Exploiting Protocol Information for Speeding up Runtime Reconfiguration of Component-Based Systems

Jasminka Matevska-Meyer, Wilhelm Hasselbring, and Ralf H. Reussner
Software Engineering Group, Department of Computing Science
University of Oldenburg, Germany
{matevska-meyer, hasselbring, reussner}@informatik.uni-oldenburg.de

Abstract

To reduce the down-time of software systems and maximise the set of available services during reconfiguration,we propose exploiting component protocol information. This is achieved by knowing the state of a running system and determining the component dependencies for the time interval from receiving a reconfiguration request until reconfiguration completion. For this forecast we use the architectural descriptions that specify static dependencies, as well as component protocol information. By considering only component interactions for the time interval of reconfiguration we can exclude past and future dependencies from our runtime dependency graphs. We show that such change-request-specific runtime dependency graphs may be considerably smaller than the corresponding static architecture based dependency graphs; this way, we are speeding up runtime reconfiguration of component-based systems while maximising the set of available services.

Keywords from CFP dynamic composition of component-based systems, dynamic architectures, system design for hot-swappable components, addressing variability requirements in component-based solutions

1 Introduction

Runtime reconfiguration plays an important role for providing high availability of software systems. One of the main issues during runtime reconfiguration is to maintain the consistency of the system. Furthermore, one is interested in minimising the down-time of the system caused by the reconfiguration. Due to that, techniques are required which determine the parts of the system to be halted during reconfiguration, and, accordingly, the parts of the system which can continue execution during reconfiguration. This lowers the down-time of the system and maximises the set of services available during reconfiguration. This is an essential requirement not only for mission critical systems, but also for the steadily increasing number of commercial web-applications.

We distinguish between three different types of reconfiguration according to their reconfiguration effort: (1) functional, (2) non-functional, and (3) structural. All types of reconfiguration can occur on different levels of granularity (i.e., can concern the entire system or a single sub-component.) In our approach, we do not distinguish between components, connectors and systems, because of the possibility to specifying them in the same manner. *Functional reconfigurations* include changes to the functionality of a single component as well as of a particular subsystem, even of the entire system. *Non-functional reconfigurations* are concerned with the quality of service of the system and can affect single components (sub-systems) or the architecture. *Structural reconfigurations* consider both, changing the interface of a single component and changing dependencies among components (architectural changes of a system).

