Model-based Runtime Reconfiguration of Component-based Systems

Jasminka Matevska
Software Engineering Group
University of Oldenburg, Germany
jasminka.matevska@informatik.uni-oldenburg.de

ABSTRACT

Software 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 of increasing importance especially for business-critical systems. Runtime reconfiguration supports evolution of systems while maintaining availability of services they provide. This paper presents a model-based approach to runtime reconfiguration of component-based systems, which aims at minimising the interference caused by the reconfiguration and thus maximising system responsiveness during reconfiguration.

1. INTRODUCTION

Runtime reconfiguration supports evolution of systems while maintaining availability of services they provide. The process of reconfiguration usually 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 runtime reconfiguration approaches with different focus on a particular step and thus aiming at different goals.

Safe redeployment approaches cover the technical aspects of runtime reconfiguration. One of the oldest approaches to managing a reconfiguration as a transaction is described by Kramer and Magee [2].

Component-based approaches like SOFA/DCUP [8] consider contractually defined components with behaviour specifying interfaces for checking consistency and interoperability. Runtime dependencies are considered for ensuring consistency, but not for maximising service responsiveness.

Architecture-based approaches following the route described by Oreizy, Medvidovic and Taylor [7] define the runtime reconfiguration basically as a replacement of a single component at architectural level. Structural changes are performed by checking and altering connector bindings.

Our research approach, called AVORR (*Availability Optimized Runtime Reconfiguration*) [3] aims at maintaining continuous availability while maximising service responsiveness during runtime reconfiguration of component-based systems. In order to achieve our goal, an additional analysis of the running system becomes necessary. A model describing only a static view of the system does not provide a sufficient information. We propose an extension of the model defining the runtime view of the system. It supports dynamic configurations and a dynamic component and system behaviour description in order to enable checking consistency of the system at runtime. For determination and isolation of the minimal set of affected components an appropriate description of runtime dependencies among components is necessary [5]. Furthermore, the approach defines a detailed concept for the technical accomplishment of the reconfiguration as a *runtime redeployment transaction* [6] in order to provide full availability of the system during reconfiguration. Finally, for minimising system interference and maximising service responsiveness, a more differentiated analysis of *runtime dependency graphs* [4] is included.

The reconfiguration process in AVORR consists of the following steps:

(1) Analysis of the reconfiguration request,

(2) Identification of a minimal set of affected components,

(3) Recognition of the appropriate point in time at runtime for a reconfiguration start, and

(4) Execution of the requested reconfiguration as a runtime redeployment transaction.

2. OUR C3 META MODEL

Our Component-Connector-Container (C3) Meta Model illustrated in fig. 1 defines a *component* as a first-class entity and derives *static component* for the static view and *live component* for the dynamic view.

- **Static (Structural) View:** A *static component* including *primitive* and *composite components* enables building a hierarchical system architecture based on a GoF-Pattern *Composite* [1].
- **Dynamic (Runtime) View:** A *live component* has two specialisations *primitive live component* and *container component* and presents the runtime view of the system. The primitive component description instantiates both specialisations of the primitive live component: *service component* or *connector component.*

