Modeling and Validation of Service-Oriented Architectures: Application vs. Style [*]

Luciano Baresi[*] baresi@elet.polimi.it
Reiko Heckel[†] reiko@upb.de
Sebastian Thöne[‡] seb@upb.de
Dániel Varró[§] varro@mit.bme.hu

ABSTRACT

Most applications developed today rely on a given *middleware platform* which governs the interaction between components, the access to resources, etc. To decide, which platform is suitable for a given application (or more generally, to understand the interaction between application and platform), we propose UML models of both the *architectural style of the platform* and the application scenario. Based on a formal interpretation of these as graphs and *graph transformation systems*, we are able to validate the *consistency between platform and application.*

The approach is exemplified by means of an application scenario from a supply chain management case study using the service-oriented architectural style. In particular, we demonstrate the potential of *model checking* for graph transformation systems for answering the above consistency question.

1. INTRODUCTION

Nowadays, applications have to be adaptable to changes in (at least) two dimensions: *Changing requirements*, like requests for new functions or services, may require flexible business collaborations where components are integrated at run-time. *Changing contexts*, like faulty communication channels or mobility leading to a reduced bandwidth may require to replace unreachable components.

Component models like CORBA, EJB, or Web Services provide the basic techniques to realize the required flexibility through reconfiguration mechanisms, like dynamic lookup, loading, and binding of components. However, the costs and the level of support provided may differ between the platforms, as well as the precise rules of interaction between components. Choosing the right platform for a given application is therefore a mission-critical question, especially as later migration may be costly, if not impossible.

In order to reduce this risk, it is common to produce a prototype implementation covering all critical application scenarios and operations, like database access, user interaction, communication with remote components, etc. In the Unified Process [17], for example, this would happen in the Elaboration phase with the objective to produce a stable architecture of the system.

Following a model-based development approach, application scenarios are expressed at a conceptual level, for example, in terms of UML [23] use case and sequence diagrams. This representation allows to reason about requirements at a high level of abstraction, without considering implementation-specific details.

Precise information on the properties and limitations of a given platform, on the other hand, is often only available at the level of framework APIs or implementation-oriented infrastructure specification documents. What is largely missing is a conceptual model of the infrastructure which would allow an understanding of the basic mechanisms, and their suitability for a certain task, at a non-technical level.

Besides a general lack of understanding of the infrastructure level among managers and customers, this means that in order to judge the realizability of a given application scenario, we have to go right to the implementation level. If the infrastructure at hand is not the one that is finally chosen, much of this is wasted effort devoted to technical details that might not even be relevant for the decision to be made.

To solve this problem, we propose to model the architectural elements supported by a middleware platform along with the associated constraints and reconfiguration rules as an *architectural style.* Based on such a conceptual platform model, we can try to answer questions like

- *Is a desired configuration reachable from an initial configuration?*
- *Is an application scenario realizable on the platform?*
- *Which sequence of operations is required to reach the*

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

[*]Politecnico di Milano – Dipartimento di Elettronica e Informazione, piazza L. da Vinci, 32 I-20133 Milano (Italy)

[†]University of Paderborn – Department of Computer Science, Warburger Str. 100, D-33098 Paderborn (Germany)

[‡]University of Paderborn – International Graduate School Dynamic Intelligent Systems, Warburger Str. 100, D-33098 Paderborn (Germany)

[§]Budapest University of Technology and Economics – Department of Measurement and Information Systems, Magyar tudósok körútja 2, H-1521 Budapest (Hungary)

desired configuration / realize the scenario?

We have argued in [3] that, in order to answer such questions, an architectural model is required which allows to reason at an abstract level, disregarding the technicalities of the component model employed. Still, this abstraction must not lead to ambiguity, as reasoning on complex problems requires a high degree of precision. This is usually provided by formal methods-based architectural description languages like Wright [2] or Rapide [19].

On the other hand, some of the questions above require knowledge of the problem domain. Therefore, the model needs to be understood and validated by domain experts with little or no background in formal specification. Here, an explicit, visual representation of the architecture in some diagrammatic language like the Unified Modeling Language (UML) [23] is often regarded as helpful, even if we risk to trade this intuitive nature for ambiguity.

At the same time, both UML-based architectural models and architectural description languages fail to describe the highly dynamic nature of today's architectures, with unbounded creation and deletion of components and connectors.

Based on this observation, we propose a combination of UML modeling and graph transformation as a visual, yet formal approach to model (and reason about) component-based architectures. In particular, we use transformation rules to specify architectural reconfiguration, but also possible changes in the environment, by graphical pre/post conditions.

Since these models are executable, they support *automated reasoning* by means of *simulation* using graph transformation tools like PROGRES [24] or Fujaba [1], e.g., in order to check the applicability of a certain sequence of basic reconfiguration steps in a given situation. Moreover, the theory of graph transformation provides the basis for *static analysis*, like the computation of critical pairs to detect conflicts or causal dependencies [5], or *model checking* [27] to answer questions about the reachability of configurations.

In this paper, we present an application of these ideas to service-oriented architectures (SoAs) which are typical for their dynamic nature, given the run-time detection of components through registry services and subsequent dynamic binding. Our model for service-oriented architectures is introduced in Section 2. Section 3 presents our application scenario, a case study on supply chain management proposed in [8]. Based on the two models, Section 4 describes the use of model checking and simulation to validate the consistency between the style and the application. Section 5 surveys the related work while Section 6 concludes the paper discussing possible future work.