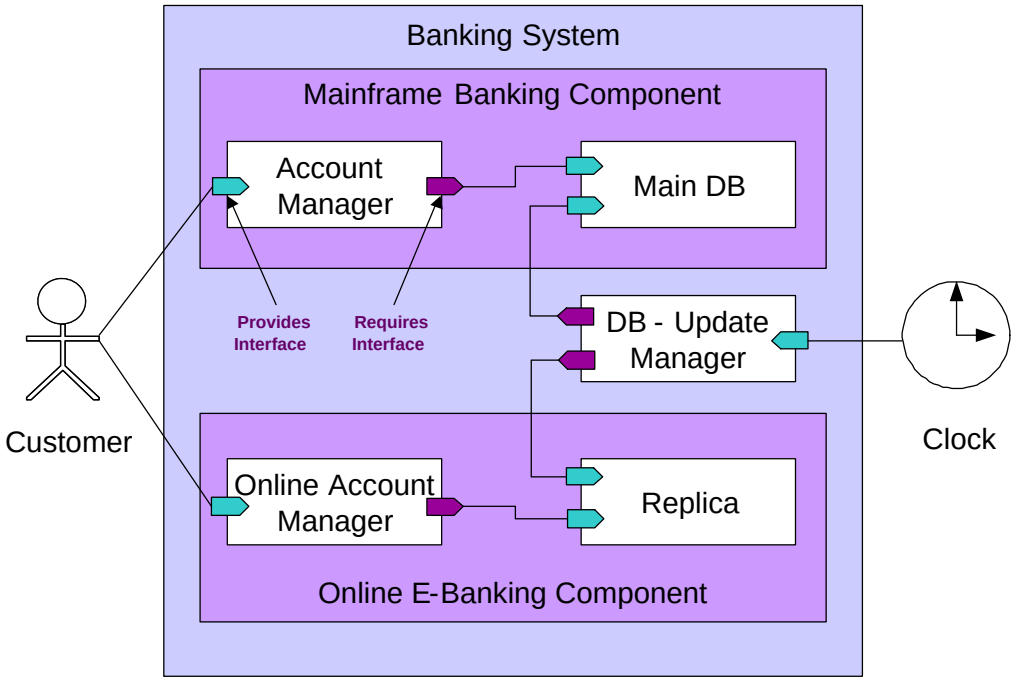

Figure 1. Banking system

The major problem to be considered when dealing with reconfiguration of component-based systems is the existence of dependencies among components. These dependencies should be defined in the static architectural description of the system (e.g., [MT00]). However, when reconfiguring a system in a concrete runtime state one usually does not need to consider all of these dependencies. Our idea is to observe a running system during a time interval from receiving a reconfiguration request until reconfiguration completion, and to determine a minimal set of affected components for that interval. Knowing the current state of all possibly affected components, and their future behaviour (by their interaction protocols), we can exclude past dependencies and late future ones. This allows us to built a runtime dependency graph containing considerably less dependencies than the graph of the static system architecture. Consequently, there are less actually affected components to be considered for individual reconfiguration requests. Hence, the set of available services of the system is maximised, while the overall down-time is reduced.

This paper is organised as follows. First, we present an example describing the problem and our suggested solution (Section 2), next, we give a formal description of our idea in Section 3. In Section 4, we propose a system architecture. Related work is discussed in Section 5. Finally, we conclude and indicate further work in Section 6.

2 Motivating Example

To illustrate our approach we chose a simple banking system (Fig. 1). On its top-level, it consists of a *Mainframe Component*, an *Online E-Banking Component*, and a *Database Update Manager*. The mainframe component itself is assembled from an *Account Manager* and the *Main Database*. The e-banking component consists of an *Online Account Manager* and a *Replica* of the database.

Transactions processed by the mainframe component are continuously propagated to the database and are persistently stored there. Opposed to that, the additional online interface of the e-banking component for web customer access works with a replica of the database, because the traditional mainframe database does not properly support online transactions. The replica stores all daily online transactions and updates the main database only during a nightly pass.

A consistency problem may occur when the same bank customer initiates, for example, the following two transactions: at first she transfers the complete balance online and then withdraws at cashpoints. The online system,

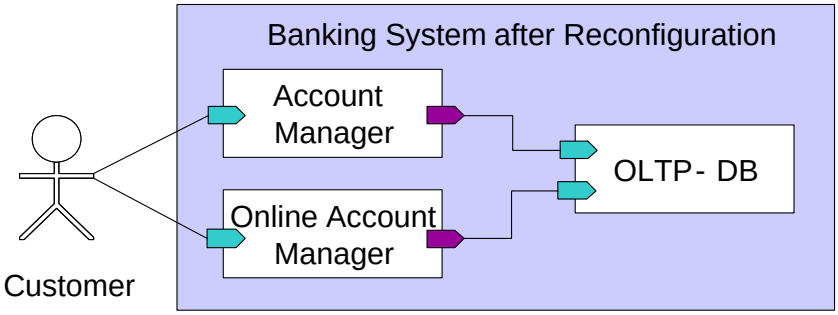

Figure 2. Banking system after reconfiguration

which works on a replica, checks whether the first transaction is allowed, but does not immediately propagate the account changes to the main database. Hence, the second transaction will temporary pass through. After the nightly update, the first (online) transaction will be refused and the customer may have a financial problem (with possibly a too late notification, if any). As this situation is not satisfactory, the system shall be changed to work on a single database.

The static dependency graph of our system, as given by the system architecture, is shown in Figure 3 (a). It shows that all components have to be halted during this reconfiguration. This means, no services will be available during merging the main data base and the replica.

Opposed to that approach, we consider the time point and expected duration of the reconfiguration. Therefore, the system is reconfigured in two separate steps:

1. the main database is replaced by an online transaction processing database during night hours
2. after the nightly update, the database-update manager and the replica are removed. Consecutely, the online account manager is linked to the main database (Fig. 2).

As we perform the first step outside business hours, so even the conventional account manager does not have to be halted. Although, there is a static dependency between the conventional account manager component and the main database, we can neglect this in our runtime dependency graph (Fig. 3 (b)) for this specific reconfiguration interval. In addition, also the online account manager does not have to be halted, as it is not affected by that change. Since the second reconfiguration step is done after the nightly update (and will not last until the next one), we can perform it with only halting the online account manager. Note that during this second reconfiguration step, the main database is already online-transaction enabled, hence, inserting the records from the replica can be done at any time (even during business hours). Anyhow, the online account manager has to be halted while inserting the records and while changing the connection from the replica to the main database to avoid inconsistent views for online customers.

