A Scenario-based Approach to Increasing Service Availability at Runtime Reconfiguration of Component-based Systems

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

Abstract

Component-based business-critical systems evolve during their life cycle in order to meet changing requirements or to improve quality properties. At the same time, maintaining continuous availability of services is an issue with increasing importance for these systems. Runtime reconfiguration supports evolution of systems while maintaining availability of services they provide. In this paper, we present a new approach to runtime reconfiguration of component-based systems, which aims at optimising availability of requested services during reconfiguration. For a particular reconfiguration request, we analyse relevant scenarios based on the usage model of the system and exploit the component protocol information. Considering varying number of requests for a service at different points in time as well as the priority of each service, we compute minimal runtime dependency graphs among component instances and thus the maximal possible availability of services provided by the system. Finally, we sketch the system architecture for evaluating our approach.

Keywords component-based systems, component provided/required services, availability, runtime reconfiguration, architecture-based reconfiguration, scenario modelling

1 Introduction

Reconfiguring component-based systems at runtime aims at maintaining the availability of provided services during the reconfiguration process. This plays a very important role especially for mission critical business systems to prevent financial loss. The process of reconfiguration consists of the following four steps: (1) initiation of a change, (2) identification of affected components, (3) accomplishment of the reconfiguration and (4) analysis/check of the consistency. There exists a large variety of reconfiguration approaches with different goals depending on the focus on a particular step. For example, context-sensitive and fault-tolerance approaches [2] aim at recognition of needed changes, graph-transformation approaches [33, 8, 9] optimise architecture transformation methods, consistency checks are the focus of protocol-based approaches [24, 1, 13], deployment strategies concentrate on the process of performing reconfiguration [28, 5]. All runtime reconfiguration approaches and techniques aim implicitly at the same goal: increasing the system availability. They usually utilise a particular basic reconfiguration technique and extend it with specific concepts needed to achieve the approach specific goal. Orthogonal to them, our approach focuses on optimisation of the complete process of reconfiguration and that way maximising the set of available services during runtime reconfiguration.

For this optimisation, we propose exploiting system usage model and component protocol information. We split each reconfiguration request into a set of component reconfiguration requests. For each component reconfiguration request we determine the set of affected components as communicating parts and thus reduce the usage model of the system to relevant scenarios concerning the affected subsystem to be analysed. Additionally, knowing the component protocol information (required service effect automata [26]), we can analyse the runtime dependencies among affected components and determine a state at system runtime with a minimal set of used services. Executing the requested reconfiguration step at this point in time increases the system availability during the reconfiguration. Each reconfiguration step is performed as a change transaction [14, 7] and thus maintaining the consistency of the system.

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

Component	Transaction Controller	Account Controller	Customer Controller	Web Client	Application Client
Provided Services	withdraw deposit makeCharge makePayment transferFunds getTxsOfAccount	createAccount removeAccount addCustomerToAccount removeCustomerFromAccount getAccountOfCustomer getDetails	createCustomer removeCustomer getCustomersOfAccount getCustomersOfLastName	withdrawAction depositAction makeChargeAction makePaymentAction transferFundsAction getTxsOfAccountAction getAccountOfCustomerAction getDetailsAction	createAccountAction removeAccountAction addCustomerToAccountAction removeCustomerFromAccount Action getAccountOfCustomerAction getDetailsAction createCustomerAction removeCustomerAction getCustomersOfAccountAction getCustomersOfLastNameAction
Required Services	getData updateData	getData updateData	getData updateData	withdraw deposit makeCharge makePayment transferFunds getTxsOfAccount getAccountOfCustomer getDetails	createAccount removeAccount addCustomerToAccount removeCustomerFromAccount getAccountOfCustomer getDetails createCustomer removeCustomer getCustomersOfAccount getCustomersOfLastName

Table 1. Provided and Required Services

2 Motivating Example

To illustrate our approach we chose a simplified Sun Microsystems Duke's Bank Application [31] in Fig. 1. On the client side it consists of a *Web Client* and an *Application Client*. The Web Client provides an interface to typical online-banking services. The Application Client provides an interface to administration services. The business logic of all services is realised with three components: a *Transaction Controller*, an *Account Controller* and a *Customer Controller*. The back-end component is a *Data Base*. Table 1 shows an overview of the services provided through and required by these components.

