Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Style-Based Modeling and Refinement of Service-Oriented Architectures*

A graph transformation-based approach

Luciano Baresi[1], Reiko Heckel[2], Sebastian Thöne[3], Dániel Varró[4]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione, Italy – e-mail: `baresi@elet.polimi.it`
[2] University of Leicester, Department of Computer Science, U. K. – e-mail: `reiko@upb.de`
[3] University of Paderborn, International Graduate School *Dynamic Intelligent Systems*, Germany – e-mail: `seb@upb.de`
[4] Budapest University of Technology and Economics, Department of Measurement and Information Systems, Hungary – e-mail: `varro@mit.bme.hu`

Received: date / Revised version: date

Abstract Service-oriented architectures (SOA) provide a flexible and dynamic platform for implementing business solutions. In this paper, we address the modeling of such architectures by refining business-oriented architectures, which abstract from technology aspects, into service-oriented ones, focusing on the ability of dynamic reconfiguration (binding to new services at run-time) typical for SOA.

The refinement is based on conceptual models of the platforms involved as architectural styles, formalized by graph transformation systems. Based on a refinement relation between abstract and platform-specific styles we investigate how to realize business-specific scenarios on the SOA platform by automatically deriving refined, SOA-specific reconfiguration scenarios.

Key words service-oriented architecture – architectural style – architecture refinement – graph transformation

1 Introduction

The service-based paradigm to structure and modularize software systems becomes more and more popular for complex distributed applications with high degree of dynamic reconfigurations and interactions among system components. This article focuses on the architectural aspect of service-based software engineering and introduces a methodology for deriving *service-oriented architectures (SOA)* from high-level business-oriented architecture descriptions.

Software architectures play an important role in software development [42]. As abstract models of the run-time structure they help to bridge 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. Thus, models of dynamic architectures do not only have to consider component structure and interactions, but also dynamic changes of that structure.

Service-oriented architectures are one import kind of dynamic architectures. They allow for automated service publication and discovery at run-time. For instance, whenever a service cannot be provided with the required quality-of-service any longer, the service requester could dynamically search for and change to a new service.

Designing such dynamic architectures is a complex task because one has to cope with both business-driven and platform-driven requirements: The business requirements prescribe certain component structures, interactions, and reconfigurations which have to conform to the interaction and reconfiguration mechanisms provided by the underlying middleware platform.

We propose to deal with this complexity by a stepwise refinement approach that covers various degrees of *platform abstraction*. At first, the software architect derives an abstract model of the architecture from the user and business requirements. This model roughly corresponds to the *conceptual architecture view* proposed in [23]. It mainly covers the functional aspects encapsulated in business-related components. Such a business-level architecture description abstracts from the concrete middleware and run-time platform of the system, and it

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

omits elements that are needed to use platform-specific communication and reconfiguration mechanisms.

For instance, a business-level architecture model describes business components and their interfaces but neglects the distinction between ordinary components and published service components. Consequently, it does not contain any SOA-specific elements like service descriptions and discovery services, either. Also, it elaborates on the various use cases of the system by business-oriented scenarios of component interactions, but it neglects SOA-specific interactions and reconfiguration operations required for service publication and discovery.

Only later in the design process, when the decision for a service-oriented middleware platform has been made, more and more non-functional requirements and SOA-specific aspects are integrated into the core functionality. This leads to a SOA-specific model of the application architecture which refines both the structural and the behavioral parts of the business-level model according to the service-oriented paradigm.

A recent example of this general modeling principle is the *Model-Driven Architecture* (MDA)[1] [30] 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.

In software architecture research, *architectural styles* are used to describe families of architectures by common resource types, configuration patterns and constraints [2]. As Di Nitto and Rosenblum argue in [32], the restrictions imposed by a certain choice of platform can be considered as an architectural style, too. Moreover, to account for component interactions and platforms that support dynamic reconfigurations like SOA, we suggest in [4] to extend the classical notion of architectural style by defining not only structural constraints but also platform-specific communication and reconfiguration mechanisms.

As described in [4], we formally define architectural styles as graph transformation systems including type graph, constraints, and transformation rules. Based on that, we investigate refinement relationships between abstract and concrete styles in [5]. They enable us to check if a given architecture is a refinement of another one with a special focus on the refinement of business-level scenarios of communication and reconfigurations into platform-specific scenarios.

Our refinement criteria guarantee both semantic correctness and platform consistency. This means that the platform-specific scenario comprises the same functional behavior as the business-level scenario, and that it is consistent with constraints and mechanisms imposed by the chosen target platform.

In this article, we apply the approach to service-based architectures and extend the SOA case study sketched in [4]. We present a complete definition of the architectural style for service-oriented architectures including all relevant mechanisms for service publication, discovery and connectivity. We also define an abstract, business-level style and a refinement relationship between the abstract and the SOA-specific style. Moreover, we show how this relationship can be used to check for correct architecture refinements and to derive SOA-specific scenarios from given business-level scenarios.

Since refinements can be tedious and error-prone, we show how the behavior refinement problem can be formulated 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 business-level architectures into SOA-specific architectures.

In order to account for user-friendly models, we also discuss how to combine the formal, graph-based representation of architectures with the *Unified Modeling Language* (UML)[2]. For this purpose, we introduce an extension of the UML meta-model, proposed as a UML *profile* for service-oriented architectures in [19], and define a mapping between the profile and the elements of the architectural style. This mapping can be used by editors and other tools to provide a user-friendly syntax for complex architectural descriptions.

The rest of the article is organized as follows. In Sect. 2, we introduce a typical service-based application as running example for this article. In Sect. 3, we explain how architectural styles can be used as conceptual platform models for different levels of platform abstraction and how they can be formalized by graph transformation systems. We define an abstract style for business-level architectures and a specific style for service-oriented architectures. In Sect. 4, we discuss the use of UML and UML profiles as concrete notation for the presented SOA models. In Sect. 5, we define refinement relationships between architectural styles and show how they can be used in order to derive SOA-specific architectural models from business-oriented models. We survey related work in Sect. 7, and Sect. 8 concludes the article.

2 The SmartCar example

Throughout this article, we use a simple scenario, taken from the automotive domain, to demonstrate the main features of our approach. The scenario foresees how the availability of special-purpose and context-sensitive services will change our way of planning trips with our car.

While driving our car, we can inquire a map service to get the best (cheapest) map of the area. Another ser-

[1] www.omg.org/mda/

[2] www.uml.org

vices computes the best itinerary to reach the final destination. The computation is not only based on static information, like the cheapest route (i.e., no fees) or the fastest one (i.e., always on highways), but also uses actual traffic conditions to better plan the itinerary.

The first step towards design leads to a platform-independent architecture that can be informally depicted by means of the UML component diagrams of Fig. 1.

The scenario can be enacted by means of the four special-purpose components of Fig. 1, which specify their interaction ports in terms of required and supplied interfaces. The Vehicle is the only active component which starts the scenario and acts as coordinator of any dynamic interaction. The MapRequest component provides both the best (highest resolution) or cheapest map. Since we do not aim at discussing the way services and their parameters can be negotiated, it is the service itself that supplies the best option without negotiating parameters. The ItineraryDefinition component complements bought maps with the itineraries we choose. Again, we allow for the cheapest and fastest itineraries. This component interacts with the TrafficInformation system to identify the fastest trip with respect to the actual traffic conditions.

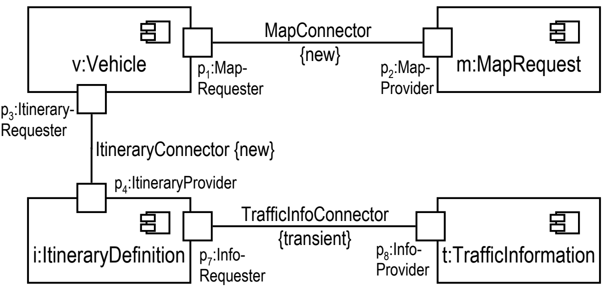


Fig. 1 Component diagram of the SmartCar system

Since the vehicle moves around a region, the actual components must be identified and used while the system is executed in a fully dynamic fashion. This means that the actual links among the components change while the scenario evolves. These changes are constrained by available reconfiguration mechanisms.

In the component diagram, those reconfigurations are indicated by attachments to affected elements: {new} for newly created connections and {transient} for connections that are created and then removed after a while.

System requirements also include certain scenarios of component interaction. For instance, as shown in Fig. 2, Vehicle (i.e., the driver) asks the MapRequest for the best map of the area. It also selects the itinerary to reach its final destination. For this purpose, it queries ItineraryDefinition, which in turn queries the TrafficInformation service to select the fastest offer.

After the platform-independent model, the next development phase requires the selection of a specific platform the system should be deployed on. Similar to the

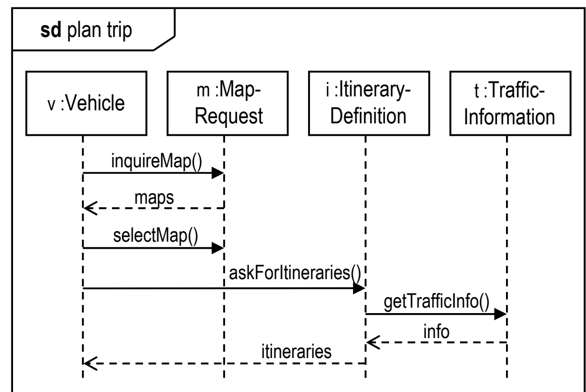


Fig. 2 Simple scenario of the SmartCar system

intentions of MDA [30], the platform-independent model is in principle portable to different platforms. In this case, the development team decides to implement the SmartCar system based on a *service-oriented architecture (SOA)*, e.g., Web Services. This means that the business components expose their functionality as *services* over a network to vehicles. A service is equipped with a description of the provided functionality including information where and how to access it.

As shown in Fig. 3, SOA involves three different roles: *service providers*, *service requesters* and *discovery agencies*. The service provider runs and exposes the service. Also, the provider has to *publish* the service description, in order to enable dynamic service discovery and to allow requesters to access the service.

Since providers and requesters usually do not know each other in advance, the service descriptions are published via third-party discovery agencies. They categorize the descriptions and deliver them in response to *queries* issued by service requesters. As soon as the service requester retrieves a service description that meets its requirements, it can use it to *interact* with the service.

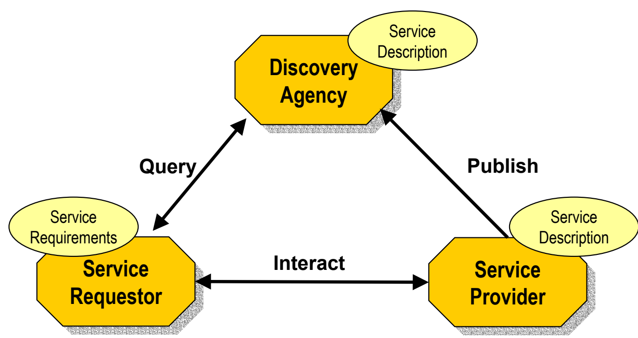


Fig. 3 Roles in a service-oriented architecture, cf. [9]

Service-oriented architectures are highly dynamic and flexible: Components and services are only loosely coupled and communicate according to standardized

protocols; service descriptions with interface specifications are exchanged at runtime; thus, service requesters can dynamically replace unsatisfactory services if other services provide a better alternative concerning functionality or quality. This might also become necessary for self-healing purposes, e.g., if a service is not reachable any longer due to some network problems.