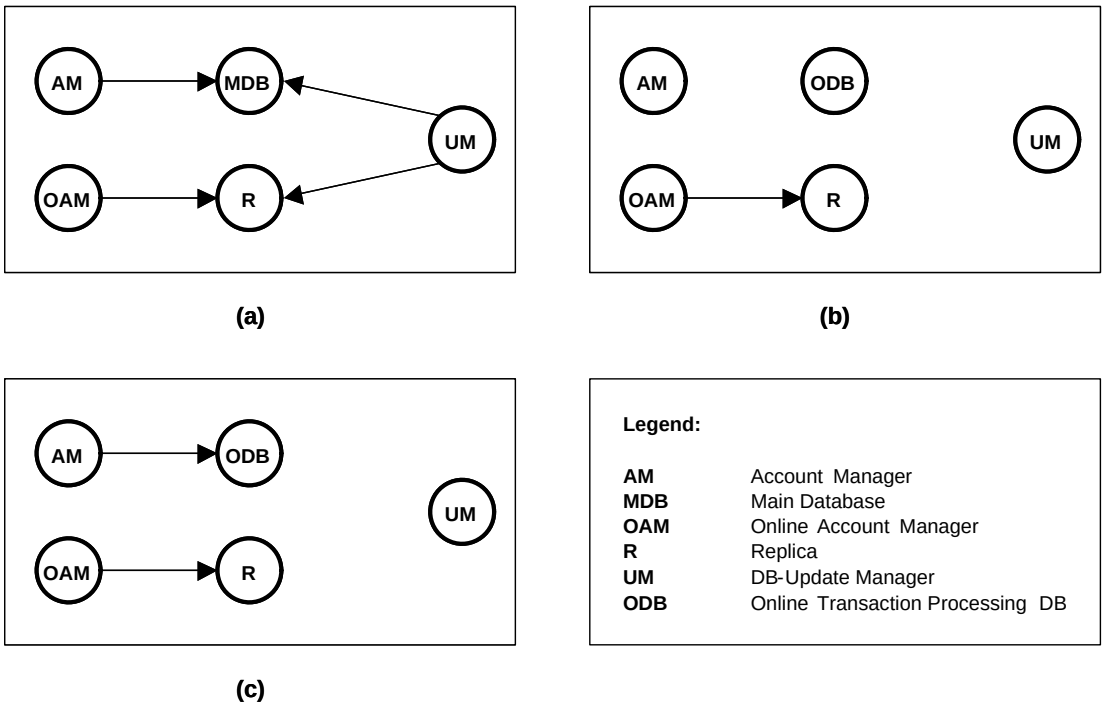

Figure 3. DependencyGraphs: (a) Static dependency graph, (b) First step runtime dependency graph, (c) Second step runtime dependency graph

3 Formalisation

The basic idea of our approach is to minimise the set of components affected by a runtime component or architecture change. If a component is changed (or even deleted in case of an architectural change) one has to ensure the consistency of the overall system. Most commonly, the architectural system description is used for determining the set of components affected by a change.

Let denote $u(x)$ the set of components directly affected by the change of a component x (i.e., $u(x)$ contains all components associated with component x in a direct use-relationship). Since all components, *directly or indirectly* affected by a change of x have to be considered, one has to build the transitive closure $u^2 := u(u(x)), u^3 := u(u^2(x)), \cdots$. (As the number of components of a system is finite, this series of sets reaches a fixpoint, that is the transitive closure $U(x) := u^\infty(x)$). In many realistic examples, this set of components affected by an update of x comprises are (nearly) all components of a system. Consequently, when updating a single component one often has to halt the whole system. Of course, from a pragmatic point of view, one wants to avoid that. Especially, in technical control systems a permanent operation is desired. A halt of the control software usually implies the halt of the whole controled system, e.g., the production line, etc.