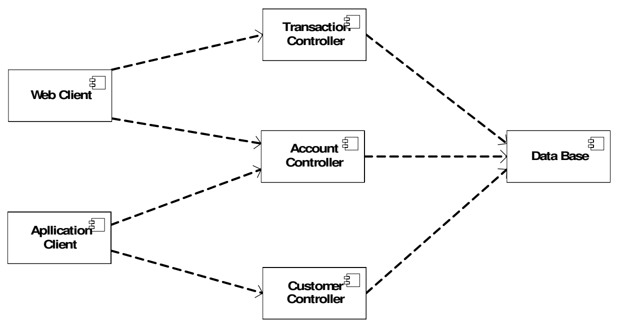

Figure 1. Component dependencies in Duke's Bank Application

Suppose, we get a reconfiguration request to change the Account Controller Component. A static reconfiguration approach would (1) shut down the system, (2) perform the reconfiguration and (3) start up the system. The system won't be available during the entire process of reconfiguration. A traditional approach to runtime reconfiguration would coarsely (1) identify concerned components, (2) identify all affected components, (3) passivate the affected components, (4) suspend upcoming requests, (5) perform the reconfiguration and (6) activate the components. Not af-

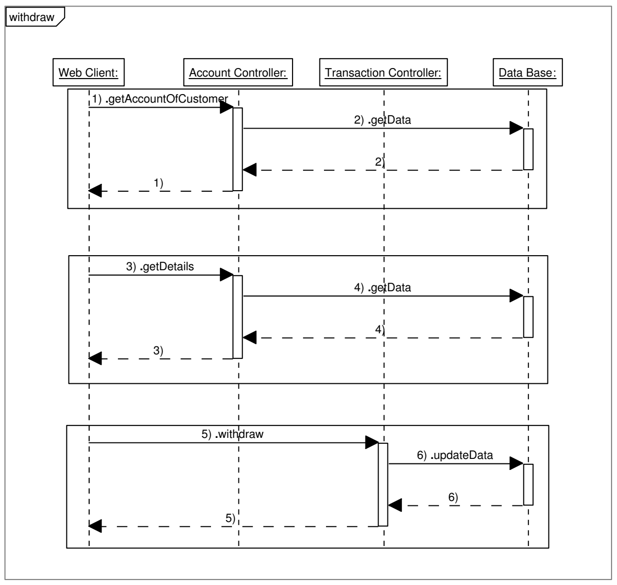

Figure 2. Withdraw Success

fected components would be available during the process of reconfiguration. The runtime reconfiguration potentially increases the availability of the system. This would, however, not increase the availability of the system for the given reconfiguration request. The static dependency graph (Fig. 1) shows that for changing the Account Controller both client components would be affected. Thus, no client component and no service would be available during the reconfiguration. Our approach additionally analyses changing dependencies among instances of components during runtime considering the usage model of the application. For simplicity we consider only the success scenario of the business process *withdraw* as a UML sequence diagram (Fig. 2). At different points in time we can observe different dependency graphs among instances of affected components. Fig. 7 (b) presents the runtime dependency graph during steps 1, 2, 3 and 4 and Fig. 7 (c) presents the runtime dependency graph during steps 5 and 6. Performing the requested reconfiguration during steps 5 and 6 maximises the availability of the system because neither the Web Client nor the Application Client are affected and have to be halted. But, for a multi user Web System, it is very improbable that there is a point in time during runtime at which there are no runtime dependencies at all for a particular component. Even in that case, we show that considering changing workload during runtime for more precise determination of minimal runtime dependency graphs can considerably contribute to increasing the availability during reconfiguration.

3 Our Approach

In this section we present our approach in greater detail. First, we describe all relevant aspects and additionally used information, and then we show how they work together to achieve our goal of increasing availability during runtime reconfiguration.