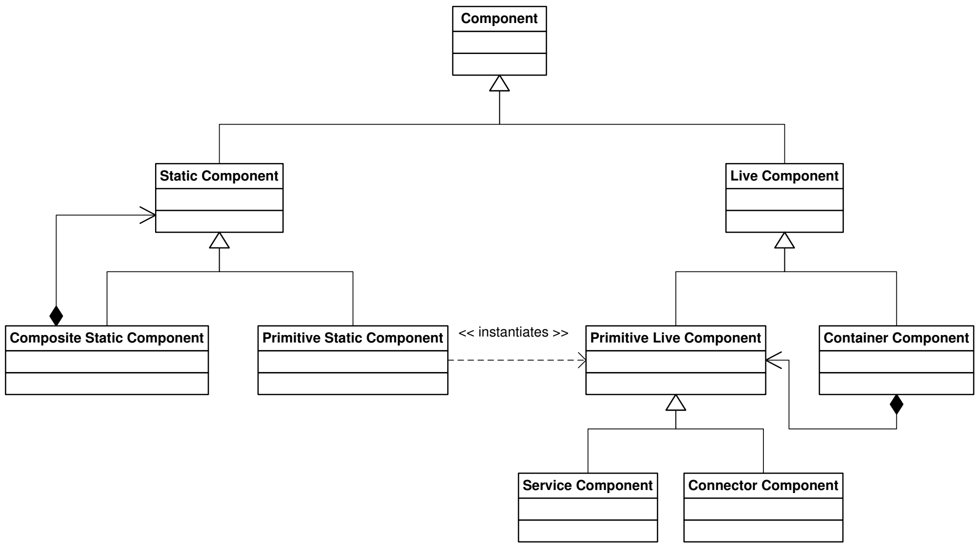

Figure 1: Our C3 Meta Model

The connector component is used to describe both *uses* and *interaction* dependencies among live components. Container components are essential extensions to establish modelling of deployment and runtime properties of the system. They host primitive live components without considering hierarchical aspects. We describe the component and system runtime behaviour using automata.

3. COMPONENT BEHAVIOUR 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 [9]. 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. By analysing SEFFs we can exclude past and late future dependencies for each component. This can be used at runtime for recognition of the convenient runtime state as a start point in time for the requested reconfiguration.

4. SYSTEM BEHAVIOUR SPECIFICATION

We describe the system behaviour as a set of finite automata. Each automata describes a use case of the system. Each state sequence describes a possible execution sequence of the use case. Each state transition of the SEFF induces a state transition of the system automata. The frequency of occurrence of each execution sequence depends on the business logic of the system and the usage behaviour of system users and is included in the usage model of the system.

5. 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. Currently, we model sequences with UML 2 sequence diagrams. An extension of the behaviour definition including transition probabilities (e.g. as a Markov chain model [11]) is being used to achieve a weighting of sequences and thus reducing the number of transition states considered as relevant for the determination of minimal runtime dependencies for a particular reconfiguration request. We omit analysis of very improbable sequences, because waiting for them (even if they would produce minimal runtime dependencies) would delay the completion of the reconfiguration. This way we speed up both the analysis and the reconfiguration.

6. RUNTIME DEPENDENCY GRAPHS

Each software system can be considered a composition of communicating components. 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. Dependencies among components can be described using weighted directed dependency graphs. Each component is assigned a node and each dependency relation an edge. The edge is directed from a component using (calling) a particular service to the component providing that service. For the illustration, we extend UML Component Diagrams by including labelled dependencies. A static dependency graph contains all possible dependencies among components within a system. The runtime dependency graph can be considered as a subset of the static dependency graph.

For each 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 importance at a particular point in time. Considering the set of affected component, relevant scenarios based on the usage model can be recognised [4]. During runtime a system executes different scenarios and thus activates particular instances of components. The runtime dependency graph among instances of components at a particular runtime state during the corresponding time period contains a subset of all possible dependencies of a system. Considering the possibility of having a multi-user system, we can observe a strong varying usage and thus varying dependencies among instances of components. Using SEFFs we can analyse the past and future behaviour of a component and its dependencies. For each scenario being executed we can determine corresponding runtime dependency graphs. Through additional analysis of changing dependencies among instances of components during runtime considering different scenarios from usage model of the application, we can determine a minimal runtime dependency graph for obtaining maximal possible responsiveness. Finally, we use the system behaviour model in order to assign the determined minimal dependency graph to a corresponding system runtime state [4]. After the analysis, the running system is being rescanned in order to detect the determined state. At that point in time the reconfiguration is started. Performing the requested reconfiguration within a time interval with minimal dependencies maximizes the availability of the system.

