Runtime Adaptation in a Service-Oriented Component Model

Florian Irmert
University of Erlangen-Nuremberg
Department of Computer Science
Computer Science 6 (Data Management)
Martensstrasse 3
91058 Erlangen, Germany
florian.irmert@cs.fau.de

Thomas Fischer
University of Erlangen-Nuremberg
Department of Computer Science
Computer Science 6 (Data Management)
Martensstrasse 3
91058 Erlangen, Germany
k.p.t.fischer@gmx.de

Klaus Meyer-Wegener
University of Erlangen-Nuremberg
Department of Computer Science
Computer Science 6 (Data Management)
Martensstrasse 3
91058 Erlangen, Germany
kmw@cs.fau.de

ABSTRACT

Developing software applications which manage, optimize or adapt themselves at runtime requires an architecture which provides adaptation of software components at runtime. An architecture model that has gained a lot of attention in recent years is SOA (service-oriented architecture). In a SOA environment services as well as applications build up complex dependencies. Therefore it is crucial for self-managing SOA applications to adapt services at runtime without interference of the application execution and the service availability. In this paper, we discuss the problems arising from the requirement of runtime adaptation and present our solution by replacing service implementations at execution time in a service-oriented component model. For a seamless integration we strive for a transparent and atomic replacement of a service implementation in respect to the other services/applications.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Life cycle*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Adaptation, service-oriented architecture, component replacement, modularity, migration

1. INTRODUCTION

In recent years a new architectural style, the so-called "service-oriented architecture" (SOA) [19, 10, 25] has become very popular. In SOA communication is performed by services and applications are composed of existing services. Thereby services are loosely coupled and describe their functionality within a service description. They are published in a service registry and applications search the registry for adequate services. A characteristic in such a loosely coupled environment is the dynamic arrival and departure of services at any time. A departing service can affect the availability of the applications which depend on that service. A growing number of companies are implementing their software applications upon service oriented architecture, e.g. to lower their integration costs and to improve the time to market. But like traditional middleware [22], a SOA application is limited in its ability to support adaptation. Two types of adaptation can be identified: static and dynamic. To develop software applications which manage, optimize or adapt themselves at runtime, dynamic adaptation (runtime adaptation) is needed as a foundation.

In this paper we introduce the CoBRA (Component Based Runtime Adaptable) architecture which enables dynamic adaptation by using a service-oriented component model [6]. Within the CoBRA architecture an application may be adapted without the need to restart or shutdown the application. Therefore the adaptation includes an exchange of the implementation, which is atomic for the underlying application, as well as a mechanism for state preservation of the adapted service.

The rest of the paper is organized as follows. In section 2 we introduce SOA. The requirements and challenges of service adaptation are discussed in section 3. The CoBRA project, which is addressing the service adaptation challenges, is presented in section 4. Section 5 presents related work, followed by future work and a conclusion in section 6.

2. SOA

In a service-oriented architecture (SOA) new applications are assembled from existing services. Services are reusable units, which can be used by applications or other services

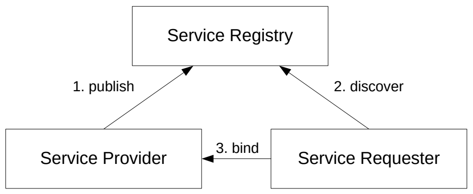

Figure 1: Service registry

without the need to know the details of their implementation.

Following [17], the definition of a service is published in the *service description*. The service description consists at least of the *service interface*, but additional information regarding functional and non-functional properties is desirable. The service description is used to select an appropriate *service provider*. Service providers offer implementations for a service and publish their service description in a *service registry*. A *service requester* can search the registry and choose an adequate service provider for a specific service description. Figure 1 shows the interrelationship of service provider, service requester and service registry.

In a SOA environment services can be added or removed at any time. The application must be aware of that behaviour and perhaps rebind a new service provider at runtime. Often releasing a service and binding a new one is supported by the environment. For example an event system can inform all depending applications and services if a service departs or arrives. This implies the service requester to handle the following events:

- *Arrival of a new service.* The arrival of a new service may cause an application (service requester) to bind the new service if the service description fits (better) to the requirements of the application.
- *Departure of an existing service.* The application can try to switch to another service. If non appropriate service is available in the registry, the application may not be able to work properly.