2. STYLE: SERVICE-ORIENTED ARCHITECTURES

This section presents our proposal for modeling architectural styles with UML diagrams and graph transformation rules and exemplifies it for service-oriented architectures. Nevertheless, the techniques are applicable to other platforms and architectural styles as well.

An architectural style includes a static and a dynamic specification. The static part, described in Section 2.1, defines the set of possible components and connectors and constrains the way in which these elements can be linked together. The dynamic part, described in Section 2.2, specifies how a given architecture can evolve in reaction to planned reconfigurations or unanticipated changes of the environment.

As shown in Fig. 1, taken from [7], service-oriented architectures involve three different kinds of actors: service providers, service requesters and discovery agencies. The service provider exposes some software functionality as a service to its clients. Such a service could, e.g., be a SOAP-based web service for electronic business collaborations over the Internet. In order to allow clients to access the service, the provider also has to publish a description of the service. Since service provider and service requester usually do not know each other in advance, the service descriptions are published via specialized discovery agencies. They can categorize the service descriptions and provide them in response to a query issued by one of the service requesters. As soon as the service requester finds a suitable service description for its requirements at the agency, it can start interacting with the provider and using the service.

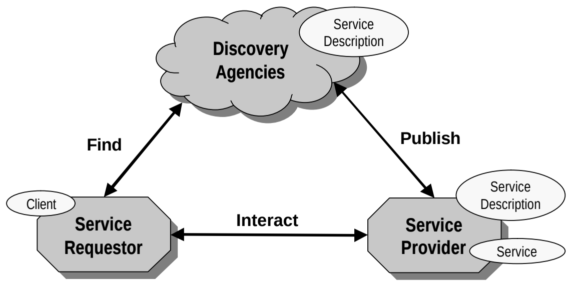

Figure 1: Service-oriented architecture

Such service-oriented architectures are typically highly dynamic and flexible because the services are only loosely coupled and clients often replace services at run-time. Firstly, this is advantageous if the new service provides a better alternative to the former one concerning functionality or quality of service. Secondly, this kind of reconfiguration might become necessary if a service is not reachable any longer because of network problems.

If a requester wants to connect to a new service but requires a certain level of quality from this service, it is imaginable that these quality properties can only be guaranteed under certain assumptions. This means that the quality or functionality of the provided service might depend on other third-party services used by the service itself. For this reason, the service provider might have to find suitable sub services on its own before it is able to confirm a request for a certain functionality or level of quality.

2.1 Static model

We model the static part of the architectural style with *UML class diagrams* [23]. The provided classes represent three different kinds of elements: *structural elements* like components and services, *specification documents* for describing services and requirements, and *messages* for modeling communication. Consequently, they are grouped into the three packages Structure (see Fig. 2), Specification (Fig. 3), and Messages (Fig. 4). Associations define how the elements can be linked in a concrete architecture, constrained by the given cardinalities[1].

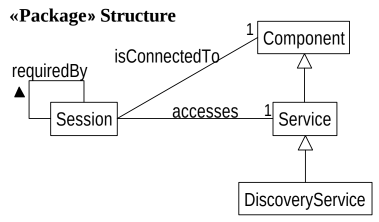

Figure 2: Package for structural elements

The package Structure contains the classes that form the core of the architectural style. In the case of service-oriented architectures, we define a Service as a special Component which exposes its functionality to other components and services. In the sense of an architectural connector, a Session is used to connect a Component to a Service. It stores the current state of the interaction between the service requester and the service. Note that, since Service is a subclass of Component, a Service can connect to another Service via a Session, too. In a service-oriented architecture, discovery agencies provide DiscoveryServices, which are special services for querying a service catalog.

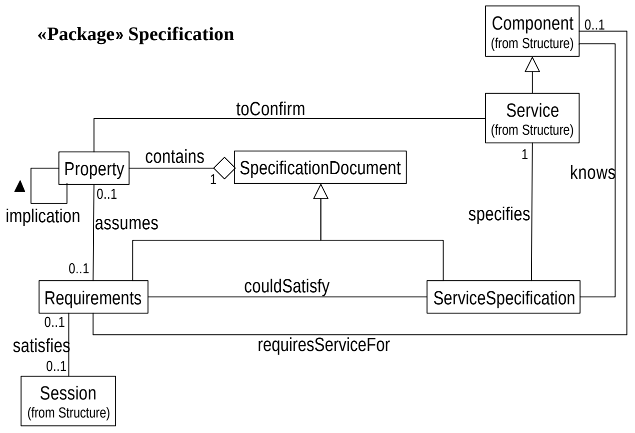

Figure 3: Package for specification documents

The package Specification adds specification documents to the static model. They are necessary to describe reconfiguration operations which dynamically search a component for certain requirements at run-time. The package defines two types of specification documents: Requirements and ServiceSpecifications. Both of them contain a set of Properties as inherited from the super class SpecificationDocument. In the case of a Requirements document, these properties are required by a Component for the service it wants to use. In the case of a ServiceSpecification describing a particular Service, these properties are guaranteed by the service provider. In some cases this might only be possible under certain assumptions which are modeled as a link from the respective Property to further Requirements. After a Session has been successfully established for the Requirements, this is marked by a satisfies link to this Session.

