well as supported communication and reconfiguration operations. Architectural styles are formalized as graph transformation systems with graph transformation rules defining the available operations. Architectural models are given as graphs to which one can directly apply the transformation rules in order to simulate operations and their effects.
In addition to previous work, we include process descriptions into our architectural models in order to control the communication and reconfiguration behavior of the components. The execution semantics of these processes is also covered by graph transformation systems.
We propose a notion of refinement which requires the preservation of both structure and behavior at the lower level of abstraction. Based on formal refinement relationships between abstract and platform-specific styles, we can use model checking techniques to verify that abstract scenarios can also be realized in the platform-specific architecture.

1 Introduction

In the development of complex software systems, a model of the *software architecture* [30] allows for early reasoning on the system at a high level of abstraction. An architectural model covers the involved run-time *configuration* of system components, the *communication* between these components, and possible *reconfiguration* operations that enable the system to react to upcoming requirements and events. Such *dynamic* architectures gain increasing attention in the context of e-business, self-healing, and mobile systems.

Since software architectures[3] are intended to bridge the gap between system requirements and implementation, they have to conform to both business-driven requirements as well as restrictions and mechanisms imposed by the chosen run-time infrastructure. In order to integrate both aspects, we propose a stepwise refinement approach starting with an abstract, *business-level* architecture which

[3] We use the term *software architecture* as a synonym for the *model* of an architecture.

can be derived from user and business requirements. This business-level architecture is then refined into a more concrete description which also integrates *platform-specific* aspects like supported reconfiguration operations and communication mechanisms.

A recent example of this general principle of model refinement is the *Model-Driven Architecture (MDA)* [26] 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. At each refinement level, more assumptions on the resources, constraints, and services of the chosen platform are incorporated into the model.

Similarly, as described in a previous paper [3], we use *architectural styles* [2], formalized as graph transformation systems, for defining the assumptions on a certain level of platform abstraction, i.e., the vocabulary, structural constraints, and available communication and reconfiguration mechanisms. Then, an architecture at a certain level of abstraction has to conform to the corresponding architectural style.

In our previous work, we applied the available style-specific reconfiguration and communication operations to an architecture without further control. In this paper (see Section 4), we provide an extension which allows the definition of *processes* and their operational semantics. These processes control the order in which available operations are invoked by the individual software components. This leads to a more detailed picture of the architectural behavior.

We do not consider architectural refinement as the internal decomposition of components into subcomponents, as done by other authors, but rather focus on porting an abstract architecture to a more platform-specific level which usually requires additional platform-related entities and resources. For this purpose, we define different architectural styles for different levels of platform abstraction, namely a generic, platform-independent style for business-level architectures and a more specific style for architectural models at the platform-specific level.

When refining software architectures from the abstract to the concrete level, we have to preserve both structural and behavioral properties. This leads to the following two requirements:

1. **Architectural consistency:** After being ported to the lower level of abstraction, the concrete architecture has to satisfy the same functional requirements as the abstract architecture. Therefore, we have to refine configurations of components, connections, and other resources in a way that all business-relevant entities of the abstract architecture are also preserved at the concrete level.
2. **Behavior preservation:** Similarly, the concrete architecture has to preserve the abstract communication and reconfiguration behavior. In particular, we require that all business-relevant scenarios of the abstract architecture are also realizable in the concrete architecture.

While porting the abstract behavior to the platform-specific level we have to respect the capabilities of the chosen target platform according to its reconfiguration and communication mechanisms. In many cases, depending on the current situation where an operation is to be applied and the effects of preceding actions, the refinement of an abstract action varies from other situations and cannot be decided locally. Thus, we believe that behavior preservation cannot be solved by a fixed syntactic mapping between abstract and concrete operations but has to be dealt with at a semantic level.

Further requirements include a high degree of *reusability* which means that the refinement relationship between certain levels of platform abstraction should not only apply for one specific system, but should be reusable for other architectures as well.

Since refinement is not an easy task and thus error-prone and cost-intensive if done by hand, we are also aiming at *tool support*. However, it is difficult to automate the *construction* of refined architectures, because this is a creative process, and computers cannot invent details for the concrete level that are not existent at the abstract level. Nevertheless, we intend to investigate tool support for *checking* if a concrete architecture satisfies the formal refinement relationship we prescribe for the refinement from abstract to concrete level. Combined with user interaction for modifying invalid concrete models, we achieve a semi-automated approach for creating refined architectural models.

The refinement relationship, as already proposed in [5], is *style-based* meaning that it is defined between two architectural styles rather than between individual architectures. Since this relationship can be applied to any instances of the styles, we achieve the desired degree of reusability.

To check for architectural consistency, we have to compare the business-relevant entities of the abstract and the concrete model. For this purpose, we use an *abstraction function* which lifts concrete models to the abstract style. To check for behavior preservation, we have to prove that all states of an abstract scenario are also reachable in a corresponding concrete scenario, preferably with the help of model checking techniques. For this purpose, we employ a contravariant *translation function* which transforms abstract states into requirements for states at the platform-specific level. A model checker can then search for concrete states satisfying these requirements.

The rest of this paper is organized as follows. We survey related work in Section 2. In Section 3, we revisit the modeling of architectural styles based on graph theory, and in Section 4 we extend the proposed architecture description technique by processes for controlling architectural behavior. In Section 5, we use this formal framework to define our notion of refinement under the obligation of architectural consistency, and Section 6 covers the problem of behavior preservation by a semantic requirement that can be checked by model checking tools. Section 7 concludes the paper.

2 Related work

Refinement is a long-known design principle in software engineering. First ideas in the context of program development go back to Wirth [34]. In the sense of a systematic top-down methodology, he argued for the expansion of high-level program instructions to lower level macros and procedures.

While Wirth mainly investigated sequential programs, the refinement of concurrent systems became popular as *action refinement* in the context of process algebras (cf. [17] for a survey on this topic). This field considers the refinement of abstract actions into sequences of concrete actions, also called *processes*, and the potential interleaving of multiple concurrent processes.

Our approach is different from this work for two reasons. First, we want to avoid a fixed, sometimes even syntactically defined substitution of an abstract action by a concrete process wherever the abstract action occurs. Instead, we are aiming at a more flexible notion of refinement which also allows for alternate refinements of an action depending on the context where the action occurs. Second, we also want to enable refinement in those cases where the two levels of abstraction are so different that it becomes hard to relate the corresponding actions with each other.

Apart from action refinement, we also have to mention the different notions of refinement in the field of software architecture. For instance, Batory et. al. [6] consider *feature refinement* which is modifying models, code, and other artifacts in order to integrate additional features with every refinement step. Different to this work, Canal et. al. [9] consider refinement as the decomposition of a software component into subcomponents and the specialization of components under certain compatibility conditions.

In our case, we neither want to add any extra-functionality to the architecture nor to look into the internals of the components, but we rather want to port a business-level architecture to a more platform-specific level considering all the restrictions and mechanisms of the chosen target platform.

Refinement of architectures in this sense has first been discussed by Moriconi et al. in [25]. Building a formalization in first-order logic, the authors describe a general approach of rule-based refinement replacing a structural pattern in an abstract style by its realization in the concrete style. The approach is related to ours, but focuses on refinement of the structure only and does not take reconfiguration and communication behavior into account. Also, applying the logic-based theory to concrete architecture description languages is not trivial. The general idea of rule-based refinement, however, is applicable in our context, too.

Garlan [16] stresses the fact that it is more powerful to have rules operating on architectural 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 refinement relationships (see Section 5). 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 architectural consistency and the dynamic semantics of reconfiguration and communication scenarios.

Other proposals on architecture refinement like [1, 12] 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 [8]. But, the paper only sketches the ideas without any concrete definition. Moreover, the approach is targeted on the translation from one Architecture Description Language to another rather than on the refinement between architectural styles that represent different levels of platform abstraction.