To integrate SOA-specific features like service discovery into system design, we refine the business-level architecture 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 a new connection to a service might require service discovery operations beforehand.

In the following sections, we show how our refinement approach can be applied to this sample application. We explain the use of architectural styles as conceptual platform models and how the architectural models can be expressed in terms of these styles. Then, we exemplify our notion of behavioral refinement between a style for platform-independent architectures and a style for service-oriented architectures.

3 Architectural styles as platform models

In this section, we revisit our approach from [4] to use architectural styles as conceptual platform models which are formalized as *typed graph transformation systems* [11]. After a brief introduction to graph transformations, we present a platform-independent architectural style for business-level architectures and a platform-specific architectural style for service-oriented architectures.

As we want to model software architectures in relation to their computational infrastructure at different levels of platform abstraction, we need a conceptual model for each of these infrastructures. For such a *conceptual platform model*, we consider the following four requirements:

1. It has to define the *vocabulary* of elements that are to be considered in an architecture description for the chosen platform. In an architecture description, the engineer can then use this vocabulary to define application-specific types as well as runtime configurations of these types.
2. It has to define and constrain the *relationships* that are allowed among the various architectural elements of the vocabulary.
3. It has to define the *communication mechanisms* that are provided by the platform to let the architectural elements interact. An architect who wants to design interaction scenarios among software components can then use these communication mechanisms in the scenarios.
4. It has to define the *reconfiguration mechanisms* that are provided by that platform to let a dynamic architecture evolve at runtime and to change its current configuration. An architect who wants to design scenarios of architectural behavior can then include reconfigurations that conform to these mechanisms in the scenarios.

Since a platform model constrains the possible application architectures according to the assumptions about the underlying platform, it can also be considered as an *architectural style* that characterizes the family of architectures which conform to the platform-specific restrictions and mechanisms.

While the classical notion of architectural style covers *structural* constraints only like common vocabulary and topological patterns [2], we extend this notion by also taking into account the communication and reconfiguration mechanisms as required above. For this purpose, we represent architectural styles as *typed graph transformation systems*.

A typed graph transformation system $\mathcal{G} = \langle TG, C, R \rangle$ consists of a *type graph* TG to define the architectural elements and their relationships, a set of *constraints* C to further restrict the valid models, and a set R of *graph transformation rules*.

Nodes of the type graph define the architectural elements, i.e., the vocabulary of the architectural style. Edges define the possible links and relationships among these elements. A type graph can be depicted as a *UML class diagram* as shown in Fig. 4. We can also define subtypes, which inherit all the associations of its supertype, and attributes, which describe additional properties of the respective element.

We use the vocabulary in a concrete application architecture by representing system configurations as *instance graphs* of the type graph. According to [11], a valid instance graph $G \in \mathbf{Graph}_{TG}$ has to be equipped with a structure-preserving mapping to the fixed type graph TG, formally expressed as a *graph homomorphism* $tp_G : G \to TG$. In combination with the UML class diagram for the type graph, we use *UML object diagrams* to depict instance graphs. One can assign attribute values to the instances in an object diagram in order to add information about their current state.

Along with the type graph comes the 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, e.g., using expressions of the *Object Constraint Language* (OCL) [33], which is part of the UML. Together, the type graph and the constraints satisfy the first two requirements stated above for platform models.

The third and fourth requirement are handled by *graph transformation rules*. They represent both communication and reconfiguration mechanisms provided by

the considered platform. Examples for such rules can be found in Table 1.

The application of a transformation rule to an instance graph results in rewriting a certain part of that graph. Since, in our case, instance graphs represent system configurations, the transformation rules are well-suited to model reconfiguration mechanisms that can be applied to change the system configuration.

In order to treat communication mechanisms in the same way, we have to encode communication-related information into the instance graphs. For this reason, we add dedicated nodes to the type graph which represent, e.g., messages with edges to their sender, receiver, and current position. Then, special transformation rules can be defined to create and transmit messages.

Altogether, the consecutive applications of transformation rules to a given instance graph, also called a *transformation sequence*, can be used to model a certain scenario of both reconfiguration and communication operations.

Formalization: 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.

According to the *Double-Pushout* semantics [14], the application of a transformation rule r to a *host graph* G, yielding a direct transformation step $G \stackrel{r,o_L}{\Longrightarrow} H$, is performed in three steps:

1. Find an *occurrence* o_L of the left-hand side L in the current host graph G, formally a structure-preserving graph morphism $o_L : L \to G$.
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.

A transformation sequence $s = (G_0 \stackrel{r_1,o_1}{\Longrightarrow} \cdots \stackrel{r_n,o_n}{\Longrightarrow} G_n)$ 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 $G_0, \ldots, G_n$ 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.

After this introduction to graph transformation theory, we now illustrate the concepts by two sample graph transformation systems that represent architectural styles for business-level and service-oriented architectures. Later in the article, these styles are used to demonstrate the stepwise refinement approach for developing complex, service-oriented architectures.

3.1 A style for business-level architectures

The first architectural style we define represents a high level of platform abstraction and can be used for business-level architecture descriptions. At the business level, we do not want to consider platform-specific aspects but concentrate on core functionalities. Therefore, we avoid as many assumptions as possible about the underlying platform and assume a basic computational infrastructure for component-based, distributed systems only.

As usual in architecture descriptions [42], the style prescribes to use *components* and *connectors* as first-class entities to configure a system architecture. Components are considered as encapsulated black boxes which can communicate with their environment through dedicated *ports* only. Ports are characterized by provided and required *interfaces*. Two components can only interact with each other if their ports are connected by a connector.

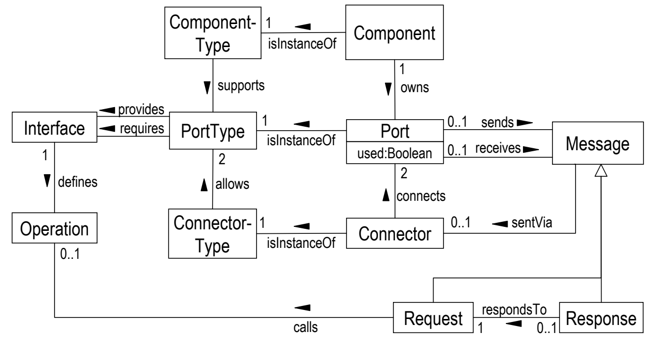


Fig. 4 Type graph of the business-level style

Type graph: The type graph of the style is shown in Fig. 4. It can be subdivided into two parts: The left half contains elements to define application-specific *types*, i.e., the ComponenTypes, the supported PortTypes (including provided and required Interfaces), and the ConnectorTypes. The right half of the diagram contains elements to define the runtime configuration of a system with Components, Ports, and Connectors, i.e., concrete instances of the aforementioned types respectively.

Consequently, instance graphs of the type graph describe both application-specific types as well as runtime configurations of concrete instances thereof. The type information allows, e. g., to determine the interfaces provided or required by a certain component. Moreover, the transformation rules presented below need to access the type information in order to check type compatibility when creating new instances. Note that the same technique of including elements for both application types and runtime instances in a single type graph is also used in the UML meta-model [34].

Attributes can be used to store additional information in a node. For example, we add the boolean attribute used to the Port node in order to distinguish between free and already used ports. The current value of the attribute can be queried, e.g., before a transformation rule is applied. For instance, the rule openPort in Tab. 1 can only be applied to open a new port for a component if there is no other free port left.

As mentioned before, we have to include special communication elements in the type graph in order to express communication mechanisms by graph transformation rules. For this reason, the type graph contains a Message node which is specialized into subtypes Request and Response.

Constraints: Besides the cardinalities given in the class diagram, there is a set of additional OCL constraints which further exclude undesired instance graphs. For example, the following expression ensures that a Connector only connects those Ports whose PortTypes are allowed by its ConnectorType:

```
context Connector inv:
  self.port.portType -> forAll(pt|
    self.connectorType.portType -> includes(pt))
```

With the help of the type graph and the constraints, we can now model system configurations as instance graphs that conform to the business-level style. Figure 5 shows an example which models the initial configuration of the SmartCar system.

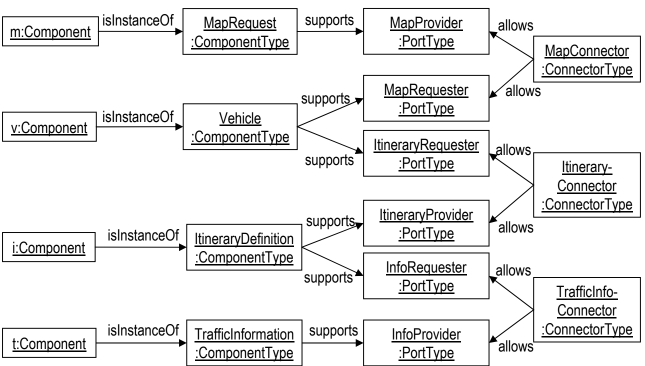


Fig. 5 Instance graph for the SmartCar application

Although such instance graphs may easily become large and unreadable, they are still suitable as formal representation inside tools. In order to facilitate the handling of larger models by end users, we recommend to apply the UML as concrete notation as discussed in Sect. 4.

Transformation rules: The architectural behavior of our models depends on the communication and reconfiguration mechanisms provided by the infrastructure or the platform. For the business-level style, we assume that ports can be opened or closed and that connectors can be created or removed, but we abstract from mechanisms for finding the right partner components and defer this question to the platform-specific style. Moreover, we assume a basic communication mechanism which is based on message exchange via established connectors.

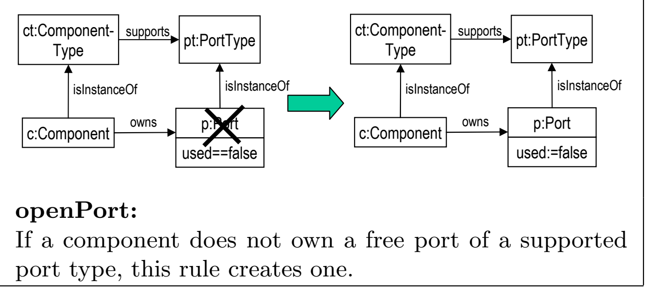
These mechanisms are defined by the graph transformation rules listed in Table 1. As an example, consider the first rule openPort which creates a new port for a component. The pre-condition on the left-hand side demands that the type of the component supports the type of the port to be created.

As we want to avoid that the rule is applied again and again to the same component creating an unbounded number of ports, we add a *negative application condition* to the rule. Such a negative condition is depicted by crossed-out elements like, e.g., the Port node p in rule openPort. It prevents the application of the rule to any occurrence of the left-hand side which can be extended by the crossed-out elements. In the case of openPort, this means that the rule is only applicable if the component does not already own a free port of the selected port type.