3.1 Usage Model

A software application is usually being developed to fulfil a particular mission. Therefore, a mission specific usage model has to be considered as part of the requirements specification. In general, every usage model consists of different usage scenarios. Each scenario defines a particular use case of the system. For each use case there is at least one possible execution sequence. Each execution sequence is defined through its behaviour and a set of participating components. Our meta model is illustrated in Fig. 3. Currently, we model sequences with UML 2.0 sequence charts. An extension of the Message Sequence Charts (MSCs), the Live Sequence Charts (LSCs) [6] and especially the scenario-based approach *play-in play-out* [10] using LSCs for specification of embedded systems would be considered for the formal specification of our scenarios. LSCs are more expressive than UML sequence charts and MSCs, because they can distinguish between possible and mandatory behaviour. A formal semantics of LSCs is expressed in terms of Timed Büchi Automata. LSCs are used for system verification and model checking. An extension of the behaviour definition including transition probabilities (e.g. as a Markov chain model [32]) could be used to achieve a weighting of sequences and thus reducing the amount of transition states considered as relevant for the determination of minimal runtime dependency graphs.

3.2 Service Effect Specification

A service effect specification (SEFF) of a component contains a set of descriptions on how each provided service of a component calls its required services. A SEFF can be seen as an abstraction of the control flow through the component. It can be modelled as a finite state machine (FSM) and can contain sequences, branches, and loops. A detailed

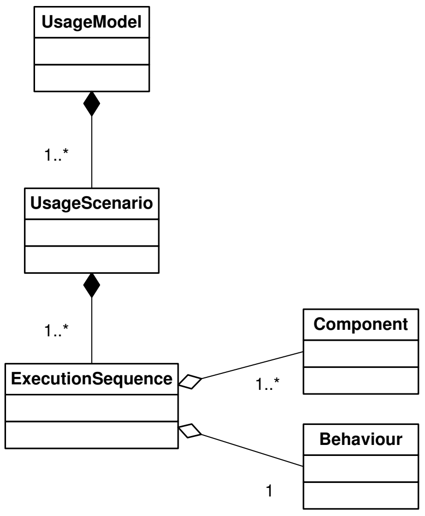

Figure 3. Usage Meta Model

and formal description of SEFFs can be found at [26]. A Service execution is a transition of a protocol state machine. For each service provided by a component we can define an appropriate SEFF. A set of all SEFFs for a component defines its external behaviour.

Let's take a look at our example, considering the withdraw-scenario. If a Web Client executes its provided service *withdrawAction* (Fig. 1) it has to call some of its required services. First it calls the service *getAccountOfCustomer*, second, it calls the service *getDetails*, both provided by the Account Controller. Finally, it calls the service *withdraw* provided by the Transaction Controller. This is illustrated with an UML state diagram in Fig. 4.

Using SEFFs we can analyse the past and future behaviour of a component and its dependencies. Considering a particular scenario being executed we can determine corresponding runtime dependency graphs.

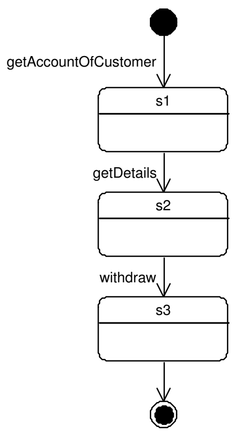

Figure 4. Web Client *withdrawAction* **service effect specification**

3.3 Runtime Dependency Graphs

Dependencies among components are static and represent the worst case dependency graph of the system. We consider only systems with acyclic dependencies. Each component can provide some services while requiring external services. A component A is dependent on component B, iff A uses/requires services provided by B (depicted in Fig. 5).

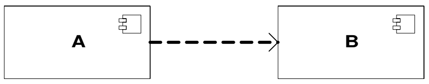

Figure 5. A is dependent on B

We assume that each component adhere to the life cycle protocol shown in Figure 6 after it has been deployed. A component is *active and not used* if there are instances of it executing some services. A component is *passive and used* if another component has an active reference to it. It is possible that a component is *active and used* at the same time. To ensure the consistency of the system, a component can be changed only if it is in the state *blocked/ready to change* and only *free (not active and not used)* components can be blocked [16].