The self-related implication association of the Property class indicates when a property logically implies another one. This relation is used to correctly match requirements and service specifications. In order to ensure the intended semantics of the association, we add the following OCL [23] constraint to the package:

```
context Property inv:
self.implication->forAll(p | self.expression
                       implies p.expression)
```

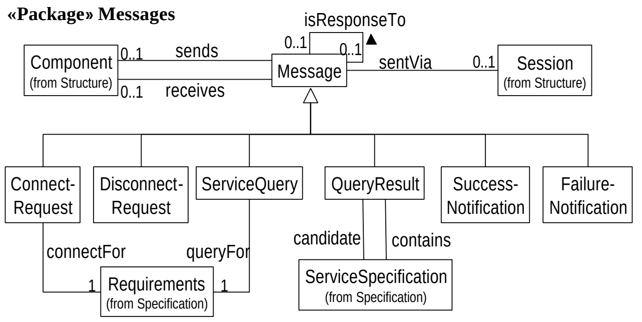

Figure 4: Package for messages

Since the components and services are loosely coupled in a service-oriented architecture, they interact by exchanging messages. For this reason, the package Messages provides the necessary classes for the communication. Therein, a Message is sent and received by Components or, because of subtyping, by Services. Messages which are dedicated to reconfiguration purposes are defined as sub classes of Message. For instance, a ConnectRequest message triggers the creation of a new Session for particular Requirements. Except from ConnectRequest, all messages are sent via a valid Session.

An architecture compliant with the style can be regarded as an instantiation of the class model. This is exemplified in Sec. 3. There, we present a simplified supply-chain management application that conforms to this service-oriented architectural style. The following section extends the style by adding a dynamic model which contains rules for the specification of possible architectural reconfigurations.

2.2 Dynamic model

In order to reason about planned or unanticipated reconfigurations of architectures, we use graph transformation rules to capture the dynamic aspects of the architectural style. There are two different ways of visualizing a graph transformation rule. The first way is to depict a rule as a pair of two instance graphs with the left-hand-side defining the pre-conditions and the right-hand-side defining the post-conditions of the transformation. Both graphs represent a part of the configuration as an instance of the architectural style defined in Section 2.1.

Figure 5 shows, as a first example, the rule sendConnectRequest in which a Component, playing the role of a service requestor, sends a request for connection to the Service it would like to connect to. As precondition the requestor has to know a ServiceSpecification which couldSatisfy its Requirements. As postcondition the request message is created and linked to the receiver.

For conciseness, we propose as an alternative way of visualization to include both pre and post conditions into one *UML collaboration diagram* [23]. Elements which are added

[1]No explicit cardinality means 0..n by default.

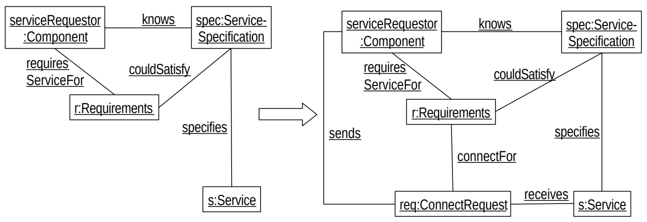

Figure 5: Transformation rule as pair of graphs

to the configuration graph by the rule are then tagged with the label {new}, and elements which are deleted with the label {destroyed}. Figure 6 shows the previous rule with the new notation.

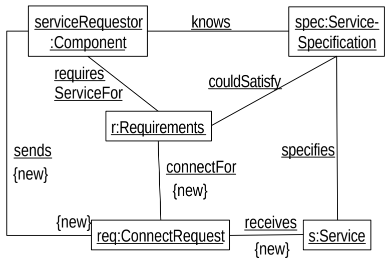

Figure 6: Condensed notation for sendConnectRequest

Because of space limitations we omit the rules which deal with the just created request and summarize them textually: At first, the requested service has to prepare a Session and to confirm all required properties. For this reason, the properties are initially marked with a toConfirm link between each Property and the Service. Whenever a property can be confirmed, e.g., by comparison with the ServiceSpecification, the toConfirm link for this property can be deleted.

Finally, when all toConfirm links have been successfully removed, the Service can respond to the request by sending a SuccessNotification as shown in Fig. 7. This rule contains a negative application condition which prevents its application if there is any Property in the requirements which still has to be confirmed by the service.

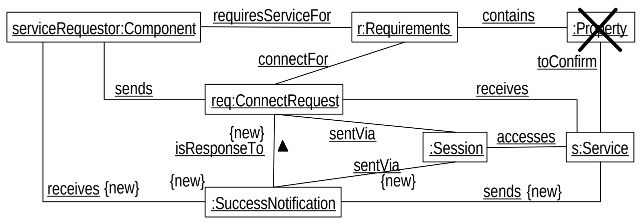

Figure 7: Rule with negative application condition

After the reception of the notification message, the rule connectToSession given in Fig. 8 can be applied to establish the link between the serviceRequestor and the new Session. In parallel, the request and notification messages are deleted, since they are not used any more.

Altogether, the dynamic model contains about 20 transformation rules which cover publishing a service description to a discovery service, querying the service catalog of a discovery service, connecting to a known service, interacting

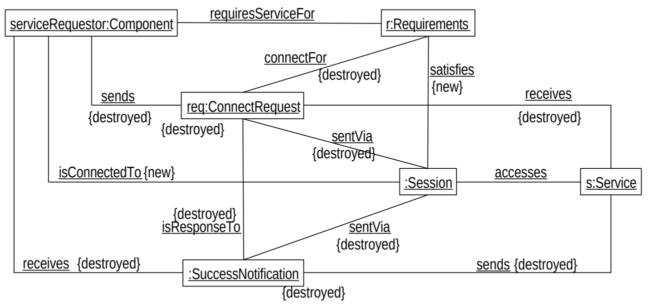

Figure 8: Transformation rule connectToSession

with the service by exchanging messages, and disconnecting from an existing session. If a more complex reconfiguration step requires a sequence of individual transformation rules, these rules could be combined using explicit control flow constructs. For instance, story diagrams [11] combine graph rewriting rules based on UML collaboration diagrams with control flow elements as provided by UML activity diagrams.