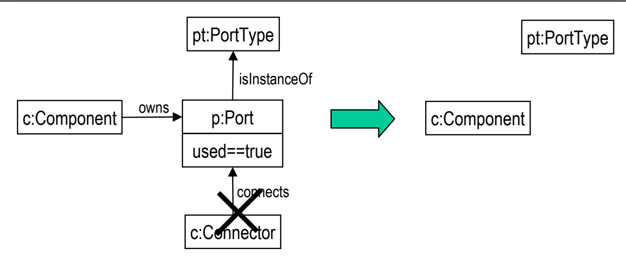
According to the right-hand side of openPort, a rule application results in the creation of a new port for the component. While on the left-hand side one can *query* attribute values, at the right-hand side one can *assign* attribute values. In this case, the value of the used attribute of the new port is initially set to false.

With these remarks, we believe that the rest of Tab. 1 should be self-explanatory. It contains further rules to create or remove connections, to send or receive requests and responses, and to remove finished messages.

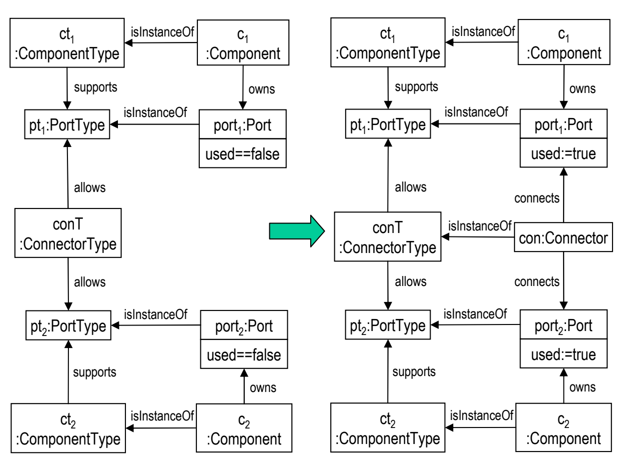
Table 1 Transformation rules of the generic style



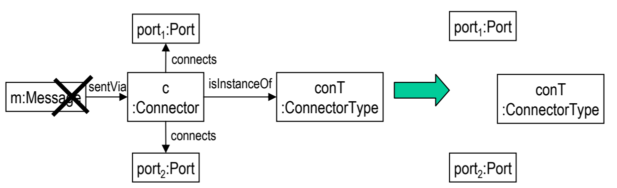
openPort:
If a component does not own a free port of a supported port type, this rule creates one.



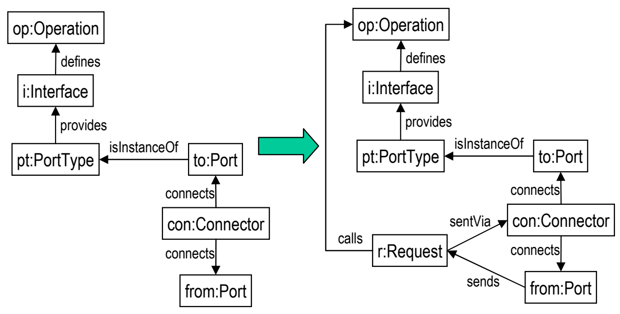
closePort:
If a used port is not any longer connected to a connector, this rule deletes the port.



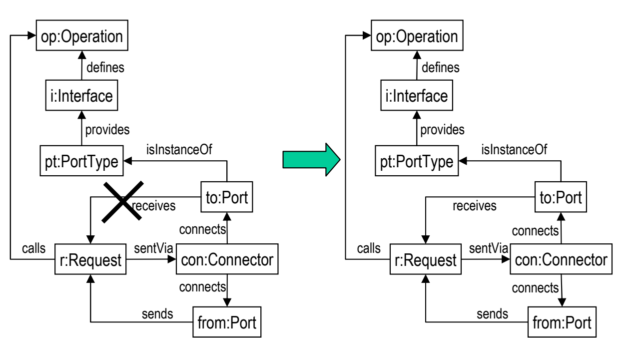
connect:
If two components own a free port each and there is a compatible connector type, this rule creates a new connector between the two ports.



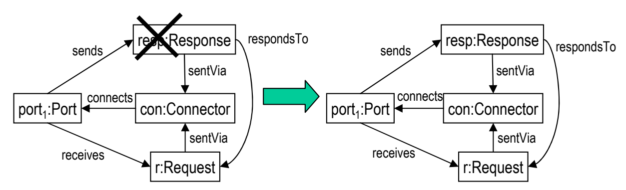
disconnect:
If a connector between two ports does currently not transport any message, it can be removed by this rule.



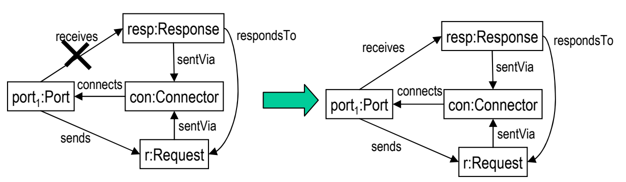
callOperation:
If a port is connected to another port that provides a certain operation, then the port can send a request calling that operation.



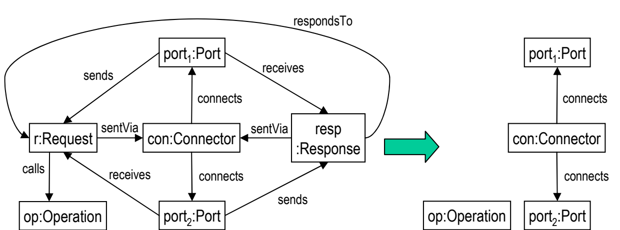
receiveCall:
The port providing an operation can receive new incoming calls from connected ports by this rule.



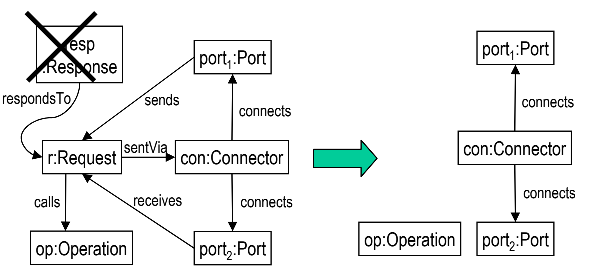
sendResponse:
If a port has received a request, this rule sends a response.



receiveResponse:
The port sending a request can receive a corresponding response by this rule.



finishRequestResponse:
This rule removes finished request and response messages.



finishRequest:
This rule removes a finished request message which did not get a response.

After having introduced the transformation rules, we can apply them to the initial configuration of the SmartCar system shown in Fig. 5. This way, we can formally model the SmartCar scenario from Sect. 2 as a transformation sequence. The beginning of the transformation sequence is partially shown in Fig. 6.

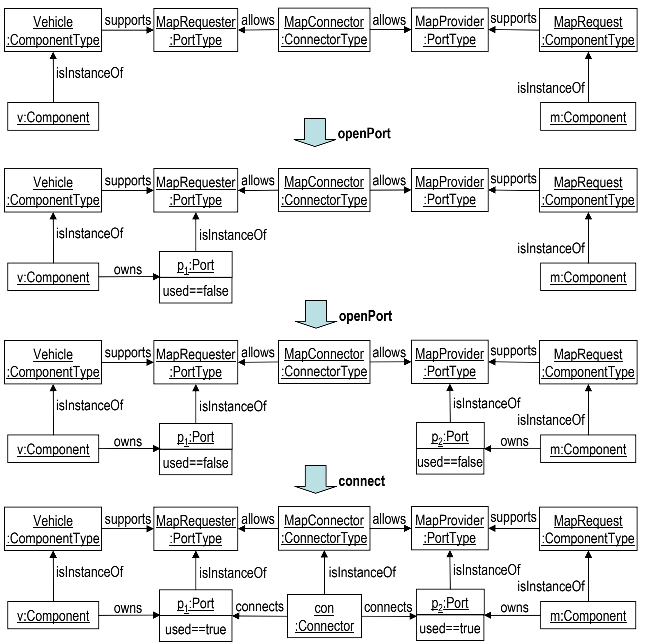


Fig. 6 Transformation sequence for the SmartCar scenario in the business style

3.2 A style for service-oriented architectures

While the above presented architectural style can be used for modeling at the platform-independent level, the following subsection presents a style for service-oriented architectures as introduced in Sect. 2. It extends the platform-independent style by SOA-specific concepts like service publication and discovery. In Sect. 5, we then explain how business-level architecture models and scenarios can be refined to the SOA style.

The SOA style does not model any vendor-specific platform. It rather represents the general SOA-specific mechanisms for service publication and service discovery. Other aspects that go beyond like quality-of-service, security, or mobility can be represented by even more specific styles which form a refinement hierarchy down to vendor-specific platform models. Our stepwise refinement approach can then be extended to this hierarchy.

Type graph: Figures 7 and 8 show the type graph of the SOA style. Due to the increased complexity, the class diagram is subdivided into two separate packages, Structure and Communication. They contain the same types as the platform-independent type graph from Fig. 4 but specialize some of them by subtyping and add further types for SOA.

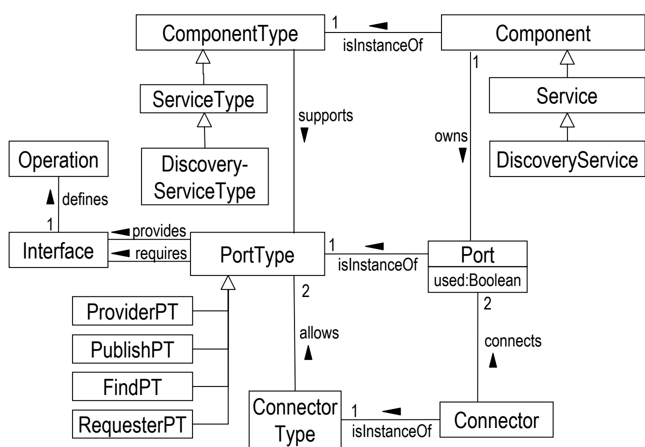


Fig. 7 Type graph of SOA style (package Structure)

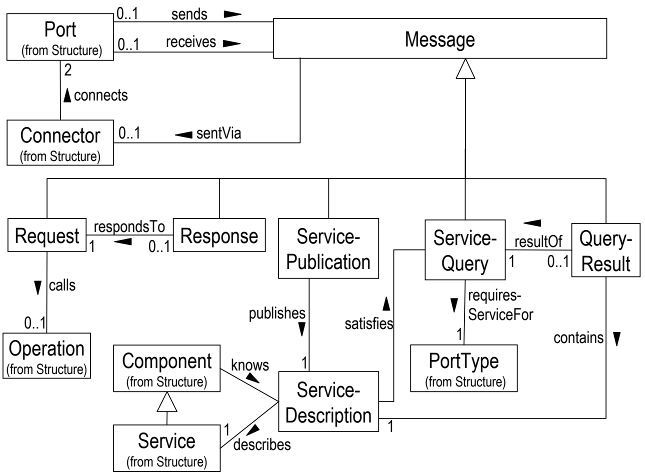


Fig. 8 Type graph of SOA style (package Communication)

The first package Structure (Fig. 7) contains subtypes of Component which can be used to declare software components as Service or, if functioning as discovery agency, as DiscoveryService. ComponentType is specialized accordingly. Also, there are special subtypes of PortType which are used to define dedicated port types for interactions with discovery services.