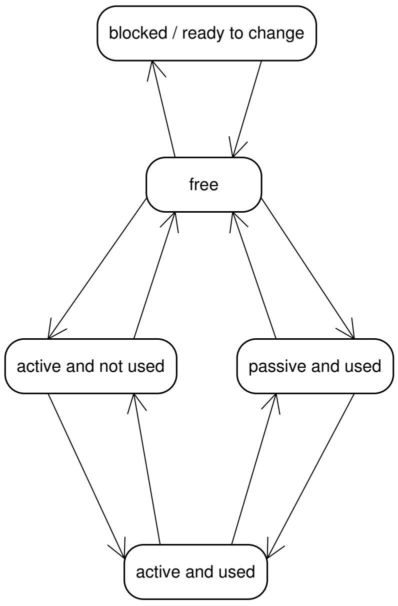

Figure 6. Component Life Cycle Protocol

During runtime a system executes different scenarios and thus activates particular instances of components. At a particular point in time there are components that are free (not active and not used). At that point in time they do not have any dependencies. The runtime dependency graphs among instances of components can never contain more dependencies (edges) than a static dependency graph of a system. For a particular reconfiguration request there exists a set of affected components (a transitive closure including components to be exchanged and dependent ones) as a subgraph of the static dependency graph. Considering the possibility of having more than one instance of a component, we define the weight of an edge. Furthermore, we have to consider the number of requests on a service and its weight for each provided service of the component (destination node). To determine which of those services are currently used for a particular scenario being executed, we exploit the required service effect specification [17]. Using this information we can define a minimal runtime dependency graph as a graph with a minimal sum weight. We calculate the sum as follows:

$$W_{set}(t) = \sum_{j=1}^{m} \sum_{i=1}^{n} W_{ij}(t)$$

where

$W_{set}(t)$ is the weight of a runtime dependency subgraph at point in time t.

$W_{ij}(t)$ is the weight of service required by a component i and provided through a component j at point in time t.

$$W_{ij}(t) = \frac{R_{ij}(t)}{O_{ij}(t)} \cdot I_{ij}(t)$$

$R_{ij}(t)$ is the number of requests on a service required from a component i and provided through a component j at point in time t.

$O_{ij}(t)$ is the amount of offers of a service required from a component i and provided through a component j at point in time t. Note that we consider only available, well functioning components, so $O_{ij}(t) \geq 1$.

I_{ij} is an importance of service required at point in time t by a component i and provided through a component j

$$I_{ij} \begin{cases} = 1 & \text{for regular services} \\ > 1 & \text{for important services} \end{cases}$$

Back to our example with a bank application. We can draw the static dependency graph and number the nodes sequentially (Fig. 7). The worst case during runtime of the system would be maximal request for all services $R_{ij_{max}}$ and a minimal offer of 1 for all O_{ij}. This would produce maximal weight of the system dependency graph:

$$W_{max} = W_{13_{max}} + W_{14_{max}} + W_{24_{max}} + W_{25_{max}} + W_{36_{max}} + W_{46_{max}} + W_{56_{max}} = R_{13_{max}} \cdot I_{13_{max}} + R_{14_{max}} \cdot I_{14_{max}} + R_{24_{max}} \cdot I_{24_{max}} + R_{25_{max}} \cdot I_{25_{max}} + R_{36_{max}} \cdot I_{36_{max}} + R_{46_{max}} \cdot I_{46_{max}} + R_{56_{max}} \cdot I_{56_{max}}$$

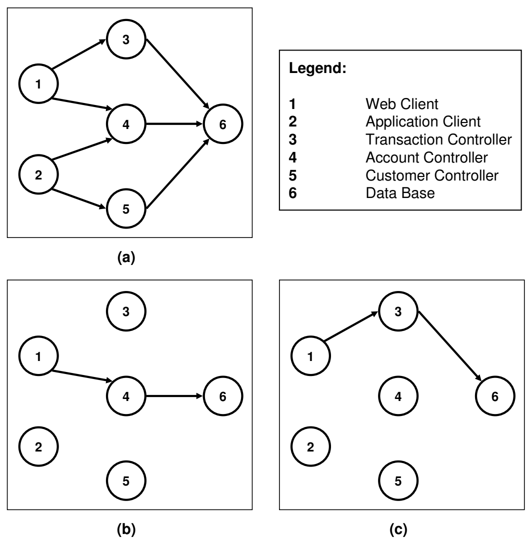

Figure 7. DependencyGraphs: (a) Static dependency graph, (b) Runtime dependency graph during steps 1,2,3 and 4 (c) Runtime dependency graph during steps 5 and 6