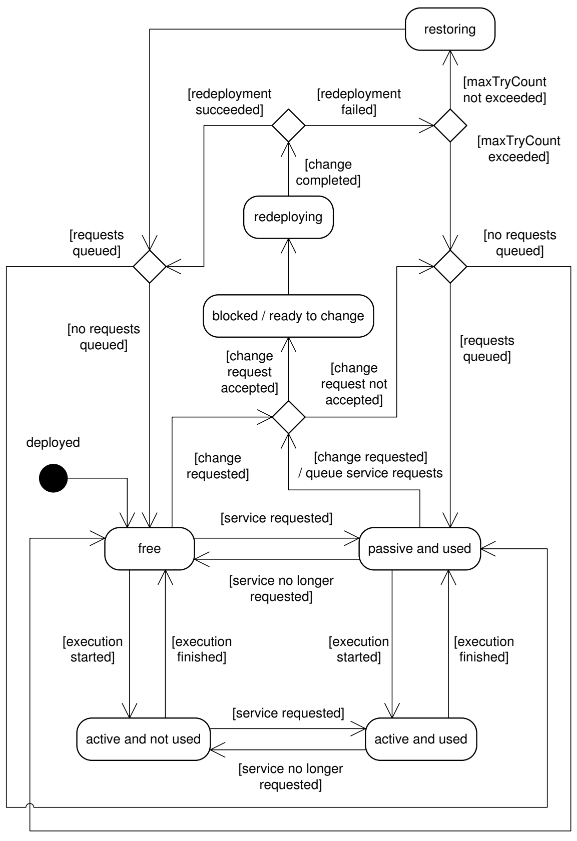

Figure 2: Component Life Cycle Protocol

7. RECONFIGURATION TRANSACTION

The completion of the reconfiguration as a *runtime redeployment transaction* is enabled under the assumption that each component adheres to the life cycle protocol illustrated in fig. 2. A component is *free (not active and not used)* 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*, which means it is suspended and all incoming invocations are queued. Only *passive (free or passive and used)* components can be blocked. The corresponding protocol on system level can be found in fig. 3.

To avoid inconsistency caused by deviation among analysed and actual runtime dependencies, a redeployment set as a transitive closure of all currently referenced components is dynamically determined and a synchronisation barrier is established. All outstanding invocations can be completed, while new invocations are suspended and have to wait on this synchronization barrier. The affected components are transferred into a *blocked* state and the changes are applied. After the change transaction is being performed, all *blocked* components are transferred back into a *free* or *passive and used* state respectively, and the synchronization barrier is released so that suspended invocations can continue execution.

8. EVALUATION

Java EE-based Implementation.

The proposed architecture of our basic system called *Platform Independent Reconfiguration Manager PIRMA* consists of the following four top-level components: *Reconfiguration Analyser*, *Dependency Manager*, *Transaction Manager* and *Reconfigurator*.

The Reconfiguration Analyser takes a *reconfiguration request*, analyses and classifies the requested change. The components directly concerned with the request are identified. The Dependency Manager acts according to the *Observer* GoF-Pattern [1]. It monitors the runtime dependencies among components, determines a minimal set of change-affected components as a transitive closure of all currently referenced components and sends a *change request* for each involved component to the Reconfigurator. The Transaction Manager and the Reconfigurator build the *Redeployment Subsystem*. They realise the reconfiguration as a dependent change transaction [2].

As a proof of our theoretical concept an Eclipse Application, based on J2EE and using the JBoss Application Server was developed and ported to Java EE [6].

Scenario-based Dependency Analysis.

The evaluation of our dependency analysis follows our scenario-based approach. For generating and analysing the monitoring data the Java-based monitoring and visualiza-

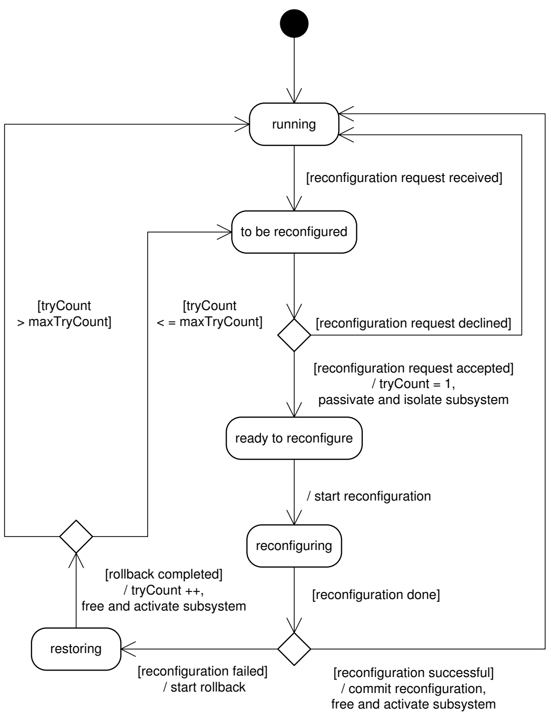
running
[reconfiguration request received]
to be reconfigured
[tryCount > maxTryCount]
[tryCount < = maxTryCount]
[reconfiguration request declined]
[reconfiguration request accepted] / tryCount = 1, passivate and isolate subsystem
ready to reconfigure
/ start reconfiguration
reconfiguring
[rollback completed] / tryCount ++, free and activate subsystem
[reconfiguration done]
restoring
[reconfiguration failed] / start rollback
[reconfiguration successful] / commit reconfiguration, free and activate subsystem