After the model of an architectural style has been completed including the dynamic part, it can be used to decide whether or not the modeled style is suitable for the requirements of a particular application. As an example, the following section presents a sample application whose architecture could follow the service-oriented style.

3. APPLICATION: SUPPLY-CHAIN MANAGEMENT

In this section, we want to apply the architectural style to a concrete application scenario. For this purpose, we choose the reference architecture for a supply chain management system as proposed in [8] by the Web Services Interoperability Organization. The scenario involves a consumer component, a retailer service, a warehouse service, a shipping service (this is an extension to the original example as given in [8]), and a manufacturer service (see Fig. 9).

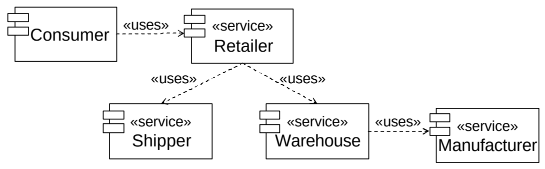

Figure 9: Services involved in the scenario

A typical scenario of this application is given by the sequence diagram in Fig. 10. After the ConsumerUI has received a product catalog from RetailerA, it can submit an order for certain products by sending a submitOrderRequest. Then the retailer service connects to its WarehouseA service and inquires if the product is available from the warehouse. After this has been confirmed, the retailer service orders the shipping service to ship the goods. Meanwhile, the warehouse might start to submit a purchase order to ManufacturerX, because the stock level of the sold product has fallen below a certain limit.

There are several imaginable requirements which make this scenario highly dynamic in terms of service connections:

- The consumer intends to choose the cheapest retailer.

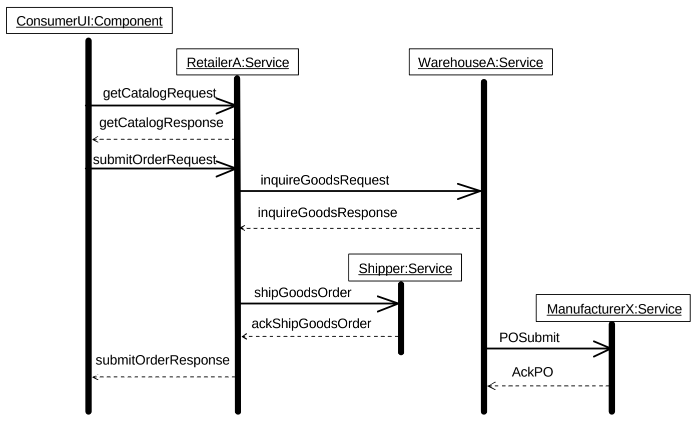

Figure 10: A supply chain interaction scenario

- The same holds for the warehouse looking for a cheap manufacturer.
- The retailer might run several warehouses and has to find the right one for the requested product and location.
- The consumer might impose certain time limits for the product delivery which then affect the choice of the shipping service.

To meet these requirements, most of the services are to be discovered at run-time. Therefore, the sample application is a good candidate to be realized in the above described service-oriented architectural style. The model of the architectural style supports the architect during the decision whether the style is suitable for his application. He could, e.g., specify a suitable initial configuration of the application following the given architectural style. Then, the dynamic model of the style is used to reason about possible reconfigurations and to check if all required configurations of the application are reachable. Figure 11 shows how an initial configuration for the supply chain example might look like based on the service-oriented architectural style described in Section 2.

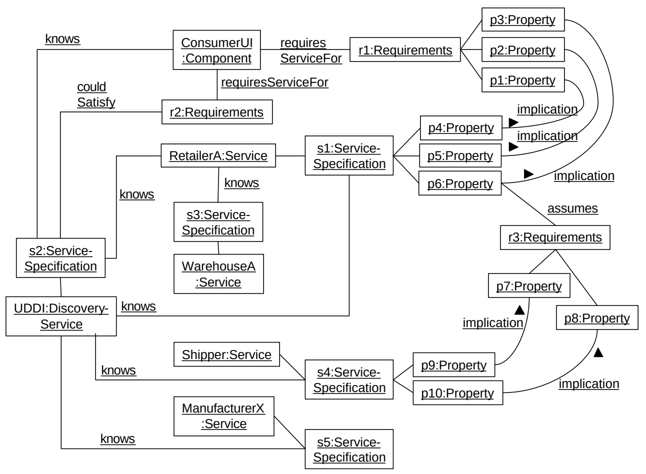

Figure 11: An initial architectural configuration

p1	Service provides product catalog
p2	Service accepts purchase orders
p3	Maximum shipping time ≤ 72hrs
p4	Service provides product catalog
p5	Service accepts purchase orders
p6	Maximum shipping time ≤ 60hrs
p7	Service ships goods from A to B
p8	Maximum shipping time ≤ 60hrs
p9	Service ships goods from A to B
p10	Maximum shipping time ≤ 60hrs

Table 1: Properties

The configuration is depicted as a graph which represents a valid instantiation of the static model of the architectural style. It represents the initial architecture of the system in a very abstract syntax. For better understandability, we are working on a more convenient, concrete syntax with distinct graphical symbols for the different elements.

In this case, the configuration comprises the involved services of the supply chain including their specifications and requirements. A DiscoveryService is added which knows all the relevant ServiceSpecifications and can therefore serve as a discovery agency for the participants of the supply chain.