Considering the reconfiguration request to update the Account Controller and observing the withdraw sequence (Fig. 2), we notice that during steps 5 and 6 there are no dependencies from/to the Account Controller (Fig. 7 (c)), so we could update this component without affecting any other component and do not have to calculate any weights. But, if we get a request to exchange a component providing frequently used service(s) as usual for web applications, there would be probably no runtime dependency graph without an edge from/to this component. Even then, we can find a point in time providing maximal availability. Therefore we calculate the weight of the affected subsystem in the (1) worst case, (2) during steps 1 and 2, say at point in time t_1 and (3) during steps 3 and 4, at point in time t_2 (Fig. 7). To achieve this, we need additional information about the workload of each service, the maximal amount of simultaneous requests: $R_{ij_{max}}$ and the minimal amount of offered service $O_{ij_{min}}$. This information is usually part of a system workload specification, but it has to be refined for each service. Even if we don't have these values, it is possible to determine them using the usage profile. For particular points in time during runtime, there is a possibility to estimate this information through monitoring the running system and logging the requests and offered service instances. In our example, we assume an existence of the values in Table 2.

According to the scenario, we consider the services *getAccountOfCustomer* and *getData* at point in time t_1 and services *getDetails* and *getData* at point in time t_2. Hence, we can have different values for the same I_{jk} at different points in time. We can calculate following weights for the worst case, point in time t_1 and point in time t_1:

Variable	Worst Case	t_1	t_2
R_{14}	20	16	4
R_{46}	25	20	10
O_{14}	1	8	2
O_{46}	1	5	5
I_{14}	5	5	4
I_{46}	10	5	1

Table 2. Estimated Values

$$W_{set_{max}} = W_{14_{max}} + W_{46_{max}} = \frac{R_{14_{max}} \cdot I_{14_{max}}}{O_{14_{min}}} + \frac{R_{46_{max}} \cdot I_{46_{max}}}{O_{14_{min}}} = 350$$

$$W_{set_{t_1}} = W_{14_{t_1}} + W_{46_{t_1}} = \frac{R_{14_{t_1}} \cdot I_{14_{t_1}}}{O_{14_{t_1}}} + \frac{R_{46_{t_1}} \cdot I_{46_{t_1}}}{O_{14_{t_1}}} = 30$$

$$W_{set_{t_2}} = W_{14_{t_2}} + W_{46_{t_2}} = \frac{R_{14_{t_2}} \cdot I_{14_{t_2}}}{O_{14_{t_2}}} + \frac{R_{46_{t_2}} \cdot I_{46_{t_2}}}{O_{14_{t_2}}} = 10$$

Note that selecting the set of affected components heretofore increases the availability of the system if the set of affected components is a subset of a system. But, this is unfortunately not always the case. Considering the changing weight of edges during runtime, we can anyway achieve a higher availability. In this example we can determine two different dependency graphs, even if they have similar edges. This differentiation makes an additional optimisation of the availability possible through choosing the point in time t_2 for performing the requested reconfiguration.

3.4 Service Dependent Availability

We define availability similar to [30, 32, 4] as point availability:

Availability is readiness of a system/component for delivering a requested service at a particular point in time.

One could assume that the availability of a service required from a component i and provided through a component j, has a constant value A_{ij} depending only on its internal structure. But, taking a look at our example in Fig. **??**, one can recognise that this assumption is very naive. The Web Client service *withdrawAction* call other external services (*getAccountOfCustomer, getDetails and withdraw*). Thus, the availability of each of them influences the availability of the service *withdrawAction*. Furthermore, we have to consider the execution environment (e.g. availability of connections) as an additional factor. Putting them all together, we can define the availability of a service required from a component i and provided through a component j as follows:

$$A_{ij} = A_{ij_{int}} \cdot A_{ij_{ext}} \cdot A_{ij_{env}}$$

where

$A_{ij_{int}}$ is the availability of the internal structure of the service.

$A_{ij_{ext}}$ is the availability of the external services call by the service.

