Modeling and Analysis of Architectural Styles Based on Graph Transformation*

A Case Study on Service-Oriented Architectures

Luciano Baresi* baresi@elet.polimi.it

Reiko Heckel† reiko@upb.de

Sebastian Thöne‡ seb@upb.de

Dániel Varró§ varro@mit.bme.hu

ABSTRACT

Modern architectural styles, like the service-oriented style underlying web services, are highly dynamic. This complicates not only their practical application, but also the modeling and prediction of their behavior. To account for this problem, we propose to model architectures as graphs, represented as instances of UML class diagrams, and to describe their reconfigurations by graph transformation rules. Based on a sample model for service-oriented architectures, we discuss what properties are interesting to be analyzed and how such analysis could be performed.

1. INTRODUCTION

Nowadays, applications have to be adaptable to changes in (at least) two dimensions: *Changing requirements*, like requests for new functions, may require to integrate new components either statically or at run-time. *Changing environments*, like faulty communication channels or mobility leading to a reduced bandwidth, may require to replace unreachable components.

Current component platforms like CORBA, EJB, or Web Services provide the basic techniques to realize the required flexibility through reconfiguration mechanisms, like dynamic loading and binding of components. However, when using a platform to implement an application, the more interesting questions are non-technical ones, 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 desired configuration / realize the scenario?*

*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)

In order to answer such questions, an architectural model is required which allows to reason on a more abstract level, disregarding implementation details like 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 *architectural description languages (ADLs)* like Wright [1] or Rapide [19].

On the other hand, some of the above questions 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 languages like the Unified Modeling Language (UML) [24] 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 are not very good at describing 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 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 [28] or Fujaba [10], 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 [4], or *model checking* [31] 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. Based on this model, Section 3 explores possibilities for automated analysis. Section 4 surveys the related work while Section 5 concludes the paper and discusses possible future work.

2. MODELING

In this section, we present our proposal for modeling architectural styles with UML diagrams and graph transformation rules. An architectural style includes a static and a dynamic specification. The static part, described in Section 2.2, 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 2.3, specifies how a given architecture can evolve in reaction to planned reconfigurations or unanticipated changes of the environment. For better understanding, we choose service-oriented architectures as a case study for our proposal.

2.1 Service-oriented architectures

As shown in Fig. 1, taken from [6], service-oriented architectures involve three different kinds of actors: *service providers*, *service requestors* 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 requestor 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 requestors. As soon as the service requestor finds a suitable service description for its requirements at the agency, it can start interacting with the provider and using the service.

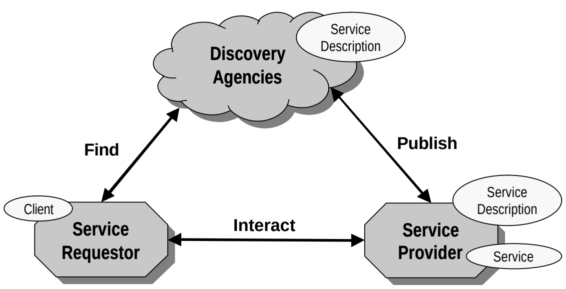

Figure 1: Service-oriented architecture

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 requestor 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 being able to confirm a request for a certain level of quality.

2.2 Static model

UML class diagrams [24] model the static part of the architectural style, which involves three different types of elements: *structural elements*, *messages*, and *specification documents*. These three groups are organized into individual packages. Due to space limitations, Fig. 2 provides an integrated overview of all three packages.

The *structural elements* like Component and Service are obviously required for the static model. In our case, a Component can play different Roles at the same time, i.e., a ServiceProvider can also be a ServiceRequestor and vice versa. The DiscoveryAgency is considered as subclass of ServiceProvider because it provides services dedicated especially to publishing and querying the service specifications. A ServiceRequestor interacts with a Service via a Session instance. This session contains the information about the current state of the interaction for each requestor since it is possible that different clients interact with the same service simultaneously. Hence, the session represents the actual connection between a requestor and a service.