Since we use graph transformation systems as the underlying formalism to describe dynamic software architectures, which is in the tradition of [21, 22, 24, 31, 33], it is also worth to look at existing work on refinement of graph transformation systems. The general idea is to relate the transformation rules and, thus, the behavior of an abstract graph transformation system to the rules of a more concrete transformation system. One can judge these refinement relationships along a continuum from syntactical relationships to more semantical ones.

Große-Rhode et. al. [18], for instance, propose a refinement relationship between abstract and concrete rules that can be checked syntactically. One of the conditions requires that, e.g., the abstract rule and its refinement must have the same pre- and post-conditions except for retyping. Based on this very restrictive definition they can prove that the application of the concrete rule expression yields the same behavior as the corresponding abstract rule. The draw-back of this approach is that it cannot handle those cases where the refining rule expression should have additional effects on platform-specific elements that do not occur in the abstract rule. And, similar to action refinement, the approach does not allow alternate refinements for the same abstract rule.

Similarly, the work by Heckel et. al. [20] is based on a syntactical relationship between two graph transformation systems. Although this approach is less restrictive as it allows additional (platform-specific) elements at the concrete level, it is still difficult to apply if there are no direct correspondences between abstract and concrete rules. Moreover, their objective is to project any given concrete transformation behavior to the abstract level and not vice versa.

In our work, we propose a more flexible, semantic-based notion of refinement. We do not define a fixed mapping between the various transformation rules but only between the structural parts of the graph transformation system. Then, we check whether all system states of an abstract model are also reachable at the concrete level, no matter by which order of transformation rules. By avoiding the functional refinement mapping between transformation rules, we can also relate transformation systems with completely different behavior, and we are flexible enough to cope with alternate refinements.

3 Graph transformation systems as architectural styles

As already introduced in [3], we use *architectural styles* as conceptual platform models. Such a platform model has to define the vocabulary of elements to be considered, to restrict the possible relationships among those elements, and to specify communication as well as reconfiguration mechanisms supported by the platform. We use different styles for different levels of platform abstraction.

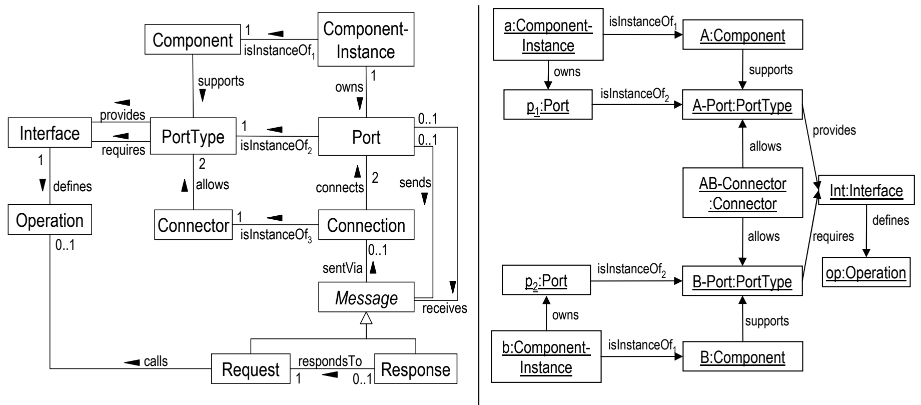
In this section, we present the formal definition of architectural styles as *typed graph transformation systems* [10] together with two exemplary styles, namely an abstract style for business-level architectures and a platform-specific style for service-oriented architectures. In Section 5, we explain how a refinement relationship between these styles can be used to refine business-level architectures, which abstract from platform-specific vocabulary and restrictions, to service-oriented architectures.

Informally, a typed graph transformation system consists of (1) a *type graph* to define the vocabulary of architectural elements, (2) a set of *constraints* to further restrict the valid models, and (3) a set of *graph transformation rules* for communication and reconfiguration operations. A system architecture that conforms to a given style is represented as an *instance graph* of the type graph.

Definition 1 (Graph and Graph Morphism). *A graph is a tuple* $G = (N, E, src, tar)$ *with a set* N *of nodes, a set* E *of edges, and functions* $src, tar : E \to N$ *that assign source and target nodes to each edge. A graph morphism* $f = (f_N, f_E) : G \to G'$ *is a pair of functions* $f_N : N \to N'$ *and* $f_E : E \to E'$ *preserving source and target (*$src' \circ f_E = f_N \circ src$ *and* $tar' \circ f_E = f_N \circ tar$*).*

Definition 2 (Typed Graph). *Given a graph* TG*, a* TG*-typed graph* $\langle G, tp_G \rangle$ *is a graph* G *equipped with a structure-preserving graph morphism* $tp_G : G \to TG$*. We call* TG type graph *and* $\langle G, tp_G \rangle$ instance graph over TG*. The category of* TG*-typed instance graphs is called* $\mathbf{Graph}_{TG}$.

The graphs we use are directed and unlabeled; for the sake of clarity, nodes (and edges) can be named by unique identifiers. Type graphs can be represented by *UML class diagrams* and instance graphs by *UML object diagrams* [19]. The typing morphism tp_G is depicted by referencing the type names. As an example, Figure 1(a) shows the type graph of the business-level style we have defined in [4]. Figure 1(b) shows a corresponding instance graph.



(a) type graph (b) instance graph

Fig. 1. Type graph and exemplary instance graph of the business-level style

According to this type graph, architectures consist of ComponentInstances which externalize their functionalities through Ports. They can interact with each other through a Connection between their Ports. The state of a communication is encoded by Request and Response message nodes.

Besides the elements for run-time configurations, the type graph also defines nodes for the application-specific types of these elements. For example, Component, PortType, and Connector nodes can be used to describe certain types of components, ports, or connections; PortTypes are characterized by provided and required Interfaces. This way, a corresponding instance graph incorporates both the actual configuration at a certain run-time state as well as application-specific type information about the involved entities.

For example, the instance graph in Fig. 1(b) defines a system that consists of an instance a of component A and an instance b of component B. Both component instances own a port of type A-Port and B-Port respectively, which could be connected by an instance of the AB-Connector. The A-Port provides the interface Int with the operation op, while the B-Port requires this interface.

Along with the type graph comes a set C of *constraints* that further restricts the set of valid instance graphs. Simple constraints already included in the class diagram are cardinalities that restrict the multiplicity of links between the elements (omitted cardinality means 0..n by default). More complex restrictions can be defined, e.g., using expressions of the *Object Constraint Language (OCL)*, which is part of the UML.

Graph transformation. Graph transformation rules [13] are used to define rewriting operations on graphs. Since our instance graphs represent system configurations, transformation rules nicely fit to define *reconfiguration operations* provided by the platform. If we encode communication-related information into the graphs, as done by the Message node and its subtypes in Fig. 1(a), then transformation rules are also suitable to represent *communication mechanisms*. A certain reconfiguration and communication scenario can be modeled as a sequence of transformation rules which are applied to an initial instance graph. The set of meaningful sequences can be restricted by additional *control processes* as discussed in Section 4.

Formally, a graph transformation rule $r : L \rightsquigarrow R$ 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 left-hand side can also state negative pre-conditions (*negative application conditions, NAG*).

According to the *Double-Pushout* semantics (DPO [14]), the application of a rule r is performed in three steps, yielding a transformation step $G \Rightarrow H$:

1. Find an occurrence o_L of the left-hand side L in the current object graph G. Formally, this is a total graph morphism $o_L : L \to G$ which maps the left-hand side L to a matching subgraph in G. The occurrence is only valid, if $o_L(L)$ cannot be extended by the forbidden elements of a NAG.

2. 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* [14] is violated and the application of the rule is prohibited.
3. Glue D with a copy of $R \setminus L$ to obtain the derived graph H. We assume that all newly created nodes and edges get fresh identities, so that $G \cap H$ is well-defined and equal to the intermediate graph D.