For conciseness, the diagram shows only part of the properties contained in the specifications and requirements. At this level of abstraction, we specify the properties informally as indicated by Tab. 1. If one property implies the other, this is shown as an implication link in the diagram. In order to compute these implications dynamically at run time, a formalism for describing and reasoning about the properties would be required. The lack of agreement on such a formalism is one of the weaknesses of current implementations of SoA, like web services, because this restricts the capabilities for dynamic binding.

The following section discusses possible analysis techniques which provide suitable support for validating the architectural style in combination with a given application and its initial configuration.

4. ANALYSIS

When using the SoA style in the supply chain management application (in general, using an architectural style in a certain application) one must ensure that the style is used consistently by the application. In the current section, we analyze the consistency of business communication scenarios captured in the form of UML sequence diagrams (see Fig. 10) with respect to the dynamic behavior of the SoA style (see rules in Sec. 2.2) by model checking techniques.

For that purpose, we encode the specification of the architectural style (consisting of the metamodel of the style, a set of graph transformation rules capturing the dynamic behavior and the model instance for the business application) into a state transition system (following the guidelines of [27,28]). Moreover, we formalize business application scenarios as (a set of) reachability properties, which is proved automatically by the model checker Murϕ by using assume-guarantee reasoning. Finally, the results obtained from the model checker can be further validated using simulation.

4.1 Transition systems

Transition systems (or state transition systems) are a common mathematical formalism that serves as the input specification of various model checker tools. They have certain commonalities (in many cases on the concrete language level as well) with structured programming languages (like C or Pascal) as the system/program is evolving by executing non-deterministic if-then-else like rules that manipulate state variables. In all practical cases, we must restrict the state variables to have finite domains, since model checkers typically traverse the entire state space of the system to decide whether a certain property is satisfied.

Formally, a *transition system* $TS = (V, Dom, T, Init)$ is a 4-tuple where (i) $V = \{v_1, ...v_k\}$ is the set of *state variables* (ii) taking their values from the corresponding finite *domains* $Dom = \{dom_1, ...dom_i\}$ (iii) $T = \{\tau_1, ..., \tau_n\}$ is the set of *transitions (guarded commands)* which is of the form $guard \longrightarrow v'_1 := e_1, ..., v'_n := e_n$ (where the $guard$ is a boolean condition and an action $v'_1 := e_1$ specifies new assignments (updates) for state variable) inducing a *transition relation* $act_\tau(V, V')$ defined as $guard \wedge \bigwedge_{i=1}^{k} v'_i = e_i$; while (iv) $Init$ is a predicate defining the *initial state*.

For our convenience, we suppose that state variables can be stored in state variable arrays ranging on the set of object identifiers, and they can be referred as $v_j[i]$. In the paper, the Murϕ notation is used as the concrete representation of transition systems due to its rather self-explanatory syntax.

The requirements (or properties to be verified) for models specified by a transition system are frequently captured by some temporal logic formulae. However, since only reachability properties are being proved in the current paper, we define these concepts without the use of temporal logic operations. In fact, a *reachability property* can be interpreted as a special transition in the transition system that immediately interrupts the model checking process if its guard expression, which refers to the class of states to be checked for reachability, is ever satisfied.

Given (i) a system model in the form of a transition system TS (with semantics defined as a Kripke structure), and (ii) a reachability property ϕ, the *model checking problem* can be defined as to decide whether the special transition for the reachability property is fireable on at least one execution path of the system (i.e., whether $TS \models \phi$).

In the sequel, model checking is enabled for graph transformation systems by automatically translating them into transitions systems. The main challenge in such a translation is that a naive encoding of the graph representation of application models would easily explode both the state space and the number of transitions in the transition system even for simple models, therefore sophisticated optimization techniques are required.

4.2 Declaring state variables

From a verification point of view, the state space of the supply chain management application is constituted from the different instance configurations of components and services. As such configurations are handled formally as directed and attributed graphs, the encoding techniques of [27, 28] are applicable to drastically reduce this state space by introducing state variable arrays only for dynamic model elements.

A model element (object, link or attribute) is considered to be dynamic if there is at least one rule that potentially modifies (creates, destroys, updates) the element. For instance, a ConnectRequest object is a dynamic element as it can be created by rule sendConnectRequest, on the other hand, the assumes links are static as no rules are provided to modify them.

The encoding of dynamic model elements into state variables is driven by the metamodel. We define

- a *one-dimensional boolean state variable array* (a unary relation symbol) for each dynamic class (such as ConnectRequest, Session, and ServiceQuery in our running example);
- a *two-dimensional boolean state variable array* (a binary relation symbol) for each association (e.g., requiresServiceFor, sends, etc.);
- a *one-dimensional state variable array with enumeration range* for each attribute (no dynamic attributes in the model this time).

For model checking purposes, we must restrict the dimension of each array and all the enumeration types to be finite during type declaration. For the corresponding graph transformation system, this restriction implies that there exists an *a priori* upper bound for the number of objects in the model for each class. In this respect, we suppose that when a new object is to be created it is only activated from the bounded "pool" of currently passive objects (deletion means passivation, naturally), and the same applies to the interpretation of links.

As a consequence, the transition system corresponding to our supply chain management model contains the following piece of Murϕ code to declare state variable arrays and their domains for the model (we only provide a partial encoding of the model due to space limitations).

```
-- Declaring domains for state variable arrays
comp_dom : enum {UDDI, RetailerA, WarehouseA,
                 Shipper, ManufactX, ManufactY,
                 ConsumerUI};
                 -- constrained by static parts
reqr_dom : enum {r1, r2, r3, r4};
reqs_dom : 1..10; -- constrained by an explicit UB
-- Declaring state variable arrays themselves
ConnectRequest :     array [reqs_dom] of boolean;
requiresServiceFor : array [comp_dom] of
                     array [reqr_dom] of boolean;
```

Note that while the domain (i.e., the potentially active instances) of components (comp_dom) and requirements (reqr_dom) are constrained by the static parts of the model, the constraint imposed on the domain of requests (reqs_dom) only provides an a priori upper bound (which can be increased at compile-time if we run out of request objects during model checking).