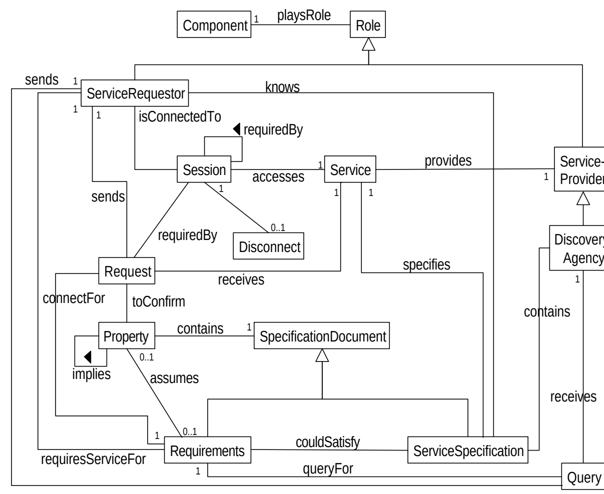

Figure 2: Service-oriented architectural style

Since a component that wants to initiate a certain reconfiguration usually has to communicate this to other affected components, we also provide the necessary types of *messages*: The Query message is used when searching the discovery agency for certain service specifications, and the Request and Disconnect messages are used for the creation and cancellation of a session.

We also include representations of *specification documents* in the static model. They are necessary to describe reconfiguration operations in which a component is dynamically searched at run-time and bound to certain requirements. There are two types of specification documents: Requirements and ServiceSpecifications, which both contain a set of Properties. In the case of a Requirements document, these properties are required by a ServiceRequestor for the Service it wants to use. In the case of a ServiceSpecification describing a particular service, these properties are guaranteed by the ServiceProvider. In some cases this might only be possible under certain assumptions (modeled by an assumes association to new Requirements).

The associations between the classes define how the above mentioned elements can be linked in a concrete architecture, constrained by the given cardinalities[1]. Other constrains and well-formedness rules can be added as OCL expressions [24]. For instance, the following expression restricts the allowed implies links between Properties to those pairs which actually satisfy a logical implication: [2]

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

An architecture compliant with the style can be regarded as an instantiation of the class model like in Fig. 3: Component comp2, which provides service s1 to the requestor sr1, also plays the service requestor role sr2 and uses the service s2. This is necessary to guarantee property p4 of the service specification whose assumptions are satisfied by s2. In this situation the session se2 is required to serve session se1. We model this dependency as a requiredBy link between the two sessions. This link can then serve as a reminder if somebody wants to close se2 while se1 is still running.

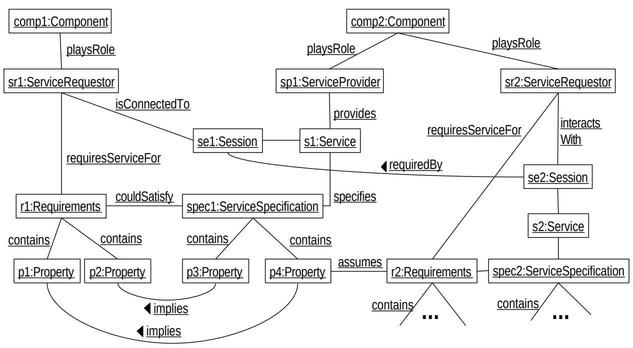

Figure 3: Example service-oriented architecture

2.3 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 (at least) two different ways of visualizing a graph transformation rule. One is to present a rule as a pair of two instance graphs of the architectural style defined in Section 2.2. The graph on the left-hand side defines the pre-conditions for the rule application, the graph on the right-hand side defines the post-conditions. In order to apply the rule, a matching of the left-hand side with a subgraph of the actual architecture has to be found.

For conciseness, we propose to integrate the two instance graphs into one *UML collaboration diagram* [24]. Elements which are then added to the architecture by the rule application are tagged with the label {new}, and elements which are deleted with the label {destroyed}. Unlabeled elements together with the {destroyed} elements form the left-hand-side, and the unlabeled elements together with the {new} elements form the right-hand-side of the rule.

Figure 4 shows a first example of such a rule in which a service requestor sends a request to the service it would like to connect to. As precondition the ServiceRequestor has to know a ServiceSpecification which couldSatisfy its Requirements. As postcondition the Request is created and linked to all Properties of the Requirements. This is done because the service provider that receives the request has to confirm all the required properties before a successful connection to the service can be established.