$A_{ij_{env}}$ is the availability of the environment relevant for the service.

Our approach does not focus on prediction or determination of these values. We rather include the explained dependencies to our availability relation and consider using of already determined values by prediction approaches [27, 20].

For a particular reconfiguration request we can determine a subsystem including a set of affected components. The availability of this subsystem would then be:

$$A_{sub} = \frac{\sum_{j=1}^{m} \sum_{i=1}^{n} A_{ij}}{m \cdot n}$$

Considering the fact that (1) a system/component can provide more than one service, (2) they can have different importance, and (3) there is a varying number of requests on a particular service at a particular point in time t, we can identify following relation between availability of services and runtime dependency graphs:

$$A_{sub}(t) = \frac{1}{W_{ij}(t)} \cdot A_{sub}$$

It is obvious that a minimal weight would lead to a maximal possible availability for a particular subsystem.

4 Platform Independent Reconfiguration Manager PIRMA

As a proof of our theoretical concept [16] we developed a system called Platform Independent Reconfiguration Manager PIRMA (Fig. 8) [18]. The current implementation is an Eclipse Application, based on Java EE [31] and uses the JBoss Application Server [12]. It consists of the following four top-level components: *Reconfiguration Analyser*, *Dependency Manager*, *Transaction Manager* and *Reconfigurator* .

Our approach concentrates on analysing and performing a reconfiguration request, rather then investigating its origin. Our system PIRMA processes reconfiguration requests coming from an external system (e.g. Fault Detection Systems). It first analyses the external reconfiguration request and generates an XML request containing all requested actions and a set of affected components for each action. An action can be (1) add, (2) delete or (3) update. For each

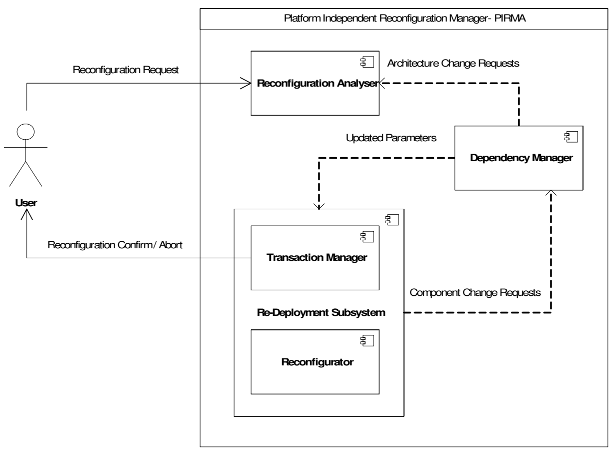

Figure 8. Platform Independent Reconfiguration Manager-PIRMA

component to be reconfigured, we can generate an appropriate *request.xml* containing information about the component and the requested action. This data is then inserted into the *ejb-jar.xml* deployment descriptor which is then opened in an integrated XML editor for further modifications. When the user has finished editing the deployment meta data for a particular module, the deployment content is packaged into a module archive and transferred to the target server system for deployment.

The performing of reconfiguration is realised as a transaction [15] called *controlled runtime redeployment* through our Redeployment Subsystem [19]. It presents an extension of the concepts of hot deployment and dynamic reloading supported by the WebSphere Application Server [11]. Both are well-suited for development and testing, but pose unacceptable risks to production environments. Our concept of controlled runtime redeployment implements the J2EE Deployment API [29] and extends it by specifying redeployment to be transparent to users thus allowing it to be used in productive systems. Our system dynamically determines a redeployment set as a transitive closure of all currently referenced components and the sets redeployment points due to achieve a transparent redeployment. Having found these, it establishes a synchronization barrier that allows suspending execution in the redeployment set. All outstanding invocations that started a new transaction [31] can be completed, while new invocations have to wait on this synchronization barrier. The suspended target component archive shall then be exchanged and recreated together with all other modules from their redeployment set. Finally, the synchronization barrier will be released so that any invocations that may have been suspended can continue to execute the new code. This redeployment transition has the following key attributes:

- Handle transparency: A redeployment transition is not interruptible by executions of the targets module codes. This means, no handle that is used by clients to communicate with components contained in the target module, will ever fail due to the module being unavailable.