Figure 3: System Life Cycle Protocol

tion tool Kieker [10] was used and extended. As an example application the iBatis JPetStore[1], a Java-implemented multi-user Web application that represents an online shopping store was used. For simulating probabilistic system usage based on Markov chains, an extension of Apache JMeter, Markov4JMeter[2] was used. As a second step, we used the results of the dependency analysis at rescanning of the running system in order to recognise the optimal point in time for a reconfiguration start.

9. CURRENT STATUS

The actual implementation of our reconfiguration manager confirmed the feasibility of our project using existing component technologies. The analysis of monitored data showed that for every component reconfiguration request it is possible to find a point in time with a minimal runtime dependency graph, even if a system has many concurrently active users. However, with increasing amount of active users the duration of the detected convenient time periods decreases. Having a limited waiting period from a reconfiguration request arrival to a reconfiguration execution, it gets more difficult to hit one of them. This makes a more differentiated analysis of runtime dependency graphs, as followed by this approach, necessary [4]. Currently, we are working on integration of the scenario-based dependency analysis into our reconfiguration manager system in order to evaluate the complete approach through a reasonable amount of experiments. As an example Java EE based system, we use the Sun Duke's Bank Application [3].

[1]http://ibatis.apache.org/
[2]http://markov4jmeter.sourceforge.net/
[3]http://java.sun.com/javaee/5/docs/tutorial/

10. REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Object-Oriented Technology. Addison-Wesley, 1995.

[2] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. on Software Engineering*, 16(11):1293–1306, 1990.

[3] J. Matevska. An optimised Runtime Reconfiguration of component-based Software Systems. In *Proc. of the 32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2008)*, pages 499–501. IEEE, 2008.

[4] J. Matevska and W. Hasselbring. A Scenario-based Approach to Increasing Service Availability at Runtime Reconfiguration of Component-based Systems. In *Proc. of 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA'07)*, pages 137–144. IEEE, 2007.

[5] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of Workshop on Component-Oriented Programming (WCOP'04)*, 2004.

[6] J. Matevska-Meyer, S. Olliges, and W. Hasselbring. Runtime Reconfiguration of J2EE Applications. In *Proc. of the 1st French Conference on Software Deployment and (Re) Configuration (DECOR'04)*, pages 77–84. Net Print, Eybens, 2004.

[7] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proc. of the International Conference on Software Engineering 1998 (ICSE'98)*, pages 177–186. ACM, 1998.

[8] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. In *Proc. of International Conference on Configurable Distributed Systems*, pages 35–42. IEEE, 1998.

[9] R. H. Reussner. Automatic Component Protocol Adaptation with the CoCoNut Tool Suite. *Future Generation Computer Systems*, 19(5):627–639, 2003.

[10] M. Rohr, A. van Hoorn, J. Matevska, L. Stoever, N. Sommer, S. Giesecke, and W. Hasselbring. Kieker: Continuous Monitoring and on demand Visualization of Java Software Behavior. In *Proc. of the IASTED International Conference on Software Engineering.* ACTA Press, 2008.

[11] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating Probabilistic and Intensity-varying Workload for Web-based Software Systems. In *SPEC International Performance Evaluation Workshop (SIPEW '08)*, volume 5119 of *Lecture Notes in Computer Science (LNCS)*, pages 124–143. Springer, 2008.