Our approach minimises this set of affected components by considering not only the architectural description of a system with its static dependencies but also (a) component interaction protocols (i.e., specifications of the dynamic behaviour), (b) timing behaviour of tasks and their periodicity (as the nightly batch update in our example), (c) information on the current state of the system (as given in the tuple of the states of each single component and connection), and (d) the expected duration of the update. In particular, we make use of the components' *requires protocols*, i.e., the sequences of method calls to external services. We denote the requires protocol of a component x as R_x. As we use a finite state machine (the *requires automaton*) for specifying the requires protocol, the requires automaton and the requires protocol (i.e., the language generated by the requires automaton) are identified.

Our example shows that the set $U(x)$ can be considerably reduced, if it is computed taking the above factors (a) to (d) into account.

In our approach, the set $u_{s,T}(x)$ is defined as the set of components using component x in the system state $s = (s_1, \cdots, s_n)$ and for the duration T (e.g., the next 200 seconds). (The state s is a tuple consisting of the sates s_i of basic components and connectors. Let m denote the number of components and connectors. For our purposes, we do not differentiate between components and connectors here, but assume that s_i denotes the state of component x_i for $0 \leq i \leq n < m$.)

In the following a more formal sketch of the computation of the set $u_{s,T}(x)$ is given to demonstrate its feasibility:

1. For each component x_i we use its requires protocol R_{x_i} for computing the language $L_{s_i}(R_{x_i})$. $L_{s_i}(R)$ is defined as the language generated by the automaton R with the start state s. (Hence, we use the automaton $\hat{R}$ which is identical to R in any respect but possibly with a different start state. Thus, $L_{s_i}(R) = L(\hat{R})$).

2. Now we make use of knowing the timing behaviour. We assume we have a function τ mapping transitions and states to the positive real numbers, denoting for each state transition the time used to change the states and for each state s_i, $\tau(s_i)$ gives the time lasted in that state before doing a subsequent transition. (By that transition the state may be changed or not.)

3. With τ one can annotate each symbol w_i of each word $w := w_0 \cdots w_l \in L_{s_i}(R)$ with the time it takes for the automaton to read it. Starting with 0 from the start state, we then have $w = w_0^0 w_1^{t_1} ... w_l^{t_l}$ with $t_i \in \mathbf{R}$ and $t_i < t_j, i < j$. The set $L_{s,T}(x)$ contains for each word $w \in L_{s_i}(R)$ the maximal prefix p_w which can be accepted in time T. Maximal prefix means that within time T the processing of the last character of p_w has started. Formally: $p_w := w_0^0 w_1^{t_1} \cdots w_k^{t_k} w_{k+1}^{t_{k+1}}$ with $t_k < T$ and $t_{k+1} > T$ (and $k + 1 \leq l$). Informally, the set $L_{s,T}(x)$ contains all call sequences the component x is expected to perform for invoking external methods being in state s within the time T according to its requires protocol. Having these call sequences one also knows the components of the methods invoked by x within time T. This set of components is the result of a projection Π projecting sequences of external method calls to the components providing these methods. Technically, this mapping is given by the architectural description of the system with its connections between components.

4. Finally, we can compute the set $u_{s,T}(x)$ of all components using x in system state s and for the time T by checking for each component x_i (besides x itself) whether $x \in \Pi(L_{s_i,T}(x_i))$. We then yield $u_{s,T}(x)$ as the set $\{x_i | x \in \Pi(L_{s_i,T}(x_i))\}$.

Based on the set $u_{s,T}(x)$ we compute its transitive closure $U_{s,T}(x)$. (Note that therefore we do not have to recompute any of the sets $u_{s,T}(x_i)$ of any component x_i.) For the sake of brevity, we omit the formal specification of the timing behaviour and illustrate our postion with the informal specification of the example as given in Section 2. (However, such interaction and timing behaviour can be specified for example in annotated Message-Sequence-Charts as used in Real-Time UML [Dou99].) From that we see that $U_{s,T}(x)$ is considerably smaller than $U(x)$. For the first reconfiguration step we yield $U_{s,T}(\textit{Main DB}) = \{\textit{Main DB}\}$ and for the second step we have $U_{s',T}(\textit{DB-Update Manager}) = \{\textit{DB-Update Manager}\}$ and $U_{s'',T}(\textit{Replica}) = \{\textit{Replica}, \textit{Online Account Manager}\}$. That is considerably less than performing all configurations in one step (what would be most likely the case with having only static dependency information and not considering the behavioural specification of the components). Considering only the static information, an update of the main database and a deletion of the replica would have affected all components, as any component uses the main database or the replica.