The fact that a service can arrive or depart at any time obviously implies a lot of custom coding, e.g. the error handling in the case no adequate alternative is available for a departing service.

2.1 Service availability

Founded on the dynamic behaviour of services, their availability is a matter of concern. In a SOA scenario multiple services depend on each other and the availability of one service often affects different applications. Therefore a definition of *service availability* for a SOA service may be given: A service is available as long as its service description stays valid and trackable by the service registry. A service description is valid as long as it is usable by the service requester.

The service availability may be harmed by service dependency issues as well as by update events for the service itself. Service dependency in this context poses a problem. If a service is changed, all depending services may not work properly. The core challenge is therefore to update or change a service at runtime without affecting other services. To solve this problem, we discuss the two major *service adaptation* concepts in section 3: One employs dynamic AOP to add or modify the behaviour of services at runtime. The other approach replaces service implementations at runtime.

Defining the addressed problem, we first have to outline service updates and their pitfalls at runtime.

2.2 Service update

To "update" a service in a SOA environment usually a new version of the same service provider (with the same service description) is published in the registry and afterwards the old version is removed. Therefore switching to another implementation is only possible if the application releases the service reference after its invocation. During the next request to the service registry the new version of the service can be bound.

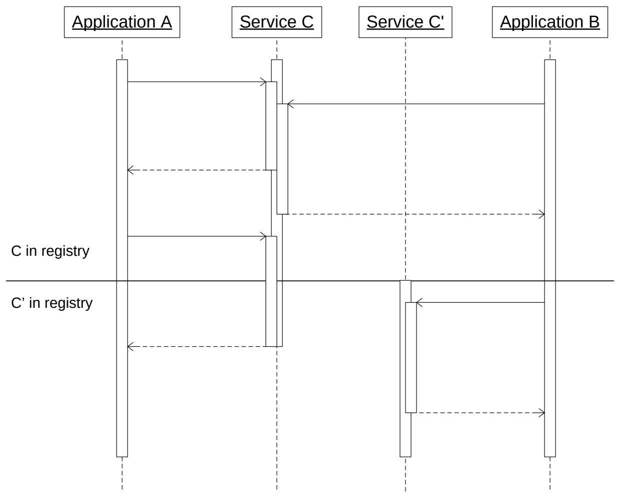

Figure 2: Example: classical service replacement

In the example in figure 2 service C is subject to replacement by service C'. Therefore service C is replaced in the registry with service C'. Obviously, application A still has a reference to the service object of service C, although service C is not available in the registry any more. For a certain time application A and B do not work with the same implementation of service C and hence during this time interval applications A and B may deliver different results.

In the previous example we assumed that C is still available, although it is not registered any more. This behaviour is caused for instance by asynchronous garbage collection. To avoid such a situation service C can be eliminated at the same time its service description is removed from the registry. If it is eliminated synchronously during the method invocation of application A, an error will occur, which has to be handled by application A.

Another approach is the usage of an event system: In this case application A and B are notified by an event on arrival of service C', which causes the change from service C to C'. However, this behaviour does not eliminate the need for error-handling if the event occurs during a method invocation on service C. Furthermore, an event system can not avoid the situation depicted in figure 2 completely. Application B can already be notified of the service update and therefore already use service C', while application A, which has not yet received the update event, still works on service C.

In conclusion, regardless which approach is used for a service update, the possibility for ambiguity of a service in a certain time period remains.

3. SERVICE ADAPTATION

As explained in the last section it is challenging to add new functionality or change the functionality of a service in a SOA environment at runtime without any side effects. The emphasis in this paper lies on the exclusion of side effects like the illustrated update problem in section 2.2. For continuous execution during and after an adaptation process it is crucial to consider state transition and synchronisation issues as core challenges.