The second package Communication (Fig. 8) extends the elements for message-based communication known from the business style. The central SOA element here is ServiceDescription which describes a specific Service (in SOA, descriptions refer to deployed, addressable services rather than to service types only). The knows relationship indicates which components have access to a description. The existence of such a knows relationship is

a precondition for connecting to a service as shown by the transformation rule connect in Table 2.

Besides the already known Request and Response messages, there are three special SOA message types for interactions with discovery services, namely ServicePublication, ServiceQuery, and QueryResult. The first one submits a service description to a discovery service for publication, the second one refers to a port type which the service requester requires a suitable service for, and the third one is returned by the discovery service containing a description that satisfies the query.

Constraints: Along with the extended type graph comes an extended set of constraints which we do not present in detail here. For instance, they ensure that a DiscoveryServiceType always supports port types of kind PublishPT and FindPT. Also, they restrict possible sender and receiver ports for SOA-specific messages; e.g., a ServicePublication message can only be sent from ProviderPT ports to PublishPT ports.

We can now model the initial configuration of the SmartCar system in the SOA style. We take the business-level instance graph from Fig. 5 and change all components except for Vehicle into the new SOA types Service and ServiceType. We attach a ServiceDescription to all services. Furthermore, we add a DiscoveryService which is used to dynamically discover services. Additional port types are inserted to enable communication with the DiscoveryService. The result is shown in Fig. 9.

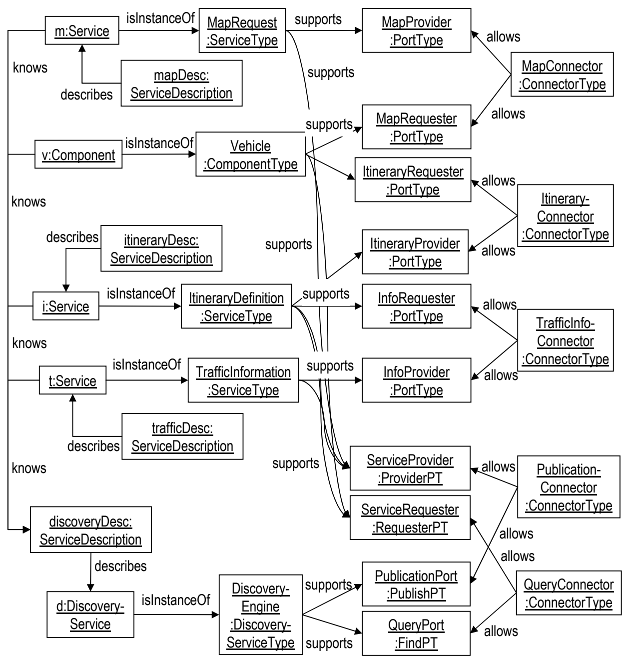


Fig. 9 SOA-specific instance graph for the SmartCar system

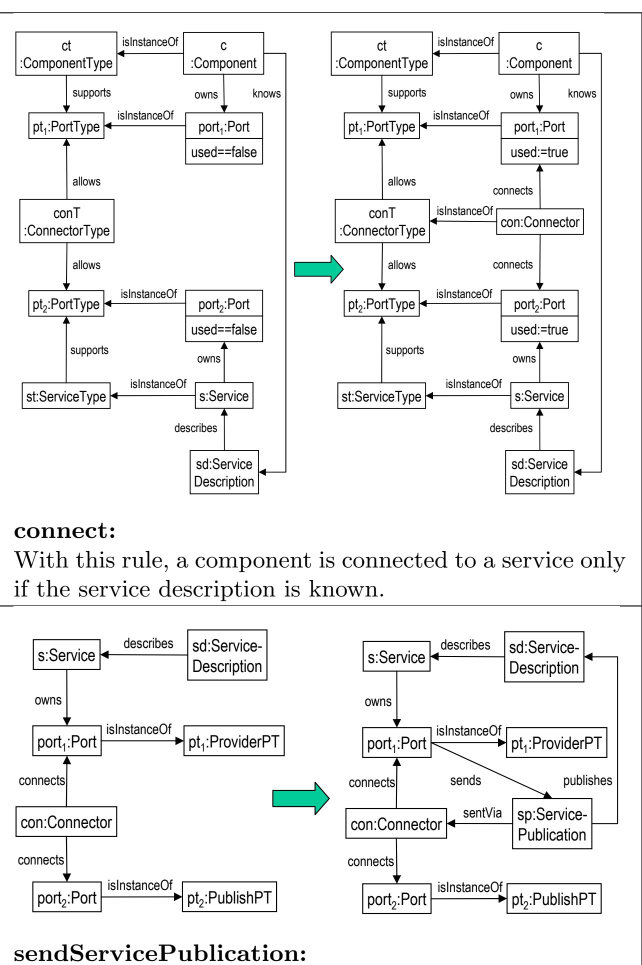
Since SOA-specific instance graphs become larger and more complex than the platform-independent ones, their readability decreases. In Sect. 4, we explain how to define a mapping between the type graph and a customized version of the UML meta-model in order to allow for a better concrete notation.

Transformation rules: Similarly to the type graph, the SOA style also inherits the transformation rules defined for the business style in Table 1. The only rule that is modified is the rule connect. Its SOA-specific variant, shown at the beginning of Table 2, has a stronger precondition demanding that a Component knows the ServiceDescription of a requested Service before a Connector to the service can be created.

In order to establish the required knows relationship, a number of other SOA-specific rules have to be applied first, which model the mechanisms for service publication and query. These additional rules are found in Table 2 after the connect rule.

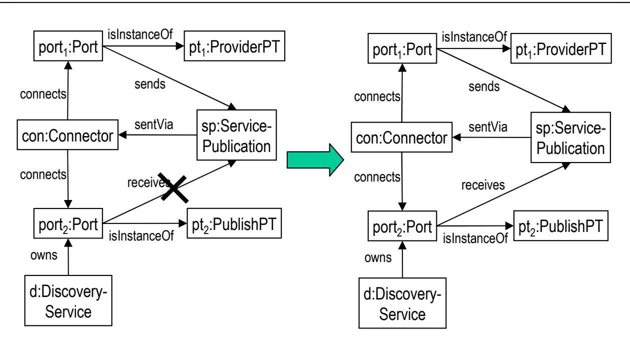
There are three rules dealing with service publications and six rules dealing with service queries. After a query has been submitted (sendServiceQuery) and received by the discovery service (receiveServiceQuery), the

Table 2 Transformation rules of the SOA style

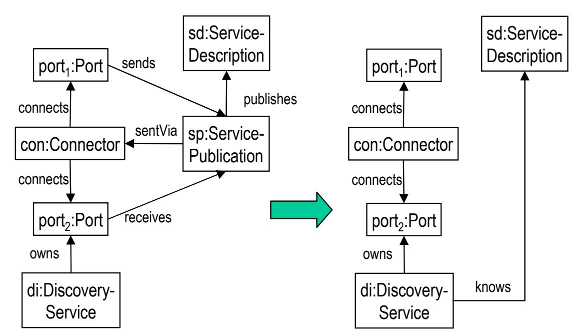


connect:
With this rule, a component is connected to a service only if the service description is known.

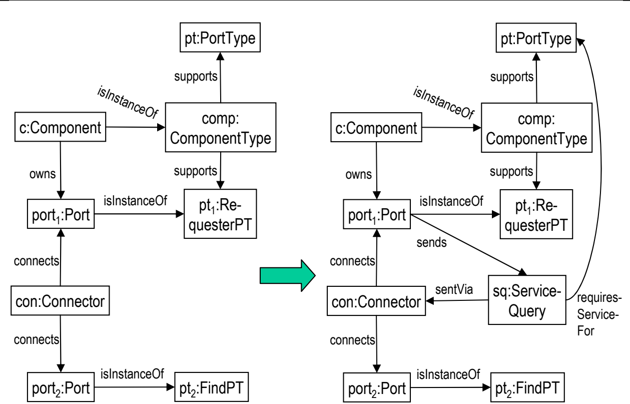
sendServicePublication:
A service submits its description to a discovery service.



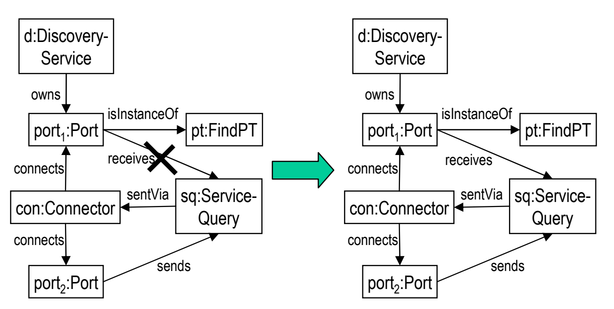
receiveServicePublication:
A discovery service receives a new request for publication.



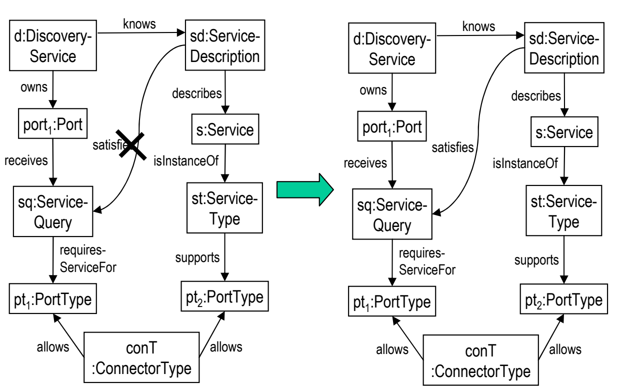
publishServiceDescription:
The discovery service stores the description of the service to be published.



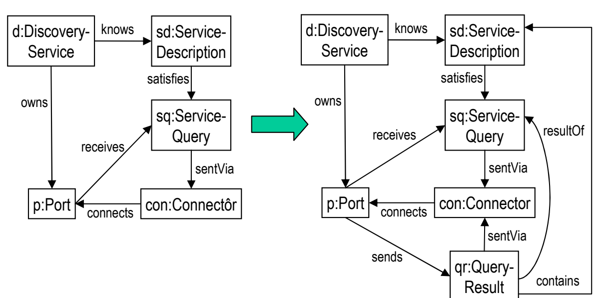
sendServiceQuery:
A component sends a service query to a discovery service.



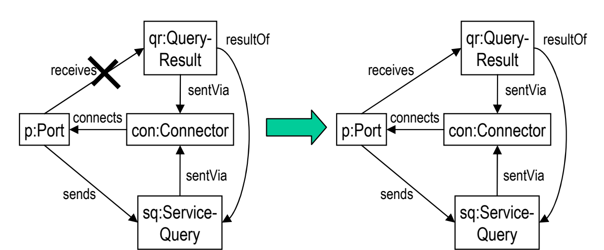
receiveServiceQuery:
A discovery service receives a service query.