- Weak consistency: Upon successful completion, the target module is in an execution-ready, deployed state. If the deployment of the module fails due to invalid meta-data declarations, the transition results in a stopped module.

For more details see [19, 21].

5 Related Work

Runtime reconfiguration is a very active research area in various disciplines of computer science. Protocol-based reconfiguration approaches [24, 1, 13] work on dynamic component updates. They consider contractually defined components with behaviour-specifying interfaces [25] for checking consistency and interoperability, but don't use the runtime state and the interaction protocols for restricting runtime dependencies among instances of components.

Architecture-based reconfiguration approach [3] a meta-framework called 'Plastik' supports the specification and creation of runtime component-framework-based software systems and facilitates and manages the runtime reconfiguration of such systems while ensuring integrity across changes is proposed. Runtime dependencies are considered for ensuring consistency, but not for increasing availability. At [22, 23], the runtime reconfiguration is basically a replacement of single components at architectural level, but no runtime dependencies are addressed. Structural changes are performed by checking and altering connector bindings.

Graph-transformation reconfiguration approaches [33, 8, 9] define possible architecture changes as a graph-transformation. Their focus is on correct definition of possible reconfigurations, rather then increasing availability during the process of reconfiguration.

Runtime redeployment approaches [28], [5] cover the technical aspects of redeployment and are thus similar to our basic system, but they consider no architectural changes of the system and use no runtime dependencies among instances of components for additional increasing of availability.

Our approach aims at optimising of the process of runtime reconfiguration. We consider the usage model of the system and component protocol information, with the explicit goal of increasing the availability of the system through maximising the set of available services during runtime reconfiguration. An additional advantage of our approach is the possibility of extending/combining each step of the reconfiguration process with other approaches. For example, including more consistency checks, by comparing protocols or determining if performing of a particular reconfiguration request would result in a consistent architecture using graph transformations.

6 Summary and Further Work

In this paper we presented a new approach to increasing service availability during runtime reconfiguration of component-based systems. On a given reconfiguration request, by exploiting additional information concerning the system usage model and service-based behaviour protocol of components, we can determine an optimal system runtime state for performing the requested reconfiguration. We briefly represent the system architecture of our Platform Independent Reconfiguration Manager PIRMA, which has been implemented based on Java EE, using JBoss and Eclipse as a basic version. Furthermore, we presented conceptual extension/optimisation of the dependency manager of our system, which establishes maximisation of the service availability:

For a particular reconfiguration request we can compare the set of affected components (a transitive closure including components to be exchanged and dependent ones) with each set of participating components corresponding to a sequence of a usage scenario. If there is an interception, then we identify a relevant scenario. In case of no interception, we can ignore the sequence. For selected scenarios we can compute all possible runtime dependency graphs of the participating subsystem. Using the service effect automata we can exclude past dependencies and later future ones. This way we can determine the minimal runtime dependency graph (see Section 3.3) and the corresponding minimal redeployment set and locate system runtime states which are convenient for performing the reconfiguration according to a given request. Next, we have to identify those states by monitoring the running system and start the reconfiguration.

At this point we do not consider reconfiguration duration as a critical factor as we perform the reconfiguration request sequentially. Furthermore, our redeployment subsystem dynamically determines the redeployment set to avoid possible inconsistency caused by incorrectly determined runtime states or their faulty recognition by monitoring. This way we can assure a safe reconfiguration with a maximal possible availability. Currently, we work on evaluation of our theoretical concept for determination of minimal runtime dependency graphs. We simulate normal user behaviour for example systems and analyse monitored data due to create the appropriate service effect automata and determine the point in time with a minimal dependency graph for particular reconfiguration requests. Next step would be considering transition probabilities in our usage model for additional optimisation of availability. As a future work we consider to integrate the extension of the dependency manager presented in this paper in our System PIRMA.

References

[1] R. Allen, D. Garlan, and R. Douence. Specifying dynamism in software architectures. In *Proceedings of the Workshop on Foundations of Component-Based Software Engineering*, Zurich, Switzerland, September 1997.

[2] N. Arshad, D. Heimbigner, and A. L. Wolf. A planning based approach to failure recovery in distributed systems. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 8–12, New York, NY, USA, 2004. ACM Press.