There exist several approaches to enable adaptation, mostly based on component models. They all may be categorized in three mayor approaches:

- One possibility is the use of dynamic AOP (d-AOP) or delegation models to apply modifications at runtime. Representatives of this technique are [12, 14, 21]
- Another approach is to use fractal components like [5], where components are compositions of subcomponents, which are target of adaptation by replacement.
- The third category adapts a component by replacement of its implementation, while the interface stays valid. [4, 16, 15, 3] proposed concepts realizing this approach.

The latter two approaches exhibit a large similarity in respect to the adaptation process. While the fractal model adapts a component by replacing small subcomponents behind a common interface, the last approach replaces the whole component behind the interface. Therefore the arising prerequisites, which have to be met to enable runtime adaptation, are the same, but in another granularity.

To enable runtime adaptation in a SOA environment, we have tested two different concepts: We have integrated a d-AOP framework in SOA environment to check the handling with respect to performance and maintenance. The results are presented in the next section. Afterwards we introduce our current solution for the adaptation by replacement of whole service implementations at runtime.

3.1 Dynamic adaptation with d-AOP

Previtali [21] exploits aspects for dynamic updates and employs AOP's (aspect-oriented programming) features like method or field interception. With dynamic AOP (d-AOP) these modifications can be applied at runtime. The term "dynamic aspect-oriented programming" is most commonly

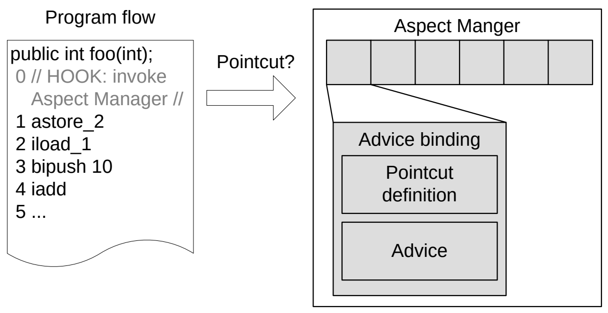

Figure 3: dynamic AOP

used if aspects can be deployed and activated at runtime. Dynamic AOP can be realized e.g. with a modified JVM [20] or bytecode modification [26]. The integration of d-AOP into a SOA framework (as we have demonstrated in [12]) constitutes a technical basis for adaptation of running services to new system conditions as well as changing business requirements.

The downside of using d-AOP for service adaptation at runtime is mostly caused by the actual available implementations of d-AOP frameworks. Chitchyan and Sommerville [7] give a review of the most popular implementations. Nearly all of them insert so-called hooks into the bytecode for every joinpoint. A hook invokes a central Aspect Manager which manages the pointcut definitions and advices. The Aspect Manger decides if the hook is a pointcut and executes the advice if required (figure 3).

The advantage of this approach is that the developer of a service implementation doesn't have to prepare it for adaptation. The hooks are inserted at load time and the aspects can be added and activated at runtime. But during our experiments, we noticed also some disadvantages:

1. *Performance overhead*: calling a central Aspect Manger each time a hook is reached in the program flow results in a great performance overhead. Hooks are inserted for all joinpoints (constructor, field and method interception). As we have tested in our experiments, adding this additional layer of abstraction constitutes a significant overhead. It is possible to specify where the hooks should be inserted and the overhead decreases if fewer hooks are defined, but this limits the definition of pointcuts at runtime. A joinpoint can only be advised, if a hook was inserted before.
2. *Maintenance:* AOP [13] should help to avoid scattered and tangled code. Crosscutting concerns should be encapsulated. So "adding" or "changing" code with the help of d-AOP to update an application or service is strictly speaking often a misuse of aspect-oriented programming. To maintain an application which was "hot-updated" with d-AOP is a challenge for the developers. Code which originally belongs to the core concern is realized as an aspect. Further evolution of such an application has to consider behaviour which has been woven into the implementation at runtime. A developer has to be aware of all pointcut definitions,

which is hardly manageable.

3. *Limited functionality:* current implementations of d-AOP frameworks are still limited in their functionality in contrast to AOP frameworks which use compile time or load time weaving.

4. *Testing:* Using aspects to add or modify the behaviour of an application makes testing difficult. The implementation may be distributed, parts are still in the original implementation and other parts are implemented in aspects. This conflicts with good object-oriented design, where objects should encapsulate a specific behaviour to ease maintenance and further development.

