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.
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](image1)

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 affected 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).

![Figure 2. Withdraw Success](image2)

Table 1. Provided and Required Services

<table>
<thead>
<tr>
<th>Component</th>
<th>Transaction Controller</th>
<th>Account Controller</th>
<th>Customer Controller</th>
<th>Web Client</th>
<th>Application Client</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provided Services</td>
<td>withdraw</td>
<td>createAccount</td>
<td>createCustomer</td>
<td>withdrawAction</td>
<td>createAccountAction</td>
</tr>
<tr>
<td></td>
<td>deposit</td>
<td>removeAccount</td>
<td>removeCustomer</td>
<td>removeAccountAction</td>
<td>removeCustomerAction</td>
</tr>
<tr>
<td></td>
<td>makeCharge</td>
<td>makePayment</td>
<td>makePaymentAction</td>
<td>makePaymentAction</td>
<td>makePaymentAction</td>
</tr>
<tr>
<td></td>
<td>transferFunds</td>
<td>transferFunds</td>
<td>getAccountOfCustomer</td>
<td>getAccountOfCustomerAction</td>
<td>getAccountOfCustomerAction</td>
</tr>
<tr>
<td></td>
<td>DataTable</td>
<td>getDetails</td>
<td>getDetails</td>
<td>getDetails</td>
<td>getDetails</td>
</tr>
<tr>
<td>Required Services</td>
<td>getData</td>
<td>updateData</td>
<td>getData</td>
<td>updateData</td>
<td>getData</td>
</tr>
<tr>
<td></td>
<td>uploadData</td>
<td>updateData</td>
<td>updateData</td>
<td>updateData</td>
<td>updateData</td>
</tr>
</tbody>
</table>

![Table 1](image3)
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 fulfill 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 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 3. Usage Meta Model

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, if A uses/requires services provided by B (depicted in Fig. 5).

![Figure 5. A is dependent on B](image)

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 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](image)

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:

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

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 $t$.
- $R_{ij}(t)$ 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 $j$.

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

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:

$$W_{max} = W_{13} \cdot I_{13} + W_{14} \cdot I_{14} + W_{24} \cdot I_{24} + W_{25} \cdot I_{25} + W_{36} \cdot I_{36}$$

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

$$+ R_{46} \cdot I_{46} + R_{56} \cdot I_{56}$$
We define availability similar to [30, 32, 4] as point availability:

\[
\text{Availability is readiness of a system/component for delivering a requested service at a particular point in time.}
\]

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. 7, one can recognise that this assumption is very naive. The Web Client service \( \text{withdrawAction} \) call other external services (\( \text{getAccountOfCustomer}, \text{getDetails} \) and \( \text{withdraw} \)). Thus, the availability of each of them influences the availability of the service \( \text{withdrawAction} \). Furthermore, we have to consider the execution environment (e.g. availability of connections) as an additional factor. Putting them all

\[
W_{\text{set}_{\max}} = W_{14_{\max}} + W_{46_{\max}} = \frac{R_{14_{\max}} \cdot I_{14_{\max}}}{O_{14_{\min}}} + \frac{R_{46_{\max}} \cdot I_{46_{\max}}}{O_{44_{\min}}} = 350
\]

\[
W_{\text{set}_{1,1}} = W_{14_{t_1}} + W_{46_{t_1}} = \frac{R_{14_{t_1}} \cdot I_{14_{t_1}}}{O_{14_{t_1}}} + \frac{R_{46_{t_1}} \cdot I_{46_{t_1}}}{O_{44_{t_1}}} = 30
\]

\[
W_{\text{set}_{2,2}} = W_{14_{t_2}} + W_{46_{t_2}} = \frac{R_{14_{t_2}} \cdot I_{14_{t_2}}}{O_{14_{t_2}}} + \frac{R_{46_{t_2}} \cdot I_{46_{t_2}}}{O_{44_{t_2}}} = 10
\]
together, we can define the availability of a service required from a component $i$ and provided through a component $j$ as follows:

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

where

- $A_{ij_{\text{int}}}$ is the availability of the internal structure of the service.
- $A_{ij_{\text{ext}}}$ is the availability of the external services call by the service.
- $A_{ij_{\text{env}}}$ 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:

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

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:

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

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 than 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 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.

![Figure 8. Platform Independent Reconfiguration Manager-PIRMA](image.png)
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 metaframework 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