4 Proposed System Architecture

We name our system *Reconfiguration Manager* (Fig. 4). It is activated on *reconfiguration requests*. It consists of the following four top-level components [MMH03]: *Reconfiguration Analyser*, *Dependency Manager*, *Consistency Manager* and *Reconfigurator*.

The reconfiguration analyser takes a *reconfiguration request*, analyses and classifies the requested change according to the reconfiguration types: functional, non-functional or structural. Also the components (including

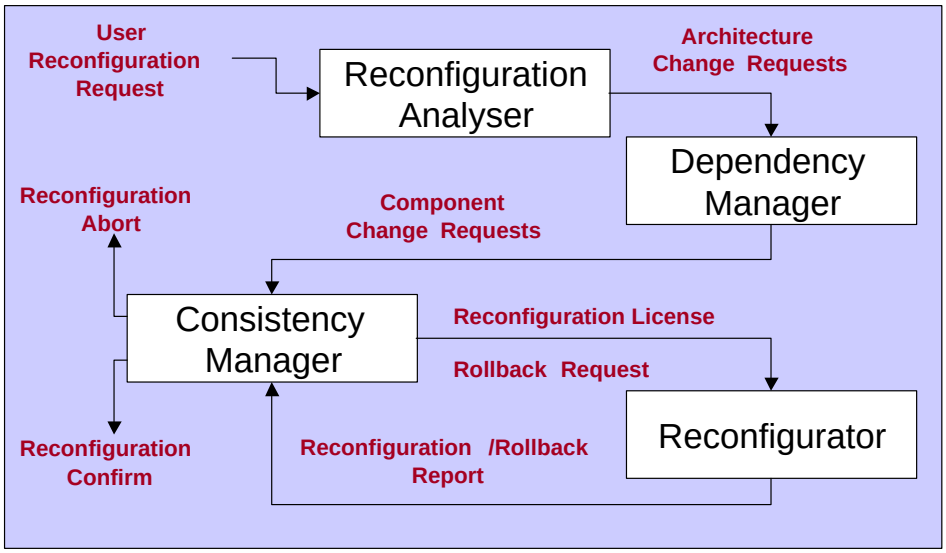

Figure 4. Reconfiguration Manager

connections) directly concerned with that *change request* are identified. The dependency manager monitors the run-time dependencies among components, determines a minimal set of change-affected components (as described in Section 3) and sends a *change request* for each involved component to the reconfigurator. The reconfigurator realises the reconfiguration as a dependent change transaction [KM90]. It starts a transaction, transfers the affected components (and only these) into a *blocked* state, isolates an affected subsystem, applies the changes, and sends a *consistency-check-request* to the consistency manager. The consistency manager then checks the consistency of the changed system by monitoring the change transaction. This is necessary, since the set of affected component computed by the dependency manager is only valid for a duration T the *change request* is expected to operate (cf. Section 3). If the transaction violates this time limit, the consistency manager detects this failure and initiates a *rollback*. If the time limit is not exceeded the transaction is committed. After the *rollback* or the *commit*, all *blocked* components are transferred back into a *running* state.

This paper focuses on the dependency manager.

5 Related Work

Runtime reconfiguration is a very active research area in various disciplines of computer science.

Distributed systems [CS02], [PBJ98], [Kni99], [KC00] work on dynamic component updates. Even if contractually defined components with behaviour-specifying interfaces are considered (e.g., [PV02]), the runtime state and the interaction protocols are not used for restricting the dependency graphs.

Architecture-based reconfiguration In [OMT98], [OT98], the runtime reconfiguration is treated as the replacement of single components at the time and at architectural level, but no component dependencies are addressed. Structural changes are performed by altering connector bindings.

Software configuration management focuses on reconfiguration (e.g. [Lar01]) where dependency graphs are used to determine the dependencies among different versions of components, but neither architectural nor runtime aspects are considered,

Reconfiguration as an extension of the deployment process [RAC+02], [CS02], but without considering architectural changes of the system and without checking its consistency at runtime.