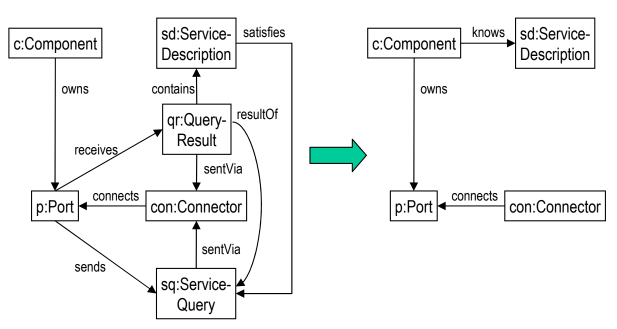
findService:
The discovery service selects an appropriate service description which satisfies the service query. This is the case if there is a compatible connector type for the port types of requester and provider.



sendQueryResult:
The discovery service sends a response with a satisfying service description.



receiveQueryResult:
The service requester receives the result of a query.



saveQueryResult:
The service requester finishes the query communication and stores the description of the found service.

rule findService is applied to decide which service description satisfies the query. A necessary condition is that there exists a ConnectorType which allows the connection of the requester port type and the port type of the service. If there are several candidates, the rule is applied non-deterministically since we abstract from detailed specification matchings in this style.

4 UML Notation for SOA-specific models

Graph transformation is a powerful formalism but, at the same time, rather difficult to use in practice because instance graphs easily grow very large. Also, the proposed notation for instance graphs does not provide a symbolic distinction between different element types which makes it difficult to read the diagram.

To address these problems, we propose to use the graph representations as underlying formalism, which can internally be handled by tools, and to place a user-friendly notation layer like the *Unified Modeling Language* (UML) on top. UML is a well-known modeling language and the de-facto standard for object-oriented modeling in industry.

Another advantage of UML is its built-in *extension mechanism* [34]. In our case, this mechanism can be used to provide a distinguished notation for style-specific elements that are not represented by the UML core. For this purpose, one defines a *UML profile* that consists of so-called *stereotypes*. Each stereotype extends and adapts classes of the UML meta-model by defining refined semantics, additional attributes, constraints, and, optionally, a new distinguished notation.

While the standard UML might be sufficient to model platform-independent architectures (cf. [29]), we propose to define dedicated UML profiles for platform-specific models. Following the tradition of [39, 29], we choose an existing meta-class from the UML meta-model that is semantically close to a construct from the platform-specific architectural style and define a stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the architectural style.

The correspondence between the resulting UML models and their equivalent graph-based representation can be maintained by special conversion tools as described in Sect. 6. Based on a mapping between the (extended) UML meta-model and the type graph of the corresponding architectural style, such tools can translate between the UML models exported from a CASE-tool and the corresponding graph-based representation.

Below, we apply the concept to our service-oriented style. At first, we define a UML profile for SOA as a set of stereotypes that extend selected classes of the UML 2.0 meta-model. Then, we provide a notation guide for these new stereotypes. Eventually, we define the mapping between the SOA type graph and the extended UML meta-model as conceptual basis for conversions between the two representations.

Stereotypes: Figure 10 defines the SOA-specific stereotypes for the UML 2.0 meta-model. The extended UML meta-classes are shown at the top of the diagram, the corresponding stereotypes below the dashed line. The extension relationship is indicated by arrows with small, filled arrow-head. Stereotypes can be specialized by subtypes.

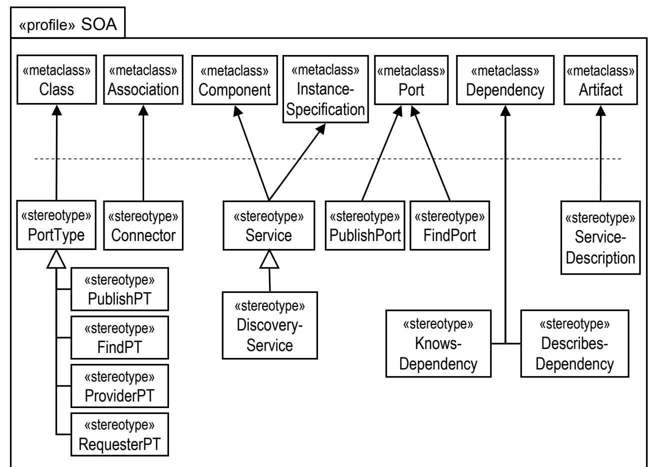


Fig. 10 Stereotypes of the UML profile for SOA

Port types and connector types can be modeled with class diagrams. Thus, we introduce the stereotype PortType and its sub-stereotypes as extensions of the UML meta-class Class. Associations between these port type classes model connector types and are marked by the stereotype Connector. In the same class diagram, one can also specify the interfaces that a port type provides and requires (cf. Fig. 11).

With the help of the two stereotypes Service and DiscoveryService, we can label components that are to be exposed as services. Since services occur as both types and instances thereof, these stereotypes can be attached to the meta-classes Component as well as InstanceSpecification. These constructs are used in component diagrams (cf. Fig. 12), where we can also specify which port types from the aforementioned class diagram are supported by a component or service type, and in communication or sequence diagrams modeling interactions between various (instances of) components and services.

UML 2.0 knows the notion of Port as an interaction point to a component instance. Thus, we do not need a separate stereotype for ports in general. But, in order to highlight those ports that are used for service publication and query, we introduce the stereotypes PublishPort and FindPort.

The stereotype for service descriptions is based on the meta-class Artifact. The relationships to these descriptions are modeled by the stereotypes KnowsDependency and DescribesDependency.

Table 3 Notation guide for SOA stereotypes

Stereotype from SOA profile:	Notation:
PortType	«portType»
PublishPT	«publishPT»
FindPT	«findPT»
ProviderPT	«providerPT»
RequesterPT	«requesterPT»
Connector	«connector» Name
Service	«service» ServiceName
DiscoveryService	«discovery» Name
ServiceDescription	Name D
PublishPort	P
FindPort	F
KnowsDependency	«know»
DescribesDependency	«describe»

Notation: The default convention for the notation of stereotyped diagram elements is to attach the stereotype name within a pair of guillemets to the symbol of its meta-class. Nevertheless, one can also define more customized notations. The notation of the SOA stereotypes is summarized in Table 3.

Relating style and UML meta-model: The relationship between the UML notation and the formal architectural style is defined by a bi-directional mapping between the type graph of the style and the extended UML meta-model. With the help of such mapping, we can render a given instance graph as UML diagrams, and, in the opposite direction, we can provide the semantics for a given UML model in terms of the architectural style.

Table 4 shows the mapping for the SOA style and the UML profile for SOA. The left column contains the nodes (and some of the edges) of the SOA type graph (cf. Fig. 7 and 8). The right column contains the corresponding meta-classes and stereotypes that should be used in UML diagrams to depict the various concepts of the SOA style.

Edges in the type graph represent relationships between nodes. If there are similar relationships in the UML meta-model (e.g., for the operations defined by an interface), then we can omit the mapping of edges. Two exceptions are the knows and describes edges because their counterparts on the UML side are real meta-classes and stereotypes.

Table 4 Mapping between SOA style and UML

SOA type graph elements	**UML meta-model and profile elements (package name::class name)**
ComponentType	BasicComponents::Component
Component	Kernel::InstanceSpecification
ServiceType	BasicComponents::Component stereotyped by SOA::Service
Service	Kernel::InstanceSpecification stereotyped by SOA::Service
DiscoveryService-Type	BasicComponents::Component stereotyped by SOA::DiscoveryService
DiscoveryService	Kernel::InstanceSpecification stereotyped by SOA::DiscoveryService
PortType	Kernel::Class stereotyped by SOA::PortType
ProviderPT	Kernel::Class stereotyped by SOA::ProviderPT
PublishPT	Kernel::Class stereotyped by SOA::PublishPT
FindPT	Kernel::Class stereotyped by SOA::FindPT
RequesterPT	Kernel::Class stereotyped by SOA::RequesterPT
Port	*[If self.portType.oclIsTypeOf(PortType)]* Ports::Port
	[If self.portType.oclIsTypeOf(PublishPT) or self.portType.oclIsTypeOf(ProviderPT)] Ports::Port stereotyped by SOA::PublishPT
	[If self.portType.oclIsTypeOf(FindPT) or self.portType.oclIsTypeOf(RequesterPT)] Ports::Port stereotyped by SOA::FindPT
ConnectorType	Kernel::Assocication stereotyped by SOA::Connector
Connector	Kernel::InstanceSpecification
Interface	Interfaces::Interface
Operation	Kernel::Operation
ServiceDescription	Artifacts::Artifact stereotyped by SOA::ServiceDescription
knows	Dependencies::Dependency stereotyped by SOA::KnowsDependency
describes	Dependencies::Dependency stereotyped by SOA::DescribesDependency

If there are several notation options for the same type graph element, then we can distinguish these options by additional OCL constraints as shown in Table 4 for Port.

Since, we restrict the UML profile to visualizing the structure of a system only, we omit a mapping of the various message types for communication.

Example diagrams: Figures 11 and 12 give an impression, how the model layout is improved when the SOA profile is applied. They visualize the SOA instance graph of the SmartCar system shown in Fig. 9.

Figure 11 presents a class diagram which defines all port types together with their provided (triangle arrowhead) and required (use dependency) interfaces. The component diagram in Fig. 12 reveals which service or component supports which port type and which service descriptions are known by which components at the beginning.

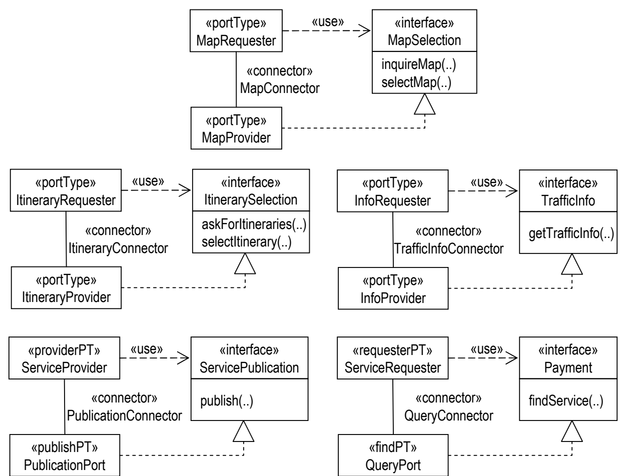


Fig. 11 SOA-specific class diagram for SmartCar

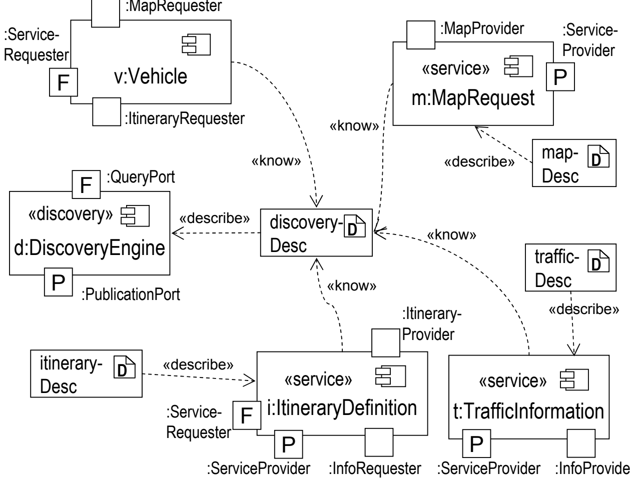


Fig. 12 SOA-specific component diagram for SmartCar

5 Behavior-preserving architecture refinement

Based on the architectural styles defined in Sect. 3 and the UML profile defined in Sect. 4, we can now model system architectures at the business-level as well as for service-oriented platforms, and we can provide an operational semantics for communication and reconfiguration scenarios in terms of graph transformations. The underlying conceptual platform model, in our case the architectural style for SOA, ensures that the architecture is consistent with the provided platform mechanisms.