Because of these disadvantages, we propose an approach that does not "add" new functionality or updates to the existing services, but exchanges the service implementation at runtime.

3.2 Dynamic adaptation: replacing service implementations at runtime

A service consists of a *service description* and a corresponding *service implementation*, as introduced in section 2. Our solution of updating a service at runtime is to switch the service implementation (which is encapsulated in a component), while the *service interface* as part of the service description stays valid. This implies that the new version of the implementation has to fulfil the same service interface. To enable this behaviour at runtime, the following requirements as introduced by [2] have to be accomplished:

- *Transparency:* The replacement of a service implementation has to be transparent for the depending services and applications. No extra code should be included into the implementation of a service to handle the replacement of its dependencies. But it is justifiable that a component implements special methods to consider the replacement of itself.

- *Atomicity:* Changing a service implementation must be uninterrupted and therefore an atomic operation. Other services or application are not allowed to see any intermediate states. Accessing an intermediate state may also result in unpredictable side effects. As described in section 2.2 solely changing the entry in the service registry can result in race conditions. Depending services may hold references to different implementations at the same point in time, which may cause indeterministic behaviour.

- *State preservation:* Attributes of the service may have a specific value at the moment of change. This state of the service has to be transferred to the new implementation. Therefore the state of the old version of the underlying component has to be saved and injected into the new version.

- *Lifecycle management:* The whole adaptation process has to be coordinated by the environment. A Lifecycle management has to control the process: Saving the state of the old version, replacing the old version by the new one and restoring the state. The Lifecycle management must also ensure the atomic execution of the whole process.

Based on these requirements a *dynamic adaptation concept* is designed, following the approach of a service-oriented component model, propagated by [6]. This model incorporates the SOA approach and a component model as defined in [24, 23]. In such a model, a service implementation is realized in a component, whose lifecycle management is part of the framework. The other requirements are met by enhancing this lifecycle management with synchronisation and state transition mechanisms as illustrated in the next section.

4. COBRA FRAMEWORK

The CoBRA Framework provides a Java-based execution environment, implementing a service-oriented component model [6] with additional capability for adaptable services as discussed in section 3. In terms of [22] the CoBRA Framework can be seen as a *dependable mutable middleware.* That means an adaptation to changing needs at runtime is possible and an interference between the adaptation process and the running application is prevented.

The dynamic adaptation of services is entirely managed by the CoBRA Framework. Services are therefore guaranteed even over version transitions. This implies some extensions regarding the events for service manipulation, resulting from the service guarantee:

- *Arrival of a new service.* A new adaptable service is propagated into the framework. With the registration in the service registry, the service is available for all components, without any further functionality compared to the classical service event.

- *Departure of an existing service.* An existing service may only depart if it not used by any service requester. In terms of the components, a component may only be uninstalled if no exported adaptable services are bound to any service requester.

- *Adaptation of an existing service.* This event is introduced for the adaptation of an existing service from one version to another without influencing the availability of the service. A service requester therefore is not affected by the adaptation process. In a classic SOA Framework, this event is represented by the combination of a departure and an arrival event. The issue of ambiguity introduced in section 2.2 and the resulting requirement for atomicity of the adaptation process require additional consistency mechanisms, represented by the adaptation event. The cause is the different semantic of this event in the adaptation context in contrast to a classic remove and add behaviour of a service replacement.

4.1 Adaptable services

The CoBRA Framework architecture is designed to ensure the described behaviour for these events in a fully transparent matter. Before we explain the architecture, a definition

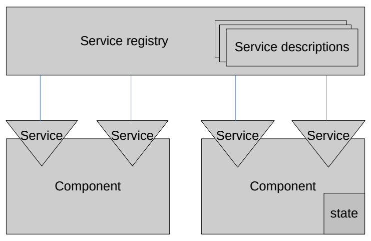

Figure 4: CoBRA services

of adaptable services in the context of the CoBRA Framework has to be given.

An *adaptable service* in the CoBRA Framework is always realized within a component, as shown in figure 4. Therefore components are the entity of consideration when illustrating the architecture. At first, two underlying component types must be distinguished: *stateful components* and *stateless components*.