Our approach combines the disciplines software architecture, software configuration management and component deployment (as on principle discussed in [vdH99]) by considering contractually-defined and contractually-used software components [RS02], [Szy98], as well as functional and non-functional changes. Furthermore, architectural and version reconfiguration of a component-based system is allowed. We concentrate on runtime reconfiguration, considering the time as an additional factor, with the explicit goal of reducing the system down-time by minimising the set of affected components.

6 Summary and further work

Runtime reconfiguration of component based systems allowing changes of the dependencies among components is discussed. We observe a running system at a particular time interval from receiving a reconfiguration request until reconfiguration completion. We assume the availability of a static system description (e.g., static dependency graph, component specification) and specified or derived component protocol information. Knowing the current state of all possibly affected components, we can exclude past dependencies and late future ones. This technique is used to create a runtime dependency graph matching the particular reconfiguration request. In our example we show that this change-request-specific runtime dependency graph can be considerably smaller than the static architecture based dependency graph.

Although, our approach requires more information than a mere analysis of an ADL system specification, we can draw on previous work in component interaction protocol specification and the automated derivation of these specifications from design documents (such as Message Sequence Charts) [Wyd01] or source-code [Hun01].

We present a high-level architecture of a Reconfiguration Manager. For one of its subcomponents, the Dependency Manager, we present a formal model for computing a minimal set of components being affected by a the change of a component. Of course, the other components of the system also have a considerable complexity, not presented in this paper.

Future work includes the evaluation of the presented approach by means of larger case studies as well as by comparison with other approaches. Besides that, further research is required for the proposed reconfiguration manager (e.g., investigating the possibilities to extend the deployment process). As the possibility of runtime reconfiguration is increasingly perceived as a benefit of component-based software (of such different areas as embedded consumer electronics, technical control systems and enterprise computing) we expect considerable benefits from minimising system down-time as well as increasing service availability during system reconfiguration.

References

[CS02] Xuejun Chen and Martin Simons. A component framework for dynamic reconfiguration of distributed systems. In Judith Bishop, editor, *Proceedings of IFIP/ACM Working Conference on Component Deployment*, pages 82–96, Berlin, Germany, June 2002. Springer-Verlag Berlin Heidelberg.

[Dou99] Bruce Powel Douglass. *Real-Time UML*. Addison Wesley, second edition, 1999.

[Hun01] Gunnar Hunzelmann. Generierung von Protokollinformation für Softwarekomponentenschnittstellen aus annotiertem Java-Code. Diplomarbeit, Fakultät für Informatik, Universität Karlsruhe (TH), Germany, April 2001. Generating Protocol Information for Software Component Interfaces from Java Code.

[KC00] Fabio Kon and Roy H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, January 2000.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[Kni99] G. Kniesel. Type-safe delegation for run-time component adaptation. In Rachid Guerraoui, editor, *Proceedings of ECOOP'99*, pages 351–366. Springer, June 1999.

[Lar01] Magnus Larsson. *Applying Configuration Management Techniques to Component-Based Systems*. PhD thesis, Uppsala University, Sweden, December 2001.

[MMH03] Jasminka Matevska-Meyer and Wilhelm Hasselbring. Enabling reconfiguration of component-based systems at runtime. In J. van Gurp and J. Bosch, editors, *Proceedings of Workshop on Software Variability Management*, pages 123–125, Groningen, The Netherlands, February 2003. University of Groningen.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[OMT98] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177–186, April 1998.

[OT98] Peyman Oreizy and Richard N. Taylor. On the role of software architectures in runtime system reconfiguration. In *Proceedings of the International Conference on Configurable Distributed Systems 4*, Annapolis, Maryland, May 1998.

[PBJ98] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proceedings of International Conference on Configurable Distributed Systems*, pages 35–42. IEEE CS Press, March 1998.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, November 2002.

[RAC+02] Matthew J. Rutherford, Kenneth Anderson, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. Reconfiguration in the Enterprise JavaBean component model. In Judith Bishop, editor, *Proceedings of IFIP/ACM Working Conference on Component Deployment*, pages 67–81, Berlin, Germany, June 2002. Springer-Verlag Berlin Heidelberg.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[vdH99] André van der Hoek. Configurable software architecture in support of configuration management and software deployment. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 732–733, New York, May 1999. Association for Computing Machinery.

[Wyd01] Bart Wydaeghe. *Component Composition Based on Composition Patterns and Usage Scenarios*. Dissertation, Department of Computer Science, Vrije Universitiet Brussel, Belgium, 2001.