The remaining problem we want to address in this section is how to ensure the consistency between architecture models in the abstract, business-oriented and the platform-specific, service-oriented style. Since these two styles represent different levels of platform abstraction, the desired consistency relationship can be defined by an appropriate notion of *architecture refinement.*

To be a valid refinement of a business-level architecture, a platform-specific or, in our case, service-oriented architecture has to realize the same functionality. This requirement can be subdivided into

1. **Structural refinement:** The platform-specific architecture has to preserve all business-relevant, functional entities and all required connections between these entities.
2. **Behavior-preserving refinement:** The platform-specific architecture has to enable all communication and reconfiguration scenarios which can also occur at the business level.

Our notion of refinement should be *style-based*, i. e., based on a relationship between the abstract, platform-independent style and the SOA-specific style which can be reused for refining any instances of these styles. For this purpose, a mapping between the two type graphs is used to induce an *abstraction function* that projects instance graphs from the concrete style to the abstract style. The rationale behind using an abstraction function rather than a refinement function is the fact that abstraction is in general simpler and more deterministic than refinement.

Based on the abstraction function, we can *check* if a given instance graph in the SOA style is a refinement of a given business-level instance graph. A similar criterion applies to transformation sequences representing reconfiguration and communication scenarios.

In order to *derive* refined, SOA-specific scenarios from given business-level scenarios including operations for service publication and discovery, we reformulate this problem as a *reachability problem* which can automatically be solved by graph transformation or model-checking tools as described in Sect. 6.

5.1 Refinement criterion for instance graphs

As mentioned above, we use an abstraction function as refinement criterion which is induced by a map-

ping at the style level. For the case of service-oriented architectures, let the platform-independent (pi) style from Sect. 3.1 be $\mathcal{G}^{pi} = \langle TG^{pi}, C^{pi}, R^{pi}\rangle$ and the service-oriented (so) style from Sect. 3.2 be $\mathcal{G}^{so} = \langle TG^{so}, C^{so}, R^{so}\rangle$. Then, we introduce a type mapping $t : TG^{so} \to TG^{pi}$, formally a *partial* surjective graph homomorphism, which maps elements of the SOA type graph TG^{so} to the elements of the platform-independent type graph TG^{pi}.

The concrete definition of t is driven by semantic correspondences between the elements of the two styles. We distinguish three different cases which are illustrated in Fig. 13:

1. Since the SOA-specific type graph is an extension of the platform-independent type graph, all nodes and edges of the latter also occur in the former. In these cases, the SOA elements are mapped to their equivalent in the platform-independent type graph. This way, the abstraction mapping t becomes surjective. For instance, as shown in Fig. 13, t maps the SOA type Component to the platform-independent type Component, and similarly with ComponentType.
2. Since services are a SOA-specific interpretation of components, t maps Service and ServiceType to the platform-independent types Component and ComponentType, too.
3. All other types (and adjacent edges) like DiscoveryService, ServiceDescription, or the SOA-specific port types and messages represent purely platform-specific concepts which do not occur at the business level. Therefore, these elements are not mapped to the platform-independent type graph.

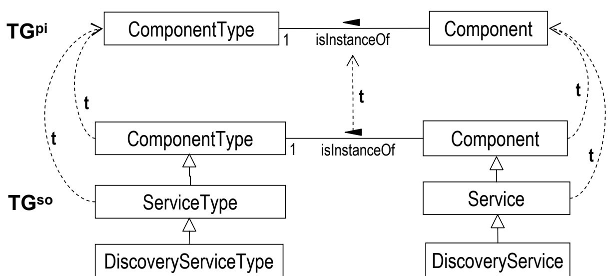


Fig. 13 Part of the type graph mapping t

The type mapping t induces the desired abstraction function $abs_t : \mathbf{Graph}_{TG^{so}} \to \mathbf{Graph}_{TG^{pi}}$ which 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 14 illustrates the effect of the abstraction function abs_t for an instance graph fragment which defines the MapRequest service in the SOA style. First, we apply the type mapping t and rename the types of the Service and ServiceInstance nodes into Component and ComponentInstance (1). 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).

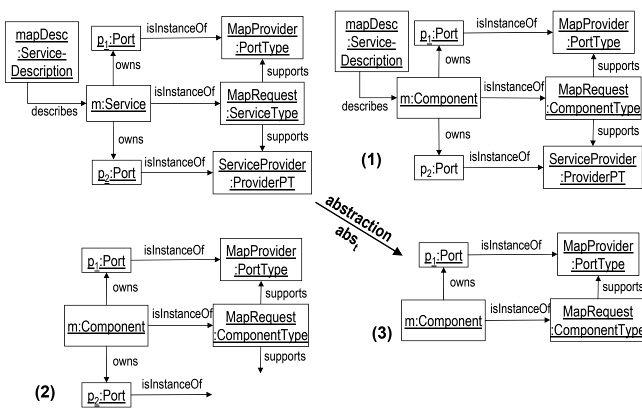


Fig. 14 Abstraction of an instance graph

Since we defined the cardinalities and constraints C^{so} in $\mathcal{G}^{so}$ stronger or as strong as the constraints C^{pi} in $\mathcal{G}^{pi}$, the abstraction of instance graphs is compatible with the constraints, that is, if G^{so} satisfies C^{so}, then $abs_t(G^{so})$ satisfies C^{pi}, too.

A service-oriented instance graph G^{so} is called a *refinement* of a platform-independent graph G^{pi}, if its abstraction into the platform-independent style reflects exactly the elements of G^{pi}, i.e., if $abs_t(G^{so}) = G^{pi}$. This definition ensures that the SOA-specific refinement preserves all functional, business-relevant entities occurring in the abstract, business-oriented architecture.

As an example, consider the graph in the upper left of Fig. 14 which is obviously a refinement of graph (3) in the lower right of the figure. Another example is the SOA-specific configuration for SmartCar, shown in Fig. 9, which refines the platform-independent configuration shown in Fig. 5 because the application of the abstraction function to the former yields the latter.

The above defined refinement criterion helps to *check* for refinements of individual system configurations as instance graphs. In order to actually *construct* the refined configurations, we refer to existing work on structural refinements such as [1,31]. Since our focus is on the refinement of scenarios, we assume that the architect uses heuristics or one of the available techniques

in order to derive correct SOA-specific configurations (with, e.g., discovery service and service descriptions) from platform-independent ones according to the above refinement criterion. Nevertheless, plain structural refinement is not sufficient to refine the behavioral aspects of a scenario as described below.

5.2 Refinement criterion for transformations

According to Sect. 3, a reconfiguration and communication 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 $s^{pi} = (G^{pi} \Rightarrow H^{pi})$ in the platform-independent graph transformation system $\mathcal{G}^{pi}$, the transformation sequence $s^{so} = (G^{so} \Rightarrow^*_{\mathcal{G}^{so}} H^{so})$ in the service-oriented transformation system $\mathcal{G}^{so}$ is a correct refinement, if G^{so} refines G^{pi} and H^{so} refines H^{pi} (formally, $abs_t(G^{so}) = G^{pi} \wedge abs_t(H^{so}) = H^{pi}$).

The refinement s^{so} is a transformation *sequence* rather than a single *step* because, at the platform-specific level, it might be necessary to perform a number of consecutive steps to realize the platform-independent step.

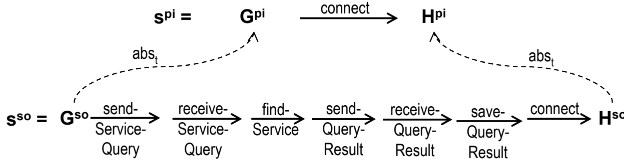


Fig. 15 Refinement of a transformation step

As an example, consider the transformation step s^{pi} of Fig. 15 which contains the application of the platform-independent rule connect (cf. Table 1). For the service-oriented refinement of this step, we have to use the SOA variant of the connect rule (cf. Table 2) which requires as precondition that the service description is known to the service requester. Therefore, it becomes necessary to submit a service query to a discovery service before the connect operation can be applied. Thus, we add corresponding rule application to the service-oriented refinement s^{so} shown at the bottom of Fig. 15.

The criterion for transformation steps is easily extended to sequences $s^{pi} = (G_0^{pi} \Rightarrow^*_{\mathcal{G}^{pi}} G_n^{pi})$ of length greater than one: A sequence $s^{so} = (G_0^{so} \Rightarrow^*_{\mathcal{G}^{so}} G_n^{so})$ over the SOA style is a valid refinement of s^{pi}, if s^{so} can be partitioned into consecutive *subsequences* that are refinements of the individual transformation *steps* of s^{pi}.

5.3 Construction of refined transformation sequences

To actually construct the refined transformation sequence, we stick to the stepwise view and decompose the abstract sequence s^{pi} into its individual steps $s_k^{pi} = (G_k^{pi} \Rightarrow G_{k+1}^{pi})$. Each step is then transformed into a *reachability problem* which can be solved by analysis tools.

Consider the first step $s_0^{pi} = (G_0^{pi} \Rightarrow G_1^{pi})$. We assume that there is a correctly refined start graph G_0^{so} in the SOA style with $abs_t(G_0^{so}) = G_0^{pi}$. Then, the first reachability problem is to find the shortest transformation sequence of SOA-specific rule applications which leads from G_0^{so} to an instance graph G_1^{so} that refines the target graph G_1^{pi}.

The length of the SOA-specific transformation sequence is required to be minimal, because we want to reach the target configuration without any superfluous steps that could have additional effects on business-relevant elements like, for instance, creating any extra connector that is not required by the target graph G_1^{pi}.

If the search within the service-oriented transformation system is successful, the reached instance graph can be taken as new start graph for the second step s_1^{pi}, and so on. If we repeat the procedure for all steps of the transformation sequence and concatenate the resulting SOA-specific transformation sequences, we receive a complete refinement of the platform-independent scenario s^{pi}.