Stateless components lack the need for a state transition in case of adaptation. They have no component wide state. Therefore no state preserving actions must be taken by the framework. *Stateful components* however have a component-wide state which is required to be transferred in an adaptation process. Which data is considered as the component wide state depends on the transition strategy. The CoBRA project follows a *weak migration* strategy [11] considering global attributes. For the state transition in an adaptation process, a *state transition protocol* is defined. At first this protocol requires unique identifiers to be introduced for each member of the component wide state. These identifiers ensure the identifiability of single state members throughout the adaptation process. Secondly, taking advantage of the identifiers, two procedures, for store and restore are defined. This will be discussed in more detail in section 4.2.2.

Another requirement for adaptable services is the *service contract*. Each service provider and service requester has to agree on a fixed service description, defining the protocol of their communication. In the CoBRA environment a service contract is represented by an interface exported by the underlying component of a service provider. These interfaces are registered in the *service repository* on framework level, as shown in figure 5.

Adaptable services respectively components exist in their component space managed by the CoBRA framework. The architecture of this Framework layer is subject of the next section, starting with the levels of management in the context of service adaptation.

4.2 Architecture

The CoBRA Framework architecture, as shown in figure 5, consists of three hierarchical managing levels. Following a top-down approach, the first level is the framework level. On this level a global *adaptation manager* coordinates components and their exported services. With focus on high availability, at this level all components supporting service adaptation are registered with the *adaptation manager* and any adaptation requests are triggered and controlled via this management service.

The second level is the component management. On this level the state management is located, i.e. considering the case of a service adaptation, the transfer of the current state from the adapted service or respectively the underlying component to the adapting component is managed.

On the third management level services themselves are responsible for management tasks. Each service is therefore wrapped by a protection proxy [9] representing the consistency management. The responsibility of the consistency management is to ensure the atomicity of the evolution process for the affected system.

These three levels of management are interwoven and coordinated by an *event manager* on framework level. As a result in case of an adaptation request the framework level management notifies the affected component and service level management instances, which take the required actions, illustrated below.

4.2.1 Adaptation manager

At framework level a global *service registry* is available. The role of the service registry is to provide the base for a framework-wide nameservice, to discover and bind to registered services. Moreover an additional *component registry* in combination with the *service repository* is introduced, where all components, exporting adaptable services, are implicitly registered at load time. For stateful components also a *state container* is maintained in the registry, to enable the proposed state transition. The service repository depicts a component containing all declarations of adaptable services, in terms of [16] acting as a repository for service contracts. The difference between the service registry and the service repository is founded on the state of the registered services. While the service repository contains all known service descriptions, even services currently not associated with a service object, the service registry registers only current available service descriptions with their corresponding service objects. Therefore the service definitions in the service registry represent a subset of the ones the service repository contains.

All adaptation requests are triggered and managed via the *adaptation manager*, by use of the state and consistency management. In case of an adaptation request, the adaptation manager coordinates the whole adaptation process. First it identifies the affected components. Then the affected state and consistency management entities are triggered via the synchronous event system. The timely fashion of this process is discussed in detail in section 4.3.

4.2.2 State management

State management takes place at component level, by the definition of the *state transition protocol* introduced in section 4.1. This protocol implements the memento pattern [9] and is maintained at framework level and so unaffected by the adaptation process. All attributes of the component-wide state are saved into the container with unique identifiers to enable identification in the restore process.