As an example, consider the reconfiguration rule connect depicted in Fig. 2. According to the left-hand side, the rule can be applied if there are two component instances with free ports whose types can be connected by a connector. According to the right-hand side, an application of this rule, e.g., to the graph in Fig. 1(b), results in the creation of a new connection between the two ports.

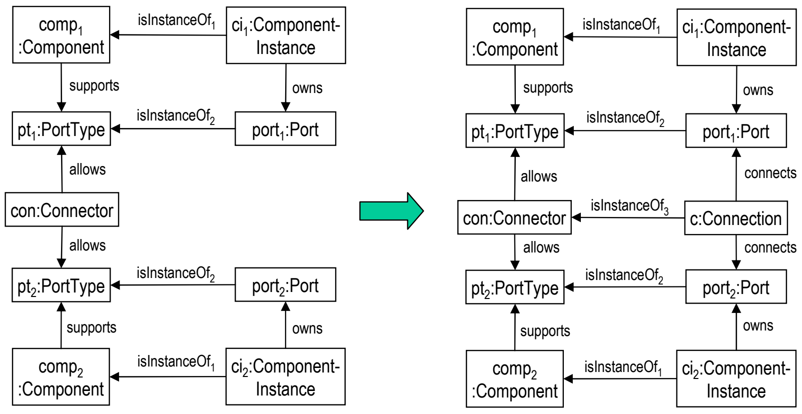


Fig. 2. Reconfiguration rule connect

Definition 3 (Typed graph transformation system). *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 transformation rules $r : L \rightsquigarrow R$ over TG.*

A transformation sequence $s = (G_0 \overset{r_1(o_1)}{\Longrightarrow}_{\mathcal{G}} \cdots \overset{r_n(o_n)}{\Longrightarrow}_{\mathcal{G}} G_n)$ in $\mathcal{G}$, briefly $G_0 \overset{*}{\Longrightarrow}_{\mathcal{G}} G_n$, is a sequence of consecutive transformations such that all graphs $G_0, \ldots, G_n$ satisfy the constraints C. As above, we assume that fresh identifiers 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.

Besides the rule connect, the graph transformation system for the business-level style contains about 10 transformation rules which handle, for instance, creation and deletion of ports and connections as well as sending and receiving of messages. The complete specification of the style can be found in [4], where we also define a platform-specific style for *service-oriented architectures* as follows.

A SOA-specific architectural style. In service-oriented architectures (SOA), software components expose their functionality as *services* over a network to service requesters. The objective of SOA is to enable *dynamic service discovery* at runtime, even if service providers and requesters do not know each other in advance. For this purpose, the service provider has to deliver a detailed description of the service with all necessary information about its interface, access point, quality-of-service, and so forth. The service description is usually published to third-party *discovery agencies* where service requesters can retrieve it from. As soon as the requester finds a description that fits the requirements, it can use it to connect to the component that provides the desired service.

For the definition of the SOA-specific architectural style, we extend the type graph of the business-level style as partially depicted in Fig. 3. The new subtypes of Component can be used to define a software component as Service or, if functioning as discovery agency, as DiscoveryService. There are also special PortTypes used for communication to discovery services. A central SOA element is the ServiceDescription which describes a specific ServiceInstance. The knows relationship indicates which components have access to the description. Besides ordinary Request and Response messages, there are additional SOA message types for service discovery, namely ServicePublication, ServiceQuery, and QueryResult.

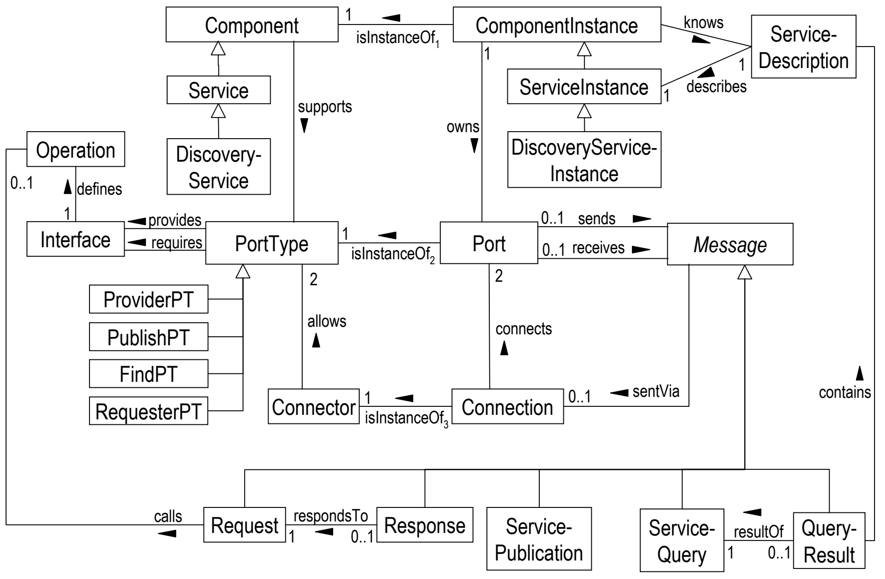


Fig. 3. Type graph of the SOA-specific style

For the creation and deletion of ports and connections, the SOA-specific style contains the same transformation rules as the business-level style, except for setting up a connection to a service which requires that the requester knows the service description beforehand. The SOA-specific variant of the connect rule, which includes this requirement as an additional precondition, is depicted in Fig. 4. To model SOA-specific platform mechanisms for dynamic service discovery, the SOA style contains additional transformation rules for publishing and querying service descriptions. Altogether, there are about 20 transformation rules which can be found in [4].

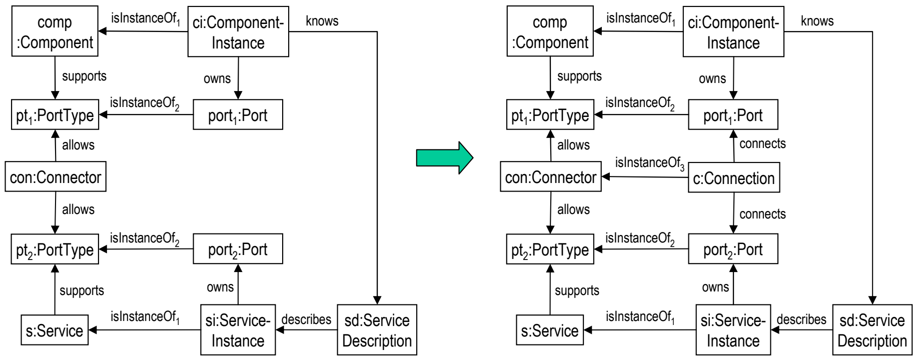


Fig. 4. SOA-specific variant of the reconfiguration rule connect

4 Processes for controlling architectural behavior

With the architectural styles presented in the previous section, we can formally define software architectures as instance graphs to which we can apply the rule-based reconfiguration and communication operations. However, such a rule-based behavior specification consisting of pre- and postconditions only is not sufficient to completely express architectural behavior.

The main problem is that a rule can be applied non-deterministically whenever and wherever its precondition is satisfied. This can lead to non-meaningful sequences of operations, for instance the deletion of a connection while the connected components are still running a certain communication protocol. Instead, we would like to be able to restrict the behavior of a system, e.g., in order to coordinate reconfiguration with communication and to specify communication protocols.

A solution to this problem should satisfy the following requirements:

1. **Process descriptions:** For each component type of a software architecture, we require the description of a *process* that restricts the order in which reconfiguration and communication operations of the underlying architectural

style can be applied to any instance of the type. Thus, an individual action of a process should correspond to the invocation of a transformation rule which conforms to the current level of abstraction. We also require operational semantics for these process descriptions which smoothly suits to the existing graph transformation framework.