One can notice that request is the state variable array introduced for the dynamic class ConnectRequest, while requiresServiceFor is the state variable array of the corresponding association in the metamodel.

4.3 Encoding initialization

In general terms, the initial configuration of the application model is projected into the initial state of the transition system. In this respect, exactly those locations of state variable arrays evaluate to true in the initial state for which the related model elements are existent in the initial configuration.

The initialization of the transition system generated for the application model depicted in Fig. 11 is given below.

```
startstate
begin
 for x : reqs_dom  do
  ConnectRequest[x] := false;
 endfor;
 for x : comp_dom do
  for y : reqr_dom do
   if (x = ConsumerUI & y = r1) |
      (x = ConsumerUI & y = r2)
   then requiresServiceFor[x][y] := true;
   else requiresServiceFor[x][y] := false;
   endif;
  endfor;
 endfor;
end;
```

Note that as no request objects are present in the initial configuration, all locations in the state variable array ConnectRequest are initialized to false. On the other hand, initially, ConsumerUI requires service for Requirements r1 and r3.

4.4 Encoding the rules

Potential applications of the graph transformation rules that specify the dynamic behavior of the style are encoded into transitions (guarded commands) of the corresponding transition system.

Since the encoding only introduces state variables for dynamic model elements, we also have to eliminate conditions that refer to the static parts of the model. For that reason, the generation process of transitions is driven by a graph pattern matching engine, which collects all the matching instances of the static parts of the preconditions of a rule. If the guard of a certain guarded command can never be satisfied due to the failure of pattern matching in the static structure then this transition is not generated at all in the target transition system.

Although this compile-time preprocessing can be time-consuming, since all the potential matches of a rule have to be encountered, we only have to traverse a relatively small part of the state space for this step as graph transformation rules define local modifications to the system state thus it is typically negligible when compared with the time required for traversing the entire state space during model checking.

For instance, we would generate (amongst many others) the following transition as a potential application of rule sendConnectRequest.

```
rule "sendConnectRequest[sr=ConsumerUI, s=uddi,
                          r=r2,sp=s2]"
  knows[ConsumerUI][s2] &
  couldSatisfy[s2][r2] &
  requiresServiceFor[ConsumerUI][r2] &
 ==>
 begin
  ConnectRequest[1] := true;
  receives[1][uddi] := true;
  sends[ConsumerUI][1] := true;
  connectFor[1][r2] := true;
 end;
end;
```

Note again that only dynamic concepts are contained by transitions. Therefore, the guard of the transition corresponding to the precondition of the rule is reduced to check for the existence of dynamic links knows, couldSatisfy and requiresServiceFor in the associated locations of state variable arrays.

As a result, the action of the transition activates (creates) the request object 1 with the corresponding links receives, sends and connectFor. [2]

For the current case study, a manual generation of the target transition system (strictly following the guidelines of the translation algorithm presented in [28]) was feasible due to the relatively small size of the application model. However, we are in an advanced phase in building a tool that implements the translation algorithm to provide automation for this step.

Additionally, rules containing multi objects were transformed into separate rules without multi objects (but with additional control conditions encoded as attributes) in order to fit into the translation approach. Alternatively, one could easily extend the original translation approach to generate the corresponding set of transitions for multiobjects as well by enumerating all the possible for matchings.

4.5 Reasoning on scenarios

The main task of our proposed analysis framework is to prove that a business scenario captured in the form of UML sequence diagrams (like the one in Fig. 10 for supply chain management) is consistent with the dynamic semantics of the architectural style.

Our proposal is to check the reachability of consecutive configurations obtained from the sequence diagrams by horizontal cuts after abstracting from application specific details of business messages. The basic idea is demonstrated on the concrete example of Fig. 12.

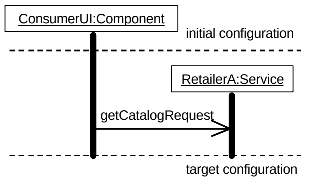

Figure 12: Slicing sequence diagrams

Here we are interested in showing that starting from the initial configuration (defined by Fig. 11) component ConsumerUI is able to connect to the service RetailerA and issue a catalog request (see message getCatalogRequest) afterwards by applying the graph transformation rules of the style in a proper sequence.

However, as the application-specific contents of business messages are hidden for an architectural analysis, we do not know, for instance, how the elementary operations (i.e., graph transformation rules) provided by the style to submit a request are actually distributed between business messages.

[2]Note that a more sophisticated solution is applied in the original encoding [28] for handling the creation of dynamic objects.

But even analyzing a scenario at such a high level of abstraction, one can ask whether the application configuration obtained after sending business message getCatalogRequest is reachable from the initial configuration. In our concrete example, such a *target configuration* can be an existing session object between component ConsumerUI and service provider RetailerA. This reachability property is encoded as a special error transition in Murϕ as follows.

```
rule "reachability property"
  exists s: sess_dom do
   session[s] &
   interactsWith[s][ConsumerUI] &
   accesses[s][RetailerA]
  endexists
 ==>
  error "A session is established between
         ConsumerUI and RetailerA"
end;
```

Note that error is a keyword in Murϕ to prescribe that the model checking process should be terminated if the transition is fired. In case of proving reachability properties, this is a desired situation.