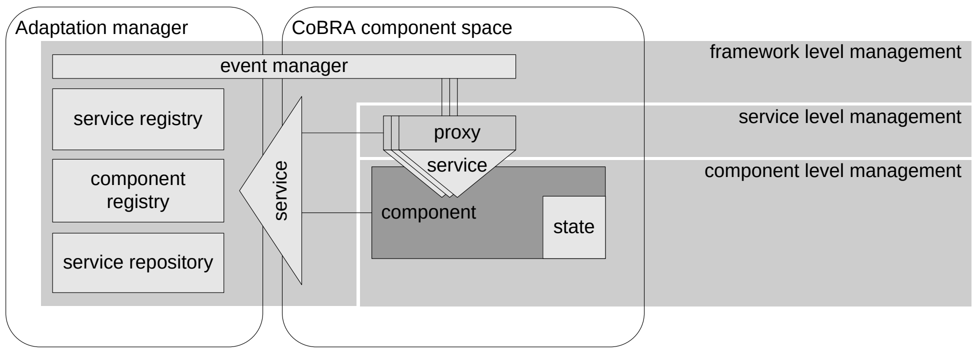

Figure 5: CoBRA architecture

During instantiation of the adapted component, the previously saved state container is injected into the component. After injection, the restore function of the state transition protocol is called by the framework and restores the state of the component. Variations in the semantic meaning of attributes must be considered and corrected at implementation time of the restore function, when accessing and processing the single fields of the container for restoration. Therefore the transition protocol is implemented in two stages: At the first deployment of an adaptable service the store function must be defined. Afterwards each adapting component must incorporate a restore method corresponding to the constraints of the store method of the adapted component. Moreover a store method must be defined for further adaptation processes to complete the transition protocol.

4.2.3 Consistency management

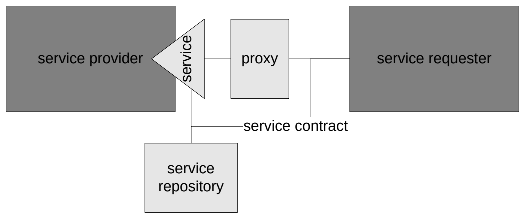

Figure 6: CoBRA service interaction

Every service in the CoBRA environment is replicated at runtime by a proxy following the *protection proxy* pattern defined in [9]. As an illustration of this, figure 6 shows, that every service requester in the CoBRA environment does not access the service providing object directly, but through a proxy, representing a *guardian object*, as introduced in [4]. The difference between these guardian objects and the CoBRA proxy system is the transparent behaviour for the service requester, due to the identical signature and type of the proxy compared to the corresponding service interface.

The need for this indirection by a protection proxy is founded on the requirement for an atomic adaptation point. The used weak migration strategy for state transition requires a consistent point of the local state in the component to adapt. As a result no manipulation on this state is allowed, once the adaptation process is initiated. In the CoBRA environment the possibility of a state change during adaptation and the resulting inconsistencies is eliminated by a blocking behaviour provided by the proxy concept. That means a service execution is stalled by a locking mechanism in the proxy before the method call is delegated to the underlying component as long as the adaptation process lasts. If the reference to the underlying component in the proxy is updated to the adapted component and the state transition of the component is completed, the stalled execution is resumed. As pointed out, this concept of a service level consistency management ensures the atomicity and transparency of the adaptation process in the viewpoint of a service requester.

4.3 Service adaptation process

Another important aspect in a dynamic environment is the adaptation process in a timely matter. In this section the interaction between the management levels throughout the adaptation process is described.

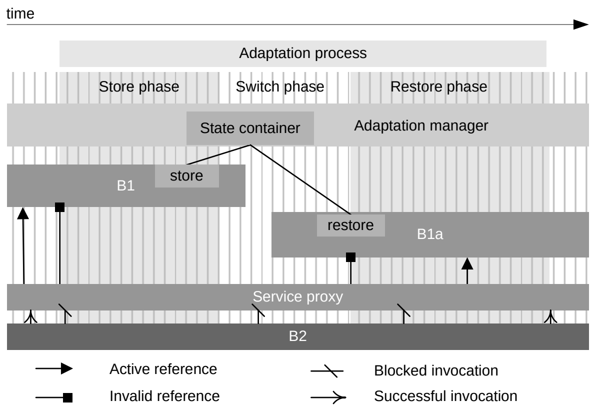

Figure 7: Service adaptation process

Figure 7 shows the adaptation process, which is divided in three phases: The *store phase*, the *switch phase* and the *restore phase*.