2. **Parameters:** Since a process specifies the behavior of all instances of a certain component type, its actions have to relate the invocation of an operation to the local context of the respective component instance. Thus, we have to allow for *input parameters* that refer to dedicated elements within the system architecture. Similarly, we have to equip actions with *output parameters* so that the result of a rule application, e.g., a newly created port or connection, can be referred to in subsequent actions.
3. **Concurrent threads:** Since an architecture may contain several run-time instances of the same component type, we have to allow for a concurrent execution of multiple independent *threads* of the same process. Moreover, one can think of component types that follow different processes simultaneously which results in a branch of concurrent threads for each component instance. Other reasons for concurrency are situations where a server has to supply a service to multiple concurrent client requests each of which is represented by its own thread.
4. **Synchronization:** Many reconfiguration and communication operations involve more than one component instance, for example, when a connection is created between two instances. In such cases, all involved components should agree on the execution of the desired operation which gives rise to *synchronization* issues: The threads which first reach the shared action have to wait until all other participating threads have also reached that action.

The process descriptions are required for models at all levels of abstraction. As a solution, we propose an extension of the architectural styles introduced in Section 3: We integrate a relatively simple meta-model for process descriptions into the type graphs which allows us to include component-specific process descriptions into the instance graphs. Furthermore, we adapt the existing graph transformation rules so that they respect the behavior restrictions imposed by the processes. Eventually, we define a few additional transformation rules that are required for managing, i.e., starting and terminating the individual threads.

Although many authors use *process algebras* [7], petri nets, or event structures to specify and reason about concurrent processes, we stick to graph transformation theory. One reason is that we can continue to apply standard graph transformation tools for executing and analyzing the *process-controlled* transformation systems. Moreover, we can save additional efforts that would be required to combine the operational semantics of graph transformations with the different process formalisms.

Another witness for the suitability of graph transformation is existing work that uses graph transformation systems for defining the semantics of process algebras like the π-calculus. In this context, graph-based approaches are especially used for algebras that support structural changes, e.g., messages that carry references to certain communication channels; of course, such structural changes play an essential role in dynamic software architectures, too.

Type graph extensions for processes. Figure 5 shows the type graph extensions related to process specification and thread management. Note that conceptually these extensions are similar to a UML-like meta-model for process descriptions. According to the left part of Fig. 5, each Component can follow one or more Processes, and each ComponentInstance runs one or more Threads as instances of a process. Each Process has a set of *Action*s ordered by the next association. A Thread has a pointer, named previous, to the most recent action it has executed. Together with the next association this determines the possible subsequent actions. A process can declare Variables, and threads can store values for these variables as References to arbitrary model elements. For this purpose, we introduce the abstract type *Element* as a supertype to all other types in the type graph.

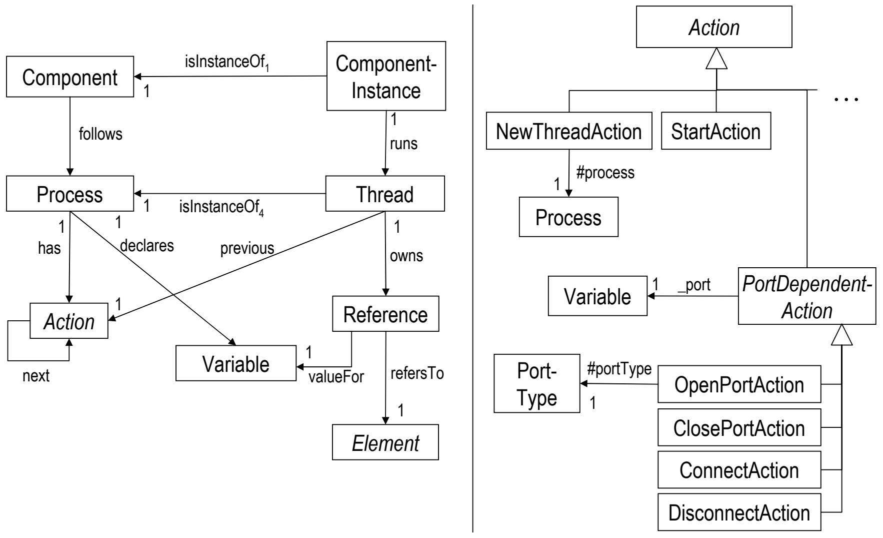


Fig. 5. Type graph extensions for processes and actions

The right part of Fig. 5 shows some of the subtypes of *Action*. Besides the NewThreadAction, which is used to create new process instances, and the Start-Action, which indicates the entry point of a process, there is a special xAction node for every transformation rule of the underlying architectural style (where x is the name of the rule). For better readability, we use additional abstract action types like *PortDependentAction* to group actions with equal parameters.

Parameters are defined by special associations outgoing from an action node. We distinguish between *constants* and *variables*. Constants, named by a preceding "#", refer to elements that define application-specific *types* like, e.g., Component, Connector, PortType, Operation, or Interface which are already known at design-time of the process. Variables, named by a preceding "_", refer to a Variable node where elements that represent *run-time instances* like, e.g., ComponentInstance, Connection, Port, or Request can be stored at execution-time of the process.

With the help of these type graph extensions, we can now include process definitions in our instance graphs in order to specify component behavior.

Transformation rule extensions for processes. The semantics of an action in a process is that the transformation rule it refers to can only be applied at that point of the process. This introduces an additional precondition that has to be checked before a rule is applied. Note that this is only a necessary but not commensurate precondition: If the other preconditions of the rule are not satisfied, then the rule cannot be applied immediately, and the process has to wait until the remaining preconditions are satisfied, too. In order to properly interpret actions in this way, we have to adapt the existing transformation rules of our graph transformation systems and to restrict their applicability. We call such adapted transformation rules *process-controlled*.

Consider, for example, the OpenPortAction in Fig. 5 which should create a new port whose type is specified by the input parameter #portType and return this port through the output variable _port. The corresponding process-controlled graph transformation rule is depicted in Fig. 6. The upper part of the rule (with gray background) indicates the original reconfiguration rule which creates a new port of the selected port type.

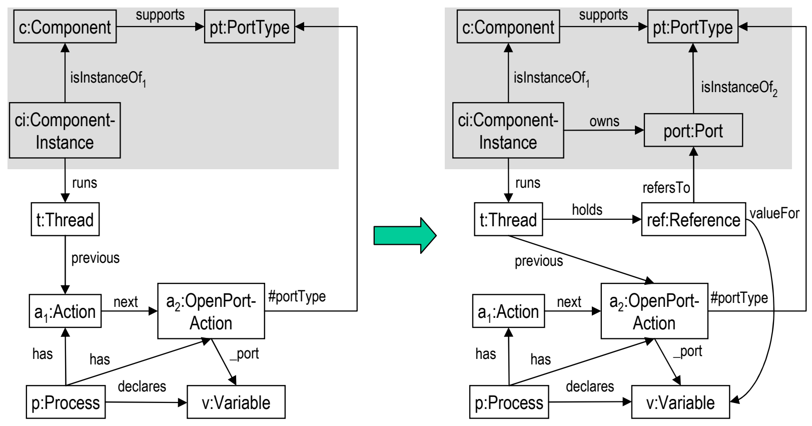


Fig. 6. The process-controlled reconfiguration rule openPort

The new, lower part of the rule restricts its application to those situations where an OpenPortAction belongs to the next actions of the running thread and where the #portType parameter selects the right PortType. According to the lower right-hand side of the rule, which defines additional process-related effects, the rule creates a Reference node for the current thread which refers to the new port node as value for the output variable _port. As another effect, all such process-controlled transformation rules update the pointer to the previous action like a program counter that is increased after completion of a command.

The concurrency requirements discussed at the beginning of this section are implicitly satisfied by the process-controlled transformation rules because the rules are non-deterministically applied wherever possible. And, since every rule application represents the execution of one of the pending actions, this corresponds to a non-deterministic interleaving of the concurrent threads. Thus, concurrency is the default behavior of graph transformation systems.