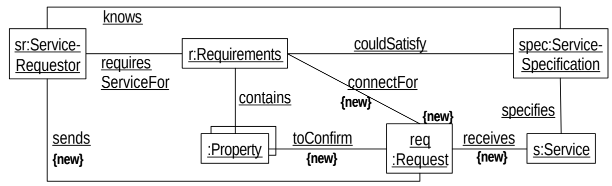

Figure 4: Creating a request for a known service

Due to space limitations we cannot present all reconfiguration rules for the service-oriented architectural style, but only a condensed excerpt. Therefore, we omit the rules which deal with the just created Request and try to confirm all required properties on the service provider side. Each time, the service provider can actually guarantee one of the required properties, the link toConfirm between that Property and the Request is deleted. Finally, if all properties have successfully been confirmed to the ServiceRequestor, a new Session for the Service is established as shown in Fig. 5.

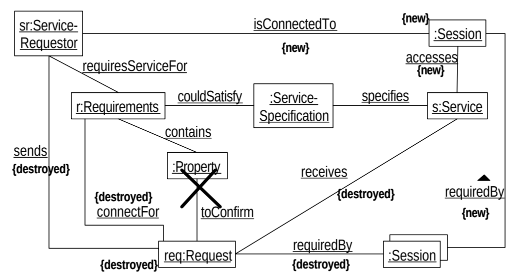

Figure 5: Create session and connect to the service

This rule contains a negative application condition which prevents its application if there are no properties in the requirements which still have to be confirmed by the provider. Otherwise, the rule can be applied to the architecture. It creates a new session instance which realizes the connection between the requestor and the service. Since the request has been fulfilled, the corresponding message can be deleted. After that, the binding of the requestor to the new service has been completed, and it can access the service.

The basic dynamic model consists of about ten more transformation rules which cover publishing a service description to a discovery agency, querying the agency for a description, creating a request for the service aiming at a new session, and disconnecting from an existing session. They do not yet capture a fault model and related repair mechanisms. 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 [9] combine graph rewriting rules based on UML object diagrams with control flow elements as provided by UML activity diagrams.

[1] No explicit cardinality means 0..n by default in Fig. 2.

[2] The first implies refers to the name of the association (see Fig. 2), the second one refers to the reserved OCL operator.

3. ANALYSIS

In this section, we identify automated means to formally reason about the correctness and consistency of architectural styles and concrete architectures captured by high-level specifications in the form of structural UML diagrams and graph transformation rules as discussed in Sec. 2.

3.1 Properties

Essentially, the analysis tasks we aim to carry out can be grouped into three main areas (as summarized in Fig. 6).

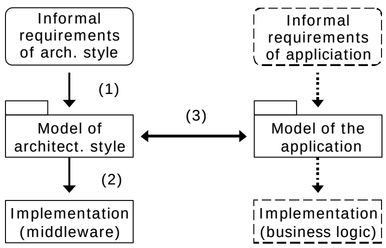

Figure 6: Analysis tasks

1. **Conformance of the model of the architectural style to the informal requirements.** At first, we have to show that the model of an architectural style fulfills the informal requirements. In this respect, the intuitively constructed graph transformation rules are *validated* whether they faithfully capture the intended dynamic behavior (i.e., the protocol or scenario) of the style. As for our SoA example, one can be interested in proving that, for instance, (i) each time a required service is provided by a certain provider, a connection will be built up sooner or later between the requestor and provider components, and (ii) eventually, the requestor and provider will be disconnected.

2. **Conformance of the implementation of the architectural style to its model.** A further analysis task is to prove that a concrete implementation of an architectural style (such as a specific middleware) corresponds to the formal model we constructed.

3. **Consistency of an application and an architectural style**. From an application point of view, it is much more important to prove that an architectural style is properly used by the application. Hence we need to show that the style and the application is consistent from both a static (well-formedness constraints are satisfied) and dynamic point of view (the application implements the protocol soundly). For instance, in a given application that uses the SoA style, one can ask whether an execution sequence of the application describing how to query a given service is consistent with the protocol defined by the style. Here, we typically perform *verification* as the behavioral model of the application is compared with a reference specification defined by the architectural style.