[3] T. V. Batista, A. Joolia, and G. Coulson. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2nd European Workshop on Software Architecture, EWSA 2005*, pages 1–17, 2005.

[4] S. Becker, W. Hasselbring, A. Paul, M. Boskovic, H. Koziolek, J. Ploski, A. Dhama, H. Lipskoch, M. Rohr, D. Winteler, S. Giesecke, R. Meyer, M. Swaminathan, J. Happe, M. Muhle, and T. Warns. Trustworthy software systems: A discussion of basic concepts and terminology. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–18, 2006.

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

[6] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[7] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[8] L. Grunske. Automated software architecture evolution with hypergraph transformation. In *Proceedings of the 7th International IASTED on Conference Software Engineering and Application*, Nov 2003.

[9] L. Grunske. Formalizing architectural refactorings as graph transformation systems. In *Proceedings of the Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD05)*, volume 00, pages 324–329, Towson, Maryland, USA, May 2005. IEEE Computer Society.

[10] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[11] IBM, http://publib.boulder.ibm.com/infocenter/wsdoc400/. *WebSphere Application Server Documentation, Version 6*, 2007.

[12] JBoss Group, http://www.jboss.org/docs/index. *JBoss Application Server Documentation*. Retrieved 2007-03-20.

[13] F. Kon and R. H. Campbell. Dependence management in component-based distributed systems. *IEEE Concurrency*, 8(1):26–36, Jan. 2000.

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

[15] M. Little, J. Maron, and G. Pavlik. *Java Transaction Processing: Design and Implementation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2004.

[16] J. Matevska-Meyer and W. 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, Feb. 2003. University of Groningen.

[17] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Exploiting protocol information for speeding up runtime reconfiguration of component-based systems. In *Proceedings of Workshop on Component-Oriented Programming WCOP 2003*, Darmstadt, Germany, July 2003. Technical University of Darmstadt.

[18] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software architecture description supporting component deployment and system runtime reconfiguration. In *Proceedings of Workshop on Component-Oriented Programming WCOP 2004*, Oslo, Norway, June 2004. University of Oslo.

[19] J. Matevska-Meyer, S. Olliges, and W. Hasselbring. Runtime reconfiguration of J2EE applications. In *Proceedings of DECOR04 - 1st French Conference on Software Deployment and (Re) Configuration*, pages 77–84, Grenoble, France, Oct. 2004. University of Grenoble.

[20] J. D. Musa. *Software Reliability Engineering: More Reliable Software Faster And Cheaper*. Authorhouse, 2004.

[21] S. Olliges. Runtime Reconfiguration in J2EE Systems. Master Thesis supervised by Jasminka Matevska and Wilhelm Hasselbring, University of Oldenburg, Germany, Department of Computing Science, Software Engineering Group, Dez 2005.

[22] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177–186, Apr. 1998.

[23] P. Oreizy and R. 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.

[24] 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, Mar. 1998.

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

[26] R. H. Reussner. Parameterised contracts for software components. Technical report, DSTC Pty Inc., Melbourne, Australia, Apr. 2002.

[27] R. H. Reussner, H. W. Schmidt, and I. H. Poernomo. Reliability prediction for component-based software architectures. *J. Syst. Softw.*, 66(3):241–252, 2003.

[28] M. J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A. L. Wolf. Reconfiguration in the Enterprise JavaBean component model. In J. Bishop, editor, *Proceedings of IFIP/ACM Working Conference on Component Deployment*, pages 67–81, Berlin, Germany, June 2002. Springer-Verlag Berlin Heidelberg.

[29] R. Searls. *J2EE Deployment API Specification*. Sun Microsystems, http://java.sun.com/j2ee/tools/deployment/, Nov. 2003. Retrieved 2007-03-13.

[30] I. Sommerville. *Software Engineering*. Addison-Wesley, 2007.

[31] Sun Microsystems, http://java.sun.com/javaee/5/docs/tutorial/. *The Java EE 5 Tutorial*, 2006.

[32] K. S. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. pub-PH, pub-PH:adr, 1982.

[33] M. Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nova de Lisboa, 1999.