In addition, the management of threads is handled by the two special transformation rules in Fig. 7: The rule newThread presented in Fig. 7(a) creates new instances of a process whenever a NewThreadAction occurs. The input parameter #process determines which process has to be started. If the parameter refers to the action's own process, a new thread of the same process is started, e.g., if multiple threads of the process are required to serve multiple incoming requests.

A thread terminates after an action that has no more subsequent actions according to the next relation. In this case, the garbage collection rule clearThread of Fig. 7(b) can be applied in order to remove the remaining Thread and Reference nodes (crossed out elements are negative application conditions).

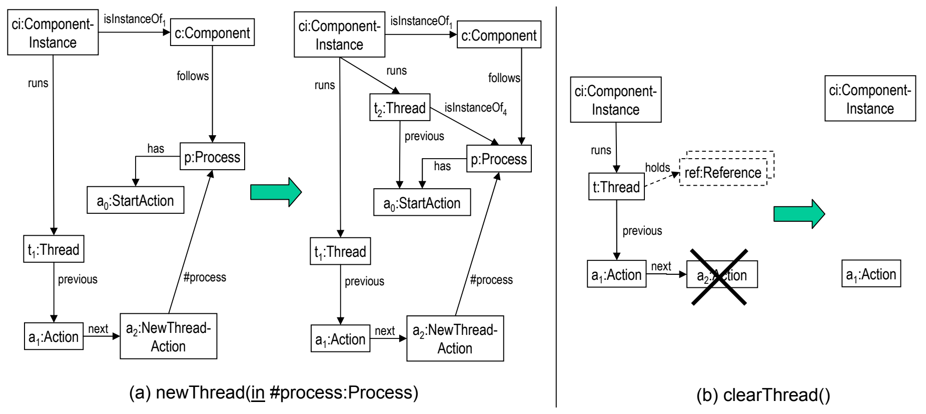


Fig. 7. Thread management rules newThread (a) and clearThread (b)

Synchronization between concurrent threads is required, if a reconfiguration or communication operation involves more than one component. Then, it becomes necessary to get agreements from all involved threads before the corresponding rule can be applied. We satisfy this requirement by *non-local rules* whose precondition comprises the current state of more than one component.

Remember, e.g., the reconfiguration rule connect depicted in Fig. 2. In this case, both components should agree to the creation of the new connection beforehand. Figure 8 shows how this synchronization requirement is integrated as a non-local precondition into the left-hand side of the rule. This way, the two involved threads synchronize at the virtually shared ConnectAction.

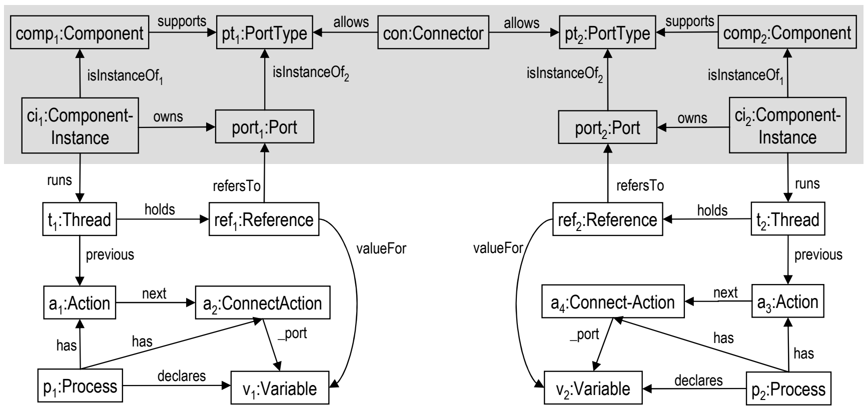


Fig. 8. The process-controlled variant of rule connect (left-hand side only!)

Although both the process-controlled transformation rules and the process-aware instance graphs become more complex, we benefit from the uniformity of the theory and its built-in operational semantics. Also, one can think of the formal graph-based model representation as an internal format for tools only, while software architects do not necessarily have to encode their architectures and process definitions as instance graphs. In fact, they can rather use a better concrete notation like *UML component diagrams* for structural aspects and *UML activity diagrams* or *statecharts* for process definitions, which is then internally translated by a suitable CASE tool. Moreover, one could also think of import and export interfaces to other process description standards like the *Business Process Execution Language for Web Services* (BPEL4WS [15]).

5 Refinement of architectural configurations

As already discussed in Section 1, refinement is an important concept for developing software architectures. This section deals with the refinement of architectural configurations while preserving the provided system functionality (architectural consistency). In an instance graph, functional elements are reflected by nodes for component and connection types and run-time instances thereof (see Section 3) as opposed to behavior-related elements for control processes (see Section 4).

We do not provide a functional refinement operator which takes an abstract architecture and returns the corresponding concrete architecture, because such a refinement is a creative, non-deterministic design process which includes several alternative options which can all lead to a valid refinement at the concrete level. However, what we do provide is a formal criterion for deciding if a given concrete architectural configuration is a valid refinement of a certain abstract configuration or not.

At first, we define this criterion based on an abstraction function. In the second part of this section, we illustrate this kind of abstraction by means of the two architectural styles from Section 3.

5.1 Abstraction-based refinement criterion

Our notion of refinement is *style-based*, i.e., it is based on a relationship between an abstract architectural style $\mathcal{G} = \langle TG, C, R \rangle$, e.g., the business-level style from Section 3, and a concrete style $\mathcal{G}' = \langle TG', C', R' \rangle$, e.g., the service-oriented style from Section 3. The refinement of architectural configurations formally corresponds to the refinement of the structural parts of an abstract instance graph $G \in \mathbf{Graph}_{TG}$ into a concrete instance graph $G' \in \mathbf{Graph}_{TG'}$.

In the previous section, we extended the type graphs by behavior-related elements in order to encode process descriptions into instance graphs. However, to reason solely about structural aspects of an architecture, we have to distinguish the structure-related part TG_S of the underlying type graph TG from the process-related extensions defined in Section 4. From this distinction, we can derive a *projection function* on instance graphs which preserves structure-related elements only and neglects all behavior-related elements.

Definition 4 (Projection). *Given a type graph TG and a subgraph $TG_S \subseteq TG$ thereof. The projection of instance graphs from TG to TG_S is defined as a function $proj_{TG_S} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG_S}$ which returns for any $G \in \mathbf{Graph}_{TG}$ with nodes N, edges E, and typing $tp : G \to TG$ a graph $G_S \in \mathbf{Graph}_{TG_S}$ with nodes $N_S = \{n \in N | tp(n) \in TG_S\}$, edges $E_S = \{e \in E | tp(e) \in TG_S\}$, and typing $tp|_{G_S}$. Note that G_S is a subgraph of G ($G_S \subseteq G$).*

According to the requirements stated in Section 1, a refinement criterion has to respect architectural consistency, meaning that for a valid refinement the concrete architecture has to preserve the functionality of the abstract architecture. Thus, all structural entities like components and connectors of the business-level model have to be preserved in the platform-specific model.

Since the two instance graphs to be compared are expressed in terms of different architectural styles, i.e., different type graphs, one cannot simply compare them and check if the abstract graph is part of the concrete graph. Before we can do so, we rather have to express one of the graphs in terms of the other architectural style.