In the presence of faults, a carefully constructed fault model (captured (i) on the model level by additional graph transformation rules or (ii) on the requirements level by further assumptions) aims at formalizing what changes in the context are encountered. Afterwards, we can first assess the fault-tolerant capabilities of the style itself by proving consistency when certain well-formedness constraints are not satisfied by the application or the context. After identifying the dependability bottlenecks where certain repair actions are indispensable, new rules can be introduced to the operational description of the style to provide such repair mechanisms. Thus, the model checking process may continue with an extended rule set.

Below, we provide a brief overview of automated validation (Sec. 3.2) and verification (Sec. 3.3) techniques that we want to use to assess the correctness and consistency analysis of graph transformation based descriptions of architectural styles.

3.2 Validation

For the validation of the graph transformation rules aiming to capture the dynamic behavior of the architectural style, we propose two different techniques (with sufficient tool support):

- **Interactive simulation.** Many existing graph transformation tools (like Fujaba [10] or PROGRES [28]) offer an interactive visual environment for simulating the rules in order to estimate the behavior of an application in various situations. Simulation allows designers to play with "what if" scenarios and to concentrate on the key aspects of the particular architecture. Results are not as complete as with analysis, but they are readily available and more interactive.

- **Critical pair analysis.** Critical pair analysis [4] is a powerful technique to statically detect potentially conflicting rule pairs by automatically generating sample models for which the application of the two rules would be in conflict. Afterwards, the validator investigates these problematic situations to decide whether they really cause problems.

3.3 Verification

For the consistency verification of architectural styles, we propose:

- **Reachability analysis by graph parsing.** Many verification problems can be formulated as the reachability (or non-reachability) of a given configuration of the system. Built upon the technique of graph parsing, one can decide whether the target configuration can be generated by the graph transformation system if started from a specific initial model, thus providing means to backward reachability analysis.

- **Model checking graph transformation systems.** Given the structural description of the architectural style, the graph transformation system, and an arbitrary (bounded) model instance of a given application, we can automatically generate a state transition system [31] and verify properties by model checking.

  While previous techniques for validation preserve all information of the modeled system, in the model checking case only dynamic parts of the application (i.e., those that can be altered by a rule) are projected into the target transition system while static parts are simplified by a compile time preprocessing in order to obtain a manageable state space.

Properties to be verified are captured in the specification language of the model checker tool, which typically take the form of temporal logic formulae (as in the case of SPIN [16] or SAL [2]), or simple transitions that are not allowed to fire during model evolution (e.g., in Murϕ [23]).

4. RELATED WORK

Several proposals have influenced our work. First of all, we should mention the many ADLs (Architectural Description Language): Rapide [19], Wright [1], Darwin [20], C2 [30], and xADL [7] that gave us the first impulse. All these approaches mainly concentrate on concepts, which are well-defined, but in too many cases, the languages that render them are too difficult for the user. This is why we decided for a well-known representation of concepts, paired with a formal definition of their interaction and composition. Given the UML-like representation, we must mention the works by Medvidovic et al. [21] and by Garlan et al. [11]. They both study the suitability of UML to represent software architectures and identify some different alternatives. They can be seen as complement – instead of alternatives – to our proposal: We do exploit UML class diagrams to define the style of the architecture, but we do not address the problem of ascribing these concepts with a concrete syntax as proposed by the two aforementioned papers.

If we move to modeling the dynamic aspects, we must mention the CHAM approach [17], in which architectural reconfiguration is studied in terms of molecules and reactions, and the proposals that represent architectural styles by means of graph grammars and reason on changes and evolution with respect to structural constraints. Some approaches [18, 29, 32] assume a global point of view when describing reconfiguration steps which, in a real system, cannot be taken for granted. Other approaches (for example [15]) model reconfiguration from the point of view of individual components which synchronize to achieve non-local effects. Here, locality corresponds to context-freeness, that is, a rule is local if it accesses only one component (or connector) and their immediate neighborhood. Synchronization of rules is expressed in the style of process calculi.