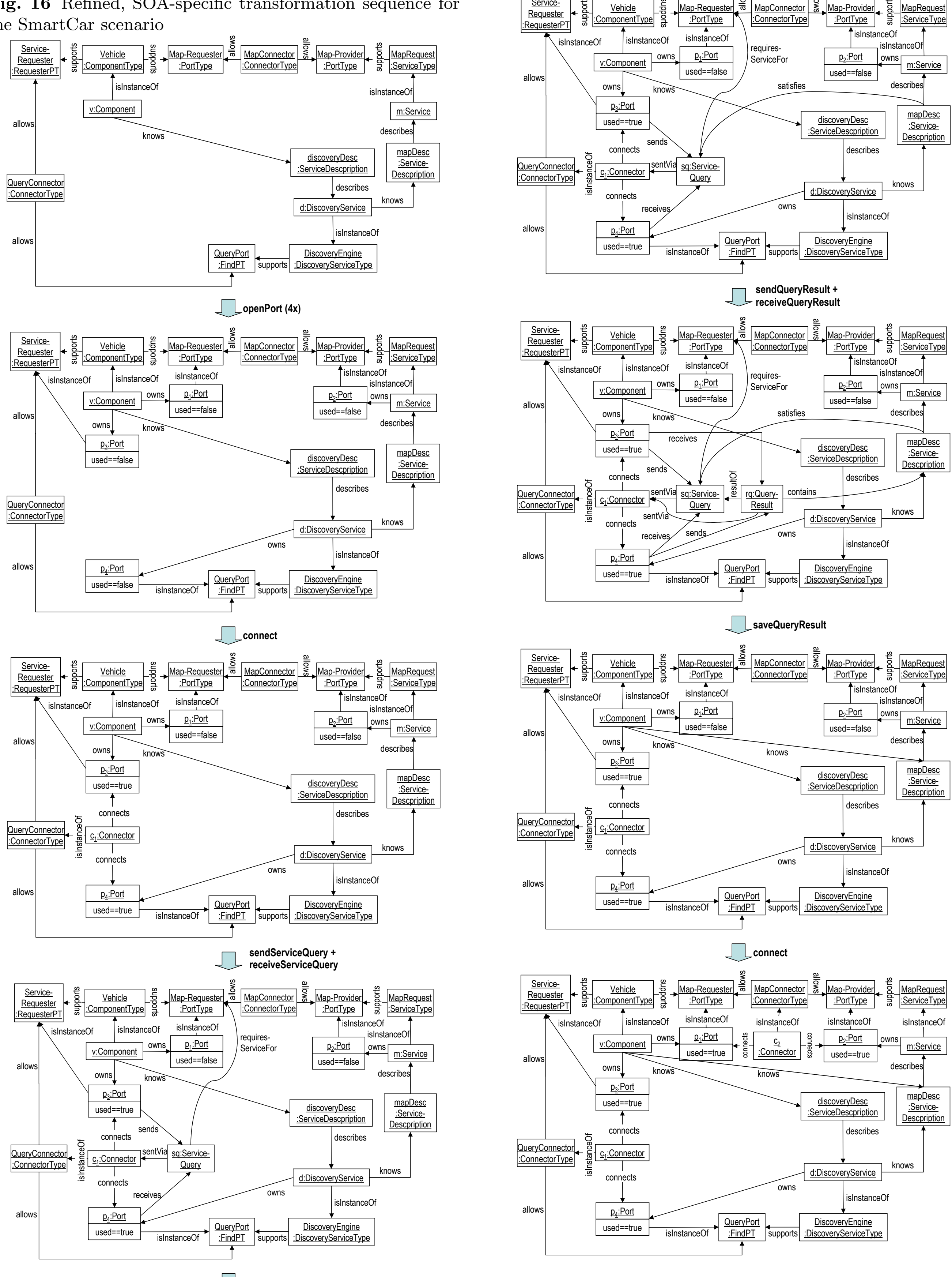
If the search fails and at least one of the steps to be refined cannot be expressed as a transformation sequence at the SOA level, then this might be caused by some missing elements in the initial configuration of the service-oriented architecture. For example, if one of the components that needs to use a service does not know the description of the responsible discovery service and, thus, cannot connect to it for submitting a service query, then this component cannot connect to the required service, either. This way, the solution of the reachability problems can also be used to validate the correctness and completeness of the initial SOA configuration.

Example: We illustrate the refinement of scenarios for the SmartCar scenario which is partially depicted in Fig. 6 as a transformation sequence in the platform-independent style. The depicted part represents the creation of a new connector between the Vehicle component and the MapRequest component and consists of four instance graphs, which we now name G_0^{pi}, G_1^{pi}, G_2^{pi}, and G_3^{pi}, and the three transformation steps

$$G_0^{pi} \overset{openPort}{\Longrightarrow} G_1^{pi} \overset{openPort}{\Longrightarrow} G_2^{pi} \overset{connect}{\Longrightarrow} G_3^{pi}$$

The refinement of this transformation sequence is depicted in Fig. 16. The individual steps of the refined transformation sequence are labeled by the applied SOA-specific rules. We do not highlight to which part of the graph a rule has been applied since this can be derived from the outcome of a rule application, and, for the sake of brevity, we have summarized some consecutive transformations into single steps. For the definition of the individual rules please refer to Table 2.

Fig. 16 Refined, SOA-specific transformation sequence for the SmartCar scenario



The start graph of the refined sequence in Fig. 16 equals the SOA instance graph from Fig. 9 which is a valid refinement of the platform-independent start graph G_0^{pi} from Fig. 6. It contains all relevant parts for the Vehicle component, the MapRequest service and the DiscoveryEngine discovery service. In order to shorten the example, we assume that the service description mapDesc of the MapRequest service has already been published to the discovery service (as indicated by the knows edge).

The refinement of the transformation sequence starts with the first two transformation steps representing two invocations of the openPort operation. Their refinement into a SOA-specific scenario is quite trivial as we can simply apply the equivalent openPort operations of the SOA style.

More difficult is the refinement of the business-level connect operation. In this case, we cannot simply apply the corresponding SOA variant to the last intermediate result because the SOA variant of the connect rule requires a knows link to the service description mapDesc which is not yet existent. For this reason, we have to try the application of other rules in order to find a transformation sequence to a valid refinement of G_3^{pi}.

The minimal solution to this reachability problem can be found in Fig. 16 after the first two openPort operations. In summary, it comprises two further openPort operations that "prepare" a connect operation to the discovery service, sending and receiving a query to the discovery service, finding an appropriate service, and submitting the query response with the required service description. Eventually, the desired connect operation is applicable after the knows links has been created by saveQueryResult (cf. also Fig. 15).

If we continue the refinement procedure until the entire business-level scenario of the SmartCar application is refined to the SOA-specific level, we receive a platform-specific reconfiguration and communication scenario which can then be rendered as a UML diagram again. Figure 17, for instance, shows the interactions of the refined scenario as a UML sequence diagram.

We do not have to perform the described reachability searches manually. As discussed in the next section, existing analysis and graph transformation tools can automatically select applicable transformation rules and test the effect of their application in order to automate the presented refinement approach.

6 Tool support

While conceiving the approach presented so far, we did not concentrate on designing a brand-new tool, but decided to exploit existing tools as components of a tool chain. Even if this paper concentrates on describing the concepts of the approach, we briefly discuss the tool support required and how we can reuse existing tools.

Roughly, we can split the task into the creation of UML models and graph transformation systems and the

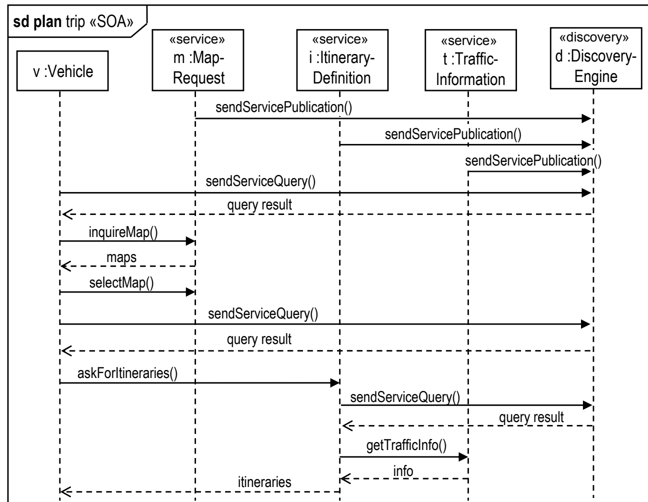


Fig. 17 UML sequence diagram of SOA-specific scenario

support for model refinement. The latter, involving the reachability analysis for a target configuration from a given initial configuration, is critical for the proposed refinement technique. This analysis can be performed using both model checking techniques and simulation features of graph transformation tools. In both cases, the solution to the reachability problem has to be integrated with the tools to create graph transformation systems and UML models.

6.1 Modeling tools

The creation of UML models is a standard task and does not require special attention. Here, we can use off-the-shelf UML CASE tools, like Poseidon[3], to define models and add suitable annotations (stereotypes) to decorate them with additional information. All UML tools support the XML Metadata Interchange (XMI) format [35] as a standard and vendor-independent way to store and exchange user models.

The creation of graph transformation systems is supported by tools, like AGG[4], PROGRES [41], and Fujaba[5], which allow the specification of rules in various notations as well as their application to a given graph. As a common XML format for graph transformation systems, the *Graph Transformation eXchange Language* (GTXL)[6] is being developed. It is based on the *Graph eXchange Language* (GXL)[7] [47] and will shortly be supported by several graph transformation tools.

[3] www.gentleware.com
[4] tfs.cs.tu-berlin.de/agg
[5] www.fujaba.de
[6] tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html
[7] www.gupro.de/GXL

6.2 Reachability analysis by model-checking

Model checking of graph transformation systems has already been investigated by Varró with the CheckVML tool [44]. This subsection gives a brief overview how the technique can be applied to solve reachability problems. The interested reader is referred to [4, 45] for more technical details.

The *model checking problem* consists in deciding by exhaustive simulation whether a certain correctness property holds in a given transition system. That requires a systematic traversal of all possible execution paths of the system, i.e., all enabled transitions must be taken in all reachable states. The properties are typically formalized as temporal logic formulae.

From graph transformation systems to transition systems: The system specification languages of most model checkers are based on *transition systems*, where the structure of a state consists of (a subset of) propositions over an *a priori* finite universe. When translating graph transformation systems into transition systems (as done by the CheckVML tool, mapping graph transformation systems to Promela, the input language of the SPIN model checker [24]) a graph is interpreted as a state, while the application of a rule for a certain occurrence of the left hand side in such a graph yields a transition in the transition system. Traversing all enabled transitions then means applying all rules with all possible occurrences.

The main challenge consists in bridging the gap in the abstraction levels: Graph transformation rules define how an arbitrary instance of a type graph should behave, while transition system specifications in, e. g., Promela are given for specific instances. That requires to generate Promela transitions for all the potential applications of a graph transformation rule during a compile-time preprocessing phase.

Moreover, sophisticated optimization techniques are needed in order to reduce the state space. For example, one can distinguish *static* and *dynamic* model elements in the type graph (only the latter are modified by graph transformation rules), representing only the dynamic parts of instance graphs as states of the target transition system. The occurrences of the rules in the static parts, instead, yield additional constraints on the potential execution paths.

A drawback of the model checking approach is its *a priori* restriction to finite state systems. Therefore, one has to fix an upper bound for the number of dynamic model elements that can be created by the transformation rules. If the analysis is not successful, one can increase the bound and repeat the analysis within certain limits.

From graph patterns to logic properties: The reachability problem of a certain target configuration can be expressed by *safety* and *reachability* properties.

– A *safety property* defines a desired property that should always hold on every execution path or (equivalently) an undesired situation which should never hold on any execution paths.
– A *reachability property* describes, on the contrary, a desired situation which should be reached on at least one execution path.