The canonical solution to this problem is by means of an *abstraction* function $abs : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$ which takes the concrete instance graph and, by abstracting from all platform-specific details, lifts it to the abstract level. Then, we can check if the resulting abstraction contains the same functional elements as the original abstract graph. For this purpose, we regard the original abstract graph as a *property* that has to be *satisfied* by the lifted concrete graph according to the following definition:

Definition 5 (Satisfaction). *Given a model represented as an instance graph G and a property represented as an instance graph P, both typed over the same type graph TG. Also, given a type graph distinction $TG_S \subseteq TG$ with a corresponding projection $proj_{TG_S}$.*
We say that G satisfies *P, i.e., $G \models P$, iff there is a total, injective graph morphism $m : proj_{TG_S}(P) \rightarrow proj_{TG_S}(G)$. This means that the relevant part of P can be embedded into the relevant part of G.*

Based on this definition, we can now formally define structural refinement based on abstraction as follows:

Definition 6 (Structural Refinement). *Given an abstract type graph TG, a concrete type graph TG', and an abstraction function $abs : \mathbf{Graph}_{TG'} \rightarrow \mathbf{Graph}_{TG}$. A concrete instance graph $G' \in \mathbf{Graph}_{TG'}$ is a* structural refinement *of an abstract instance graph $G \in \mathbf{Graph}_{TG}$, if $abs(G') \models G$.*

5.2 Abstraction function based on a type graph morphism

The abstraction function *abs* is a semantic mapping, associating with each concrete configuration a corresponding abstract configuration. There is a range of possibilities for the concrete definition of *abs* depending on the characteristics of the respective architectural styles. For example, it is sufficient to base the abstraction function *abs* on a mapping between the abstract and the concrete type graph, if the abstraction of a concrete instance graph consists of adapting the types of business-relevant elements and omitting platform-specific elements.

Other cases, not discussed in detail in this paper, might require more complex transformations which map entire patterns of concrete elements to abstract elements. For instance, a combination of two unidirectional channels at the platform-specific level could be used to realize an abstract bidirectional channel. These complex mappings can be defined by graph transformation systems or even more sophisticated methods like *triple graph grammars* [29].

In the rest of this section, we take the platform-independent (pi) style of Section 3 as abstract transformation system $\mathcal{G}^{pi} = \langle TG^{pi}, C^{pi}, R^{pi} \rangle$ and the service-oriented (so) style as concrete transformation system $\mathcal{G}^{so} = \langle TG^{so}, C^{so}, R^{so} \rangle$. We define a mapping between the two type graphs TG^{so} and TG^{pi} and exemplify how to derive an appropriate abstraction function from this mapping.

Type graph mapping. A type graph mapping is a partial graph morphism $t : TG^{so} \rightarrow TG^{pi}$ which maps structure-related elements of the concrete type graph TG^{so} to structure-related elements of the abstract type graph TG^{pi} as partially shown in Fig. 9.

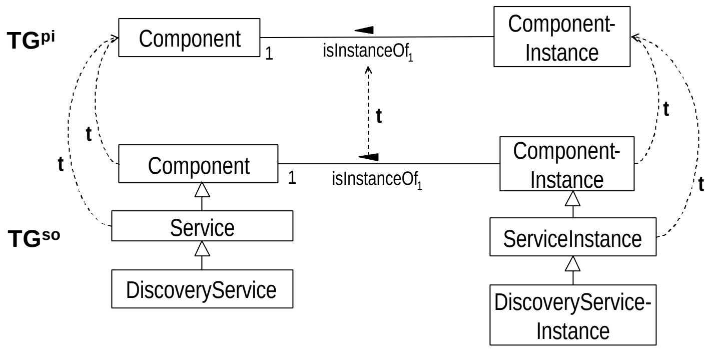


Fig. 9. Part of the type graph mapping t

The concrete definition of t is driven by semantic correspondences between the structure-related elements of the two type graphs. For instance, both Component and Service nodes in a service-oriented architecture represent what we call a Component in the business-level style. Since there is no distinction between private and published components at the abstract level, t maps both types to the abstract type Component.

Those elements that represent purely platform-specific concepts not occurring at the abstract level like, e.g., DiscoveryService, ServiceDescription, or the SOA-specific port types (see Fig. 3) are not mapped to the abstract type graph. For behavior-related elements like, e.g., Process and Action nodes (see Fig. 5), the type mapping is undefined, too, because the abstraction function only needs to lift structural aspects to the abstract level.

Abstraction function. From the type mapping t, we can now derive an abstraction function $abs_t : \mathbf{Graph}_{TG^{so}} \rightarrow \mathbf{Graph}_{TG^{pi}}$ that abstracts instance graphs typed over TG^{so} to those typed over TG^{pi}. This abstraction informally consists of (1) renaming the types of all elements whose type has an image in TG^{pi} according to the definition of t, (2) deleting all nodes and edges which, due to the partiality of t, have a type in TG^{so} but not in TG^{pi}, and (3) deleting all dangling edges and those adjacent nodes whose number of connected neighbor nodes falls below the lower bound of the relevant cardinality constraint.

Figure 10 illustrates the effect of the abstraction function abs_t for a small instance graph in the service-oriented style (shown in the upper left corner). The instance graph defines a service S that supports a port type AccessS for using the service and another port type PublishDesc for sending a service description to available discovery services. The represented run-time snapshot contains one instance si of the service which owns a port for each of the supported port types. Besides, there is a description document descr describing the service instance.

In a first step, we apply the type mapping t and rename the types of the Service and ServiceInstance nodes into Component and ComponentInstance (1).

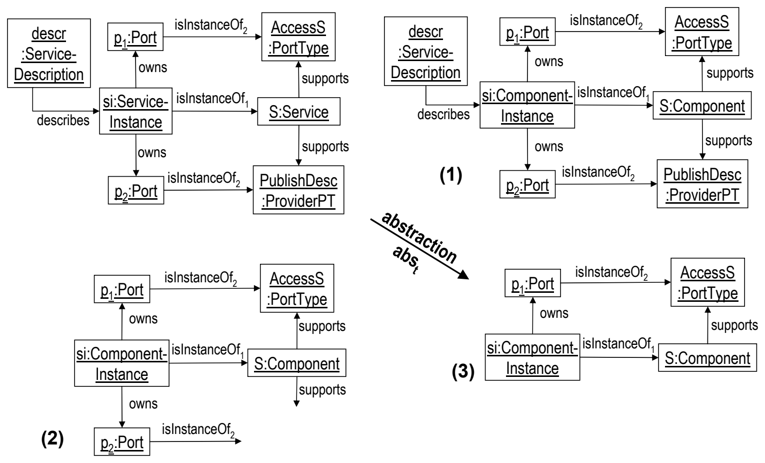


Fig. 10. Abstraction of a small instance graph

Then, we delete the ProviderPT and ServiceDescription nodes and the describes edge because they have no mapping to TG^{pi} under t (2). The deletion of the ProviderPT node leads to the deletion of the adjacent Port node in the third step, because otherwise the cardinality constraint would be violated which says that every Port requires a PortType. Eventually, all dangling edges are removed (3).

6 Behavior-preserving refinement

In addition to structural refinement, we also want to check if the refinement preserves the architectural behavior. In this section, we extend our refinement criterion by a corresponding semantic condition, discuss possibilities and requirements for an automated verification of this criterion, and exemplify the concepts with the two style examples from Section 3.

6.1 Extended criterion for behavior-preserving refinement

While a single instance graph represents a certain system state, the application of a transformation rule represents a transition from one state to a subsequent state. This way, the potential behavior can be represented as a *transition system* whose states are the reachable instance graphs and whose transitions are generated by rule applications. If we also encode application-specific process definitions into the instance graphs, as introduced in Section 4, then the architectural behavior represents the concurrent execution of these processes.

Given the initial state of the architecture as a start graph, one can generate and explore the transition system by continuously applying the transformation rules to previously generated states. We can reduce the state space by considering isomorphic graphs as a single state only. This is sensible, because for isomorphic graphs the same set of transformation rules is applicable, and the result of a rule application is only determined up to isomorphism in the DPO approach anyway (cf. [11]). Note that, however, in some cases the resulting transition system may still be infinite.

Recently, the automated generation and exploration of transition systems from graph transformation systems is supported by tools like GROOVE [27] and CheckVML [28, 32].