Our proposal differs from the others since we do not use a grammar to generate the particular architecture. We use a model (i.e., class diagram and constraints) to express the valid instances of a given style. Graph transformation rules are exploited only to render the dynamic aspects like evolution and reconfiguration. The advantage is that a declarative specification is more abstract and easier to understand, even if a constructive/operational one is better for analysis and tools. As to the use of graph transformation, this approach is clearly inspired by what presented in [14] where the same ideas are applied to modeling the dynamic evolution of Web applications.

The most comprehensive work on analyzing architectures, is the Ph.D. thesis by Muccini [22] which offers a wide and complete presentation of the efforts on modeling, analyzing, and testing software architectures. He also proposes model-checking as a means to analyze architectures, but we start from two different perspectives. He directly models the particular instances as automata and their communication using message sequence charts; we adopt a wider approach and concentrate on evolution at style level. Architectures can evolve only because they are instances of particular styles.

We also want to take into account implementation-oriented approaches and proposals on self-adaptive and self-healing systems [12]. In the first set, we want to mention the work by Rutherford et. al. [27] that uses Enterprise JavaBeans as the underlying component model. It is interesting because of the "concrete" viewpoint, but they do not maintain a neat architecture model as we propose. In the second set, Oreizy et al. [25] discuss the problem and identify a set of significant needs. Georgiadis et al. [13] model structural architectural styles by means of Alloy: Their models are concise and elegant, Alloy supports automatic analysis, but the expressiveness of these models is not self-evident.

5. CONCLUSIONS AND FUTURE WORK

The paper presents a case study on modeling and analyzing architectural styles with graph transformation, exemplified on service-oriented architectures.

Our current work is on experimenting different solutions – besides that presented in the paper – to express rules, constraints and control mechanisms to find the right balance between expressiveness and analyzability.

Rules can be extended to address adaptability and the capability of automatic recovery ([12]). If we consider modern scenarios where applications are ubiquitous and they must adapt their behavior to the context in which they are executed, a disciplined approach to modeling these aspects is essential. Rules offer a clean and neat way to specify how the architecture should react to the different stimuli, but the analysis capabilities – both model-checking and simulation – complement the design with the capability of automatic reasoning and predicting the behavior of specified architectures. In a similar way, rules can specify the *self-healing* capabilities associated with the specific style or family of architectures.

Notice that the graph transformation system can also be seen as the coordinator that supervises the reconfigurations. In this context we do not want to discuss all related problems and implementation issues, which would be premature, but rather to pinpoint the possibility of using the same technology both as modeling means and as run-time supervisor.

If we do not embed the graph transformation system in the running environment, we can use it to test the architecture. Plans here cover the derivation of both suitable test cases – at architecture level – 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 [3]), 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 is almost standard practice if we consider pure textual grammars ([5]).

Similar considerations apply to oracle generation: model-based oracles are well-known (e.g., [26]), but the use of a graph transformation system as abstract level is innovative.

6. REFERENCES

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

[2] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.

[3] 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.

[4] 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.

[5] 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.

[6] 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/`.

[7] 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.

[8] 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.

[9] 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* 6^{th} *International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, LNCS 1764. Springer Verlag, 1998.

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

[11] 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.

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

[13] 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.

[14] R. Heckel and M. Lohmann. Model-based development of web applications using graphical reaction rules. In *Proc. Fundamental Approaches to Software Engineering (FASE'2003), Warsaw, Poland*, LNCS. Springer-Verlag, 2003. To appear.

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

[17] 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.

[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* 5^{th} *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] *The Murϕ Model Checker*. `http://verify.stanford.edu/dill/murphi.html`.

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

[25] P. Oreizy, M.M. Gorlick, R.N. Taylor, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[26] D.J. Richardson, S. Leif-Aha, and O.T. Owen. Specification-based Test Oracles for Reactive Systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118, May 1992.

[27] M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A.L. Wolf. Reconfiguration in the Enterprise JavaBean Component Model. In *Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 67–81, 2002.

[28] A. Schürr, A.J. Winter, and A. Zündorf. *In [8]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.

[29] 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.

[30] 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.

[31] 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.

[32] 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/`.