From a model checking point of view, safety and reachability properties are dual: the refutation of a safety property is a counter-example which satisfies the reachability property obtained as the negation of the safety property. On the other hand, if a safety property holds (or a reachability property is refuted) the model checker has to traverse the entire state space.

A safety or reachability property can be interpreted as a special graph pattern (called *property graph*) which immediately terminates the verification process if it is matched successfully. As shown in [37], the properties expressible in this way are equivalent to the $\exists\neg\exists$ fragment of ($\forall$-free) first order logic with binary predicates.

An alternative solution for model checking graph transformation systems has been proposed by Rensink in the GROOVE system [36]. The essence of the approach is to use the core concepts of graphs and graph transformations all the way through during model checking. This means that states are explicitly represented and stored as graphs, and transitions as applications of graph transformation rules. Furthermore, graph-specific model checking algorithms are applied for traversing the state space. This solution exploits the symmetric nature of problems by intensive graph isomorphism checks.

A comparison on the two approaches for model checking graph transformation systems can be found in [38].

6.3 Reachability analysis by simulation

In order to avoid the complete generation of the state space of a model, and thus allow the refinement of infinite state systems, we may use graph transformation tools for the interactive simulation. The PROGRES [41] tool is especially suitable for this purpose since it supports, besides the mere execution, depth-first search and backtracking. In PROGRES, we can define a type graph, the set of transformation rules, the start graph, and constraints for safety invariants. The reachability property is represented by a so-called *test graph* which models the target pattern.

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 can define PROLOG-like cuts that interrupt the backtracking at certain points and guarantee termination by limiting the search depth.

6.4 Tool integration

The integration of all these components can be carried out through suitable XML data: XMI representations of class and communication diagrams can be transformed into GXL documents, e.g., using XSLT scripts or simple tools.

GXL is the format supported by both AGG and CheckVML. The former exploits GXL for type and start graphs, while the latter uses GXL graphs for the initial configuration and the encoding of properties (i.e., scenarios). AGG and CheckVML can also exchange graph transformation rules encoded in GTXL, the Graph Transformation eXchange Language.

A data flow, which is still missing, but nevertheless important, is the propagation of analysis results produced by the model checker back to the UML case tool. This flow does not only require a suitable data format, but also a rigorous transformation of traces and counter examples into meaningful decorations of UML elements.

6.5 User roles

The tools and artifacts discussed above imply two kinds of users, namely *style architects* and *application architects*.

If the presented styles for business- and service-oriented architectures are to be modified or adapted to other platforms, then users who are proficient in both graph transformation and architectural styles can serve as *style architects* to design the graph transformation systems that specify platform-specific concepts and reconfiguration mechanisms. The style architect also defines the 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. Eventually, the style architect has to relate the styles for different levels of platform abstraction by suitable refinement relations.

While specific rules and mappings are required for each architectural style, several architectures can exploit the same style. 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.

7 Related work

The work presented in this paper is rooted in four main research directions: architecture description languages, model checking of software architectures, graph transformations, and architectural refinement.

Besides the many proposals for Architecture Description Languages (ADLs), like Rapide [27], Wright [3], or Darwin [28], we must mention those approaches that exploit graph transformation [21,22,25,43,46] to reason on the consistency of reconfiguration operations and interaction of components with respect to structural constraints. Our work is in this tradition, but it combines the formal approach with the notion of style-based refinement.

Le Métayer [25] describes architectures by graphs and the valid graphs of an architectural style by a graph grammar. Reconfiguration is described by conditional graph rewriting rules. He uses static type checking to prove that the rewriting rules are 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 [46] provide an algebraic framework based on Category theory where architectures are represented as graphs of CommUnity programs and superpositions. The architectural style, given as a type graph, restricts the ways connectors can be applied to components. Dynamic reconfigurations are 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 [21], 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 run-time interactions among components, reconfigurations, 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 [15] 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.

The use of model checking techniques for verifying software architectures has been thoroughly studied by several proposals. vUML [26], veriUML [10], JACK [17], and HUGO [40] support the validation of distributed systems, where each statechart describes a component, but do not support any complex communication paradigm.

JACK and HUGO only support communication based on brodcasting, where the events produced by a component are notified to all the others. vUML and veriUML support the concept of a channel, that is, each component writes and reads messages on/from a channel. These proposals aim at general-purpose applications and can cover different domains, but are not always suitable when we need a specific communication paradigm.

They study static systems whose topology cannot vary at run-time. Similarly, Garlan et al. [12] and the researchers involved in the Cadena project [18] applied model-checking techniques to analyze specific architectures based on the publish/subscribe paradigm. The fixed topology distinguishes these approaches from our work. In fact, we propose the study of the dynamic evolution of architectures with almost no attention to the internals of components. Given our interest, we treat components as black-box entities, while all these approaches analyze the behaviors of such components. They consider a given system (architecture) as if it were a complex and fixed automaton, but neglect the possibility that such automaton changes while the system evolves. Even if different, these approaches can also be seen as the natural complement of our approach: We study what they do not address and they analyze what we neglect, mainly because of the size of resulting models. So far, no proposal attempts to address the whole picture.

There are also 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 a refinement of architectures has first been discussed by Moriconi et al. in [31]. 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 refinement, however, could be applicable in our context, too.

Garlan [16] 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. 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 the dynamic semantics of reconfiguration and communication scenarios.

Other proposals on architecture refinement like [1,8, 13] 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.

8 Conclusions and future work

In this paper, we have given a formal definition of service-oriented architectures, seen as an architectural style. We have defined a refinement relation from a generic style of component-based systems to the SOA style that can be used to study the specialization of platform-independent scenarios, and we have discussed the use of model checking techniques and tools to automate this task. The results are based on the use of graph transformation systems as models of architectural styles at different levels of platform abstraction, representing reconfiguration and communication scenarios as graph transformation sequences.

While we demonstrated the approach for service-oriented architectures in this article, it should also be applicable to other kinds of middleware infrastructures modeled by corresponding architectural styles as sketched in Sect. 6.5.

As stated in Sect. 7, a current challenge is to combine descriptions and analysis of component behavior with runtime changes of component configurations. In a parallel paper [20], we elaborate on this problem and propose an extension of the architectural styles presented in this article. These extensions allow to equip active entities like components, services, and connectors with process definitions that prescribe the order in which communication and reconfiguration operations can be applied.

All applications of communication and reconfiguration rules have to respect the process definitions. This way, we integrate descriptions of component behavior and of topological changes which are required to realize the desired business processes. In [20], we demonstrate how this integration can be achieved without any new formal concepts. Consequently, we are still able to apply the aforementioned model checking-based analysis techniques. The restricted architectural behavior even facilitates the analysis due to the smaller overall state space. We also discuss in [20] how the behavior-preserving refinement can be guaranteed in face of the new process descriptions.

Our future work addresses the development of an integrated CASE environment for the analysis and stepwise refinement of software architectures which is a prerequisite for validating the approach on other non-

trivial examples. We are proficiently conducting experiments with existing graph transformation tools and model checkers in isolation, but the final objective is a tool chain that seamlessly integrates the different components. The problem is largely one of incompatible input formats. Only the backward translations of analysis results into user models poses conceptual questions.

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. 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.
5. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Style-based refinement of dynamic software architectures. In *Proc. WICSA4 – 4th Working IEEE/IFIP Conference on Software Architecture*, pages 155–164. IEEE Computer Society, 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. 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/`.
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. M. Champion, C. Ferris, E. Newcomer, and D. Orchard. *Web Service Architecture, W3C Working Draft*, 2002. `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`.
10. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An automatic verification tool for UML. Technical Report CSE-TR-423-00, University of Michigan, EECS Department, 2000.
11. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–265, 1996.
12. D. Garlan and S.Khersonsky and J.S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th SPIN Workshop*, volume 2648 of *LNCS*, May 2003.
13. 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/`.
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. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In *Proc. UML 2000 - The Unified Modeling Language*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
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. S. Gnesi, D. Latella, and M. Massink. Model checking UML statecharts diagrams using JACK. In *Proceedings of the 4th IEEE International Symposium on High Assuarance Systems Enginering (HASE)*, pages 46–55. IEEE Press, 1999.
18. J. Hatcliff, W. Deng, M.B. Dwyer, G. Jung, and V. Ranganath. Cadena: An integrated development, analysys, and verification environment for component-based systems. In *Proceedings of the 25th International Conference on Software Engineering*, pages 160–172, May 2003.
19. R. Heckel, M. Lohmann, and S. Thöne. Towards a UML profile for service-oriented architectures. In *Proc. of Workshop on Model Driven Architecture: Foundations and Applications (MDAFA)*, CTIT Technical Report TR-CTIT-03-27. University of Twente, Enschede, The Netherlands, 2003.
20. R. Heckel and S. Thöne. Behavior-preserving refinement relations between dynamic software architectures. In *Proc. of the 17th Int. Workshop on Algebraic Development Techniques, WADT 2004*, LNCS. Springer, 2004. to appear.
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. C. Hofmeister, R. Nord, and D. Soni. *Applied Software Architecture*. Addison-Wesley, 2000.
24. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
25. 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.
26. J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 255–258, October 1999.
27. 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.
28. 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.

29. N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J.E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(1):2–57, January 2002.
30. J. Miller and J. Mukerji. *MDA Guide Version 1.0.1*. Object Management Group, 2003. `www.omg.org/docs/omg/03-06-01.pdf`.
31. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
32. E. Di Nitto and D. Rosenblum. Exploiting ADLs to specify architectural styles induced by middleware infrastructures. In *Proc. of the 21st International Conference on Software Engineering, ICSE 99*, pages 13–22. IEEE Computer Society Press, 1999.
33. Object Management Group. *UML 2.0 OCL Final Adopted Specification*, 2003. `www.omg.org/cgi-bin/doc?ptc/2003-10-14`.
34. Object Management Group. *UML 2.0 Superstructure Final Adopted specification*, 2003. `http://www.omg.org/cgi-bin/doc?ptc/2003-08-02`.
35. Object Management Group. *XMI: XML Metadata Interchange, v2.0*, 2003. `http://www.omg.org/cgi-bin/doc?formal/2003-05-02`.
36. A. Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl, J. Pfalz, and B. Böhlen, editors, *Proc. Application of Graph Transformations with Industrial Relevance (AGTIVE '03)*, volume 3062 of *LNCS*, pages 479–485. Springer, 2003.
37. A. Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *LNCS*, pages 401–415. Springer, 2004.
38. A. Rensink, Á. Schmidt, and D. Varró. Model checking graph transformations: A comparison of two approaches. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd International Conference on Graph Transformation, ICGT 2004*, volume 3256 of *LNCS*, pages 226–241. Springer, 2004.
39. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating architecture description languages with a standard design method. In *Proc. of the 20 th International Conference on Software Engineering, ICSE 98*, pages 209–218. IEEE Computer Society, 1998.
40. T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):13 pages, 2001.
41. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In 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.
42. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.
43. 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.
44. 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.
45. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, 2004.
46. M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.
47. A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL graph exchange language. In S. Diehl, editor, *Software Visualization: International Seminar, Dagstuhl Castle, Germany, May 20-25, 2001. Revised Papers*, volume 2269 of *LNCS*, pages 324–336. Springer, 2002.