Definition 7 (Graph Isomorphism Class). *Two TG-typed graphs G and H are isomorphic, briefly $G \cong H$, if there is a bijective graph morphism $i : G \to H$, called isomorphism, which also preserves their typings, i.e., $tp_H \circ i = tp_G$.*

An isomorphism class $[G]$ is the set of all graphs that are isomorphic to G, i.e., $[G] = \{H \in \mathbf{Graph}_{TG} \mid G \cong H\}$.

Definition 8 (Architectural Behavior). *Given an architectural style represented by a typed graph transformation system $\mathcal{G} = \langle TG, C, R\rangle$ and an initial system state represented by an instance graph $G_0 \in \mathbf{Graph}_{TG}$, then the architectural behavior is defined by a transition system $TS_{G_0} = (S, \Rightarrow)$ with*

- *a set of states $S = \left\{[G] \mid G_0 \overset{*}{\Longrightarrow}_{\mathcal{G}} G\right\}$ consisting of isomorphism classes of all graphs reachable in $\mathcal{G}$, with $[G_0] \in S$,*
- *a transition relation $\Rightarrow \subseteq S \times S$ which is defined by all possible rule applications in $\mathcal{G}$, i.e., $G_i \Rightarrow G_j$ iff $G_i \Rightarrow_{\mathcal{G}} G_j$.*

For the sake of simplicity, we continue to denote the states of the transition system as graphs rather than as graph isomorphism classes. Thus, when speaking of $G \in S$, we precisely mean $[G] \in S$. Moreover, the concatenation of consecutive transitions starting from G and leading to H is called a *path* from G to H, denoted by $G \overset{*}{\Longrightarrow} H$.

For the intended refinement criterion, we again consider an abstract architecture as instance graph G of a platform-independent style $\mathcal{G} = \langle TG, C, R\rangle$ and a concrete architecture as instance graph G' of a platform-specific style $\mathcal{G}' = \langle TG', C', R'\rangle$. Besides structural refinement, we now also require that the concrete architecture preserves the behavior of the abstract architecture.

This property is expressed in terms of structural refinements for all states reachable in the abstract behavior. To be more precise, we demand, that for every path $G \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ in the abstract transition system TS_G there exist paths $G' \overset{*}{\Longrightarrow} G'_1 \overset{*}{\Longrightarrow} \ldots \overset{*}{\Longrightarrow} G'_n$ in the concrete transition system $TS_{G'}$ with G'_i structurally refining G_i (that is, $abs(G'_i) \models G_i$) for all $i = 1 \ldots n$.

Since we are now dealing with isomorphism classes, we require that the abstraction function *abs* preserves isomorphisms: $G \cong H \implies abs(G) \cong abs(H)$. As a consequence, it is indifferent to which representative of an isomorphism class the function is applied.

In terms of software architecture, a path represents a certain scenario of communication and reconfiguration operations, and, for a behavior-preserving refinement, we want to ensure that every abstract, business-level scenario can also be realized at the platform-specific level. This criterion can be formulated as a co-inductive definition as follows:

Definition 9 (Behavior-Preserving Refinement). *Given an abstract architectural style $\mathcal{G} = \langle TG, C, R \rangle$, a concrete architectural style $\mathcal{G}' = \langle TG', C', R' \rangle$, and an abstraction function $abs : \mathbf{Graph}_{TG'} \to \mathbf{Graph}_{TG}$ which preserves isomorphisms. A concrete instance graph $G' \in \mathbf{Graph}_{TG'}$ with behavior $TS_{G'}$* refines *an abstract instance graph $G \in \mathbf{Graph}_{TG}$ with behavior TS_G, if*

- *$abs(G') \models G$*
- *for every transition $G \Rightarrow H$ in the abstract system TS_G there exists a path $G' \overset{*}{\Longrightarrow} H'$ in the concrete system $TS_{G'}$ such that H'* refines *H.*

According to this definition, a single transformation *step* $G \Rightarrow_{\mathcal{G}} H$ is refined by a transformation *sequence* $G' \overset{*}{\Longrightarrow}_{\mathcal{G}'} H'$. This is because it might be necessary to perform a number of platform-specific steps in order to realize the abstract step. For example, consider an application of the reconfiguration rule connect (see Fig. 2). In a service-oriented architecture, it is not directly possible to apply the corresponding SOA-specific connect rule (see Fig. 4), because connecting to a service requires knowledge about its description beforehand. If the description is not known to the requester, other SOA-specific rules for service publication and discovery have to be applied first as shown in Fig. 11.

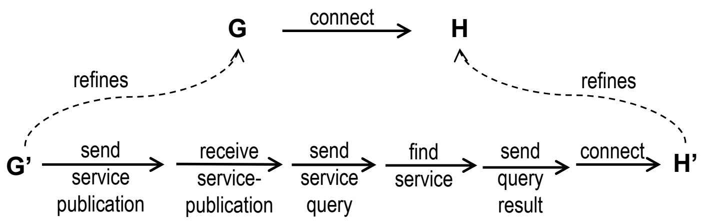


Fig. 11. Refinement of an abstract transformation step

Since the behavior-preserving refinement solely depends on the structural refinement of reachable states, we do not need to provide a fixed mapping between the transformation rules of the two involved styles. This is especially advantageous, if abstract and concrete operations are very different.

Moreover, this approach implicitly allows for alternate refinements of an abstract operation depending on the context of its application. In the above case, for instance, the service description might have been published to the discovery agency already. Then, the first two operations of the transformation sequence in Fig. 11 can be omitted. Or, the description might already be known to the requester due to some previous look-up so that even the query operations can be omitted.

6.2 Tool-based verification of behavior preservation

Based on the tools for automated state space generation mentioned above, one can apply various analysis techniques like *model checking* to the resulting transition system. We propose to express the behavioral refinement check as a *reachability problem* in the concrete transition system that can be solved by model checkers.

According to Definition 9, for a given abstract transformation step $G \Rightarrow H$ the reachability problem consists of searching a path $G' \overset{*}{\Longrightarrow} H'$ in the concrete transition system with $abs(H') \models H$. Consequently, the abstraction function abs has to be applied to every visited state in order to find an appropriate target graph H'. Since this affects the computational complexity, we would rather express the same property solely at the level of the concrete system.

For this purpose, we assume a second translation, contravariant to abstraction. A function $trans : \mathbf{Graph}_{TG} \rightarrow \mathbf{Graph}_{TG'}$ associates an abstract instance graph with a concrete one representing the reformulation of an abstract state over the concrete type system. Note that the concrete graph does not necessarily represent a complete state of the concrete architecture, but rather a minimal pattern which has to be present in order for the requirements of the abstract graph to be fulfilled. Thus, we consider a concrete instance graph G' as a valid structural refinement of an abstract graph G if it *satisfies* this pattern, formally $G' \models trans(G)$.

Since the abstraction function abs is in general not injective, there are various alternative possibilities for translating an abstract configuration to the concrete level which are all valid structural refinements. The translation function $trans$ selects for every abstract graph only one possible translation instead of returning the set of all potential translations. Thus, the definition of $trans$ already includes certain design decisions determining the specific refinement of abstract elements.

As a consequence, not every concrete configuration that is a valid refinement according to abs is a valid refinement according to $trans$, too. However, in the opposite direction we require that the translation function has to be compatible with the abstraction function so that a refinement according to $trans$ entails a refinement according to abs. This is formally expressed as a *satisfaction condition*, reminiscent of similar conditions in algebraic specification or logics.

Definition 10 (Satisfaction Condition). *Given abstract type graph TG and concrete type graph TG'. A translation function $trans : \mathbf{Graph}_{TG} \rightarrow \mathbf{Graph}_{TG'}$ is* compatible *to an abstraction function $abs : \mathbf{Graph}_{TG'} \rightarrow \mathbf{Graph}_{TG}$, if the following* satisfaction condition *holds for all $G \in \mathbf{Graph}_{TG}$ and all $G' \in \mathbf{Graph}_{TG'}$:*

$$G' \models trans(G) \Longrightarrow abs(G') \models G$$

Note that, due to the argument above, the opposite direction of the entailment is in general not true. This reflects the fact that the translation function contains more information than the abstraction function. Thus, the left condition is more specific and not necessarily entailed by the right condition. It would be different if we had defined the translation function as returning the set of all possible translations of an abstract configuration. But then, it would be more difficult to reformulate the reachability problem solely for the concrete transition system.