Murϕ automatically found an "error" trace consisting of 37 rule applications that proves the reachability of this target configuration in (an average) 14.2 seconds by a depth-first search (running on a 550 MHz Pentium III machine with a limit of 16M of system memory).

After that, the rest of the scenario can be verified by a kind of *assume-guarantee reasoning* in order to reduce the computational complexity of individual verification steps.

1. We identify relevant intermediate configurations (after each message sent for a worst case analysis) by slicing the sequence diagram of the business scenario.
2. First we prove that the target configuration defined by the first cut is reachable from the initial configuration by model checking.
3. Supposing then that the system is in the configuration reached at the first cut, we prove afterward that the configuration at the second cut is reachable as well (and so forth for additional neighboring cuts).
4. As a result, individual verification steps can be carried out independently from each other.

Finally, we would like to emphasize that unsuccessful preliminary model checking attempts were also useful from a validation point of view, as they revealed several unexpected side effects of graph transformation rules. Resolving these problems required to refine the rules with additional (typically negative) application conditions.

4.6 Simulation

The model checking analysis can be complemented by simulation. Due to the state explosion problem and the resulting limitation of the state space, a model checker cannot take into account the internal state of the involved components and services. Thus, it could happen that, although the model checker returns a certain sequence of architectural reconfigurations steps in order to prove the reachability of a given configuration, this sequence is not feasible according to the specific requirements of the actual application.

But, nevertheless, the resulting trace of operations can be validated by simulation taking the model and the sequence of operations as input and executing (at the model level) the individual reconfigurations. Since such simulations do not have to cope with the state explosion problem, the specific application information and the internal state of the components can now be added to the model.

Simulation requires a tool for executing graph transformation rules. The object-oriented modeling tool Fujaba [1] suits this purpose very well, because it combines an editor for UML class diagrams and graph rewriting rules with a dynamic object browser that visualizes the effect of the rewriting rules on a given model instantiation.

From the trace returned by the model checker, one can derive a *simulation driver* which automatically instantiates the initial configuration (see Fig. 11) and applies the given trace of transformation rules to this configuration. With such a driver the simulation is easier to handle and to repeat after some changes of the model. This is comparable to a test driver, only that we are not testing an application, but only simulating the execution at the model level. In a similar way, the model checker could be used for the generation of test cases for the actual implementation.

5. RELATED WORK

The work presented in this paper has been influenced by several different proposals. First of all, we must say that the idea of working on modeling and analyzing software architectures came from the many ADLs (Architectural Description Language) like Rapide [19], Wright [2], Darwin [20], C2 [26], and xADL [9]. In all these approaches, we noticed a mismatch between the abstraction level at which we usually model the software architectures of our systems and the abstraction level offered by these languages. Our opinion is that, while the semantics of concepts is clear and well-defined, the concrete representation does not always offer "usable" means to reason on complex architectures.

Our proposal overcomes the problem by pairing a well-known notation, like UML, to represent concepts and supply a usable means, and graph transformation to formally specify the way these elements can be composed and interact. Also Medvidovic et al. [21] use UML to suitably render software architectures. Their proposal emphasizes the definition of suitable stereotypes to represent the peculiar concepts, but does not consider the problem of analyzing and validating designed models.

Even more attention to the representation of concepts is payed by Garlan et al. in [12], where they propose several different alternatives for modeling architectural structures using UML. They present a systematic approach to the use of UML and describe the strength and weaknesses of adopting a particular modeling strategy. The main goal of our work is not on rendering architectural concepts with UML, thus we can say that this work can be seen as a complement to our proposal to better address the concrete syntaxes offered to users.

The formal foundation and analysis capabilities of our approach come from both graph transformation as a means to reason on software architectures and model-checking as a way to assess their quality. In this context, we must mention the CHAM approach [16], in which architectural reconfiguration is studied in terms of molecules and reactions, and the proposals that represent architectural styles by means of graph grammars [15,18,25,29] and reason on changes and evolution with respect to structural constraints.

Some of these approaches, use a graph grammar to specify the class of admissible configurations of the style. Our proposal utilizes a static model – class diagrams along with constraints – to identify valid instances of a given style. Graph transformation rules model only the dynamic aspects like evolution and reconfiguration. The advantage is that a declarative specification is more abstract and easier to understand, even if constructive/operational ones are better for analysis and tools.

In fact, the use of model checking techniques for graph transformation seems to be original. In the area of software architectures, Muccini, in his Ph.D. thesis [22], proposes a way to transform architectural descriptions into suitable SPIN specifications. The three-step approach starts with transforming architectural models, which describe the behavior of components, into a Promela specification. This specification can be enriched by adding information on the communication type that can be obtained from the scenarios associated with the architecture. In the second step, scenarios are translated into LTL formulae and – as third step – he uses SPIN to check if the LTL formulae are verified on the obtained global model. The main difference is that he models behavioral aspects through state-based machines and scenarios which are not directly able to model dynamic reconfiguration, but only the communication among components. In our case, scenarios are only used to encode reachability properties while graph transformation rules specify the reconfiguration explicitly.

Self-adaptive and self-healing systems [13] is the last domain with which we think that we should compare our work. For example, Georgiadis et al. [14] model structural architectural styles for these systems by means of Alloy. The model is neat and elegant, but again we think that a clear and separate representation of how the system can adapt or repair itself is important to make designers fully understand how their models can evolve and try to identify – through suitable analysis – possible problems.