Under the assumption of the satisfaction condition, we can now easily derive the following theorem which allows to express behavior-preserving refinement solely at the concrete level.

Theorem 1. *Given an abstract architectural style $\mathcal{G} = \langle TG, C, R \rangle$, a concrete architectural style $\mathcal{G}' = \langle TG', C', R' \rangle$, and compatible abstraction and translation functions abs and trans. A concrete instance graph $G' \in \mathbf{Graph}_{TG'}$* refines *an abstract instance graph $G \in \mathbf{Graph}_{TG}$ according to Definition 9, if*

- *$G' \models trans(G)$*
- *for every transition $G \Rightarrow H$ in the abstract system TS_G there exists a path $G' \overset{*}{\Longrightarrow} H'$ in the concrete system $TS_{G'}$ such that H'* refines *H.*

Proof. Since *trans* is compatible to *abs*, we can conclude from $G' \models trans(G)$ that $abs(G') \models G$ holds, too. The second clause already conforms to Definition 9.

A model checker like SPIN [23] can now be used to refine an abstract transformation step $G \Rightarrow H$ by looking for a state that satisfies $trans(H)$, technically speaking a graph that contains $trans(H)$ as a subgraph. With the help of temporal logics such as *linear-time temporal logic* (LTL), we can even formulate the reachability of entire abstract sequences $G \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ as:[4]

$$\Diamond(trans(G_1) \wedge \Diamond(trans(G_2) \wedge \ldots \Diamond(trans(G_n)) \ldots))$$

Since we require only one path to satisfy the above formula while an LTL formula always refers to all paths of the transition system, we have to negate the above formula and let the model checker look for a counter example. A counter example that violates the negated formula can then be used as a witness for the original formula.

Although we cannot verify the refinement of the complete abstract transition system this way, we are able to check at least the most important scenarios of business-level behavior.

6.3 Compatible translation function for a type graph-based abstraction function

In order to satisfy the required compatibility, the definition of a translation function heavily depends on the definition of the contravariant abstraction function. In this subsection, we revisit the abstraction function from Section 5.2, which is based on a mapping between the concrete and the abstract type graph elements, and we show how a compatible translation function looks like for this kind of abstraction function.

[4] The LTL operator $\Diamond$ means "at some time in the future"

In Section 5.2, we define a semantic mapping $t : TG^{so} \to TG^{pi}$ from the concrete type graph to the abstract type graph. From this type graph morphism, we derive the abstraction function abs_t on instance graphs which consists of renaming concrete types to abstract types and, due to the partiality of t, deleting purely platform-specific elements.

For the definition of a compatible translation function, we *invert* the type mapping t. However, since t is not injective (e.g., both Component and Service are mapped to Component in Fig. 9), the resulting inverse $\bar{t}$ is a *relation* between the elements of the two type graphs, which can be expressed by a function $\bar{t} : N_{TG} \to \mathcal{P}(N_{TG'})$. If t maps a concrete node type nt' to the abstract type nt, then $nt' \in \bar{t}(nt)$. Analogously, $\bar{t}$ can be extended to edge types as well.

From the inverted type mapping $\bar{t}$, we can now derive a translation function $trans_{\bar{t}} : \mathbf{Graph}_{TG} \to \mathbf{Graph}_{TG'}$. For an instance graphs $G \in \mathbf{Graph}_{TG}$ with typing morphism tp, it

1. deletes all nodes n whose type has no image under $\bar{t}$, i.e., $\bar{t}(tp(n)) = \emptyset$
2. changes the type of n to a certain $nt' \in \bar{t}(tp(n))$ else.

The first case is relevant for behavior-related nodes, e.g., for process-descriptions, which are also excluded from the original type graph mapping t. The second case adapts the types of the remaining elements to the vocabulary of the concrete style. Since there might be several alternatives returned by $\bar{t}$ for adapting the type of an abstract element, the translation function cannot completely be defined at the type level but requires additional user decisions at the instance level for translating individual nodes.

Technically, these user decisions can be integrated by additional node attributes that are set by the engineer to determine the desired translation option for a node. By evaluating the values of these attributes for a given instance graph, the translation function determines the intended translation to the concrete level.

What remains is to show the compatibility of $trans_{\bar{t}}$ to the original abstraction function abs_t. According to Definition 10, we have to show that

$$G' \models trans_{\bar{t}}(G) \Longrightarrow abs_t(G') \models G$$

Proof sketch. For arbitrary $G \in \mathbf{Graph}_{TG}$ and $G' \in \mathbf{Graph}_{TG'}$ be

$$G' \models trans_{\bar{t}}(G) \tag{1}$$

Since $\bar{t}$ is the inverse of t, $trans_{\bar{t}}(G)$ contains only elements whose types are in the domain of t. These elements are preserved by abstraction on both sides of (1). Thus, the satisfaction relation still holds after application of the abstraction function:

$$abs_t(G') \models abs_t(trans_{\bar{t}}(G)) \tag{2}$$

Since the application of $trans_{\bar{t}} \circ abs_t$ is the identity for structure-related elements, we receive:

$$abs_t(G') \models G \tag{3}$$

□

7 Conclusion

In this paper, we introduced a formal technique for modeling dynamic architectures as instances of graph transformation systems. Graph transformation rules were used to express available communication and reconfiguration operations in a certain architectural style. Architectural models were enriched by process descriptions with operational semantics that restrict and coordinate the architectural behavior.

We have discussed semantic conditions for the behavior-preserving refinement of architectural models under the obligation of architectural consistency across different levels of platform abstraction. Style-based abstraction and translation functions were introduced as formal refinement criteria that can be checked with the help of analysis tools.

The presented approach is very flexible because it is not based on a fixed syntactical mapping between the different operations but on a semantic relationship that also respects context-dependent alternatives for refining abstract behavior.

Future work includes further investigations on the verification of behavioral refinement by existing simulation algorithms for related transition systems. We are planning to support the approach by a coupling of CASE tools with editors and analysis for graph transformation systems, presently conducting experiments with existing model checkers.

References

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

[4] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based modeling and refinement of service-oriented architectures. Technical Report TR-RI-04-250, University of Paderborn, 2004. `ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-04-250.pdf`.

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

[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] J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. Elsevier Science, 2001.

[8] 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/`.

[9] 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.

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

[11] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation; basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations. World Scientific, 1997.

[12] 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/`.

[13] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.

[14] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.

[15] T. Andrews et al. *Specification: Business Process Execution Language for Web Services Version 1.1.* BEA, IBM, Microsoft, SAP AG and Siebel Systems, 2003. `http://www-106.ibm.com/developerworks/library/ws-bpel/`.

[16] 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.

[17] R. Gorrieri and A. Rensink. Action refinement. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1047–1147. Elsevier, 2001.

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

[19] Object Management Group. The UML website. `www.uml.org`.

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

[21] D. Hirsch. *Graph transformation models for software architecture styles.* PhD thesis, Departamento de Computación, Universidad de Buenos Aires, 2003.

[22] 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.

[23] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[24] 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.

[25] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.

[26] Object Management Group. Model-Driven Architecture. `www.omg.org/mda/`.

[27] A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfalz, and B. Böhlen, editors, *Proc. Application of Graph Transforma-*

tions with Industrial Relevance (AGTIVE '03), volume 3062 of *LNCS*. Springer, 2003.

[28] 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.

[29] A. Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[31] 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.

[32] 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.

[33] M. Wermelinger and J. L. Fiadero. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.

[34] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.