6. CONCLUSION

The paper presents an approach to modeling and analyzing software architectures based on modeling the architectural style through class diagrams – along with constraints – and the dynamic behavior using a graph transformation system. The whole approach is instantiated on service-oriented architectures to better exemplify presented concepts. Besides modeling the style, we exemplify our approach by means of a simple *supply chain management system.* This is intended to both show how a particular architecture should look like and to exemplify the analysis capabilities that we have associated with our approach. The model of the style and configurations specific to the application are encoded in a way suitable to the Murphϕ model checker. Scenarios, modeled as sequence diagrams, become reachability properties on which we want to validate the model.

The first results on applying the approach are giving encouraging results, but our current work is mainly devoted at refining our proposal. The elements on which we are working are:

- Different solutions – besides those presented in the paper – for expressing rules, constraints, etc. to find the right balance between expressiveness and analyzability. This includes alternative control mechanisms for graph transformation, like priorities or explicitly programmed control.

- Extensions to address adaptability and the capability of automatic recovery [13]. Rules offer a clean and neat way to specify how the architecture should react to the different stimuli, and the analysis capabilities allow the designer to predict the behavior of specified architectures.

- Implementation issues. The graph transformation system can be seen as the coordinator that supervises the adaptation process. In this context we do not want to discuss all related problems, but rather we want to identify the possibility of using the same technology both as modeling means and as run-time supervisor.

- Derivation of suitable test cases. We would like to use the graph transformation system to derive both suitable test cases and model-based oracles to assess the quality of test results. The two aspects can be tackled independently: Test case generation for architectures is nothing new (see for example [4]), the novelty is the rule-based derivation. To the best of authors' knowledge, there are no proposals to derive test cases from graph transformation systems, but grammar-based test case generation has been already proposed if we consider pure textual grammars ([6]).

- Tool support. Any methodology must be suitably supported by tools. In this case, we are planning to extend some open-source environments like *eclipse*, along with its UML add-ins, to model software architectures and directly integrate with Murpϕ or another model-checker.

7. REFERENCES

[1] From UML to Java and Back Again: The Fujaba homepage. `www.fujaba.de`.

[2] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997.

[3] L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modelling and analysis of architectural styles based on graph transformation. In *Proc. 6th ICSE Workshop on Component-Based Software Engineering (CBSE6): Automated Reasoning and Prediction*, 2003. To appear.

[4] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving test plans from architectural descriptions. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 220–229. ACM Press, June 2000.

[5] P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rom.

[6] A. Celentano, S. Crespi Reghizzi, P.L. Della Vigna, C. Ghezzi, and F. Savoretti. Compiler testing using a sentence generator. *Software — Practice & Experience*, 10:897–918, 1980.

[7] 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/`.

[8] M. Chapman, M. Goodner, B. Lund, B. McKee, and R. Rekasius. *Sample Application Supply Chain Management Architecture*. Web Services Interoperability Organization, 2002. `http://www.ws-i.org/SampleApplications/SupplyChainManagement/2002-11/SC%MArchitecture-0-11-WGD.pdf`.

[9] E.M. Dashofy, A. van der Hoek, and R.N. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 266–276, New York, May 19–25 2002. ACM Press.

[10] 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.

[11] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the unified modeling language. In G. Engels and G. Rozenberg, editors, *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

[12] D. Garlan, S.-W. Cheng, and A. J. Kompanek. Reconciling The Needs of Architectural Description with Object-Modeling Notations. *Science of Computer Programming*, 44(1):23–49, July 2002.

[13] D. Garlan, J. Kramer, and A.L. Wolf, editors. *Workshop on Self-Healing Systems*, 2002.

[14] I. Georgiadis, J. Magee, and J. Kramer. Self-Organising Software Architectures for Distributed Systems. In *Proceedings of WOSS'02: Workshop on Self-Healing Systems*, pages 33–38, 2002.

[15] D. Hirsch and M. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. CONCUR 2001, Aarhus, Denmark*, volume 2154 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, August 2001.

[16] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[17] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.

[18] Le Métayer, D. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23, New York, October 16–18 1996. ACM Press.

[19] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[20] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings the 5th European Software Engineering Conference*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153. Springer-Verlag, September 1995.

[21] 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*, 11(1):2–57, January 2002.

[22] H. Muccini. *Software Architecture for Testing, Coordination and Views Model Checking*. PhD thesis, Universtà degli Studi di Roma "La Sapienza", 2002.

[23] Object Management Group. UML specification version 1.4, 2001. `http://www.omg.org/uml/`.

[24] A. Schürr, A.J. Winter, and A. Zündorf. *In [10]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

[25] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change manegement by distributed graph transformation: Towards configurable distributed systems. In *Proceedings TAGT'98*, volume 1764 of *Lecture Notes in Computer Science*, pages 179–193. Springer-Verlag, 2000.

[26] R.N. Taylor, N. Medvidovic, K.M. Anderson, E.J. Whitehead Jr., J.E. Robbins, K.A. Nies, P. Oreizy, and D.L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6):390–406, June 1996.

[27] D. Varró. Towards symbolic analysis of visual modelling languages. In Paolo Bottoni and Mark Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 of *ENTCS*, pages 57–70, Barcelona, Spain, October 11-12 2002. Elsevier.

[28] D. Varró. Towards automated formal verification of visual modeling languages by model checking. 2003. Extended version of [27]. Submitted.

[29] M. Wermelinger and J.L. Fiadero. A graph transformation approach to software architecture reconfiguration. In H. Ehrig and G. Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000), Berlin, Germany*, March 2000. `http://tfs.cs.tu-berlin.de/gratra2000/`.