The *store phase* is initiated when a request for adaptation is received by the adaptation manager. In the CoBRA framework, adaptation only takes place if no affected service is currently executed. Therefore, at the beginning of the store phase, the adaptation process blocks as long as a service is in use (*service lock*), but no further service calls are allowed (*adaptation lock*). These consistency issues are managed by the proxies introduced in section 4.2.3. When the *service lock* is released, the adaptation process begins. First the references to the service object held in the service proxies are removed. With the adaptation lock set, all service invocations affected by the adaptation block in their proxies as shown in figure 7. The second step in the store phase is the storage of the component state associated with the bundle under replacement as described in section 4.2.2. In short, the store phase ensures a consistent state in the adapted component via the consistency management and the first part of the state transition protocol is applied, by storing the local state in the adaptation manager registry.

The *switch phase* represents the time interval, where the underlying component is replaced. Assuming that a consistent state is reached in the store phase, the adaptation manager first removes the component to adapt and if successful, installs the replacing component. In case of an unsuccessful installation of the replacing component, a rollback mechanism restores the adapted component to ensure continuous service availability. During the installation process of the replacing component, the adaptation manager injects the state container of the adapted component for state restoration purpose. This is the beginning of the *restore phase*.

The *restore phase* is the final part of the adaptation process. At first the state of the replacing component is restored by the restore function in conjunction with the previously injected state container. Then the component propagates all services to the framework. At this point all affected currently blocking service proxies receive a notification about the service arrival. As a response all notified proxies update their reference to the adapted service object and resume all stalled service invocations. The adaptation lock is finally released by the adaptation manager and the adaptation process is finished. From the viewpoint of a service requester the above depicted approach of the CoBRA Framework ensures a fully transparent and atomic adaptation of services in a consistent matter.

5. RELATED WORK

Service-oriented approaches like Jini [1] or OSGi [18] are frameworks referring to SOA concepts introduced in section 2. Jini is a Java based platform and supports multiple registries in a distributed environment and introduces a service leasing concept in which the service access time is restricted. OSGi provides a centralized service registry and a lightweight Java based component model with a dynamic lifecycle management. Therefore OSGi incorporates not only the SOA approach, but a service-oriented component model, introduced in [6].

Service adaptation is discussed among others in [4] or [16], which both follow a replacement strategy. These approaches both use handler objects to decouple user and provider of an interface and to control access. Architectures based on the concept of [15] enable runtime adaptation by controlling the sent messages between components. Other approaches like [5] or its integration in OSGi [8] compose components from subcomponents, which are the target of replacement, to enable the adaptation of the whole component to changing demands. These approaches based on the FRACTAL Component model lack the runtime support for adaptation due to the missing focus on synchronisation and state transition mechanisms.

Another strategy to overcome the problem of adaptation is based on delegation. [21] proposes an approach, where adaptation is a computation of a difference function between the adaptation partners. This difference function is realized as an aspect which is woven into the component to adapt. Kniesel [14] introduces a concept for adaptation on object-level based on a delegation concept. It propagates a concept for object inheritance. In other words, not only classes may be subject to inheritance, but also instances of classes at runtime.

For the CoBRA Framework an integration of both, service orientation and a component model is necessary. The service-oriented component model introduced by [6] provides an approach to integrate the two concepts of service orientation and component based design. The CoBRA approach takes this idea on and currently provides a prototype based on the OSGi Service Platform enhanced with the discussed service adaptation concept.

6. CONCLUSIONS AND FUTURE WORK

This paper presented an approach to replace service implementations at runtime to provide a foundation for autonomic, self-managing, self-healing, self-optimizing, self-configuring and self-adaptive applications.

Because services are highly connected in a SOA environment, taking a service offline, even if a replacement is available, may influence a lot of depended services and applications. We have presented a possibility to achieve runtime adaptation using dynamic AOP and described the advantages and disadvantages of that approach. Due to the fact that the disadvantages outweigh the benefit of using d-AOP, we introduced the replacement of components at runtime to switch the implementation of a service. The challenges in our runtime adaptation approach are the transparent and atomic replacement, while preserving the state of the components. To fully meet the introduced requirement of transparency the extraction of the state transition from component code is the ongoing core research field, as well as the integration of *dynamic adaptable aspects* for the realisation of crosscutting concerns, that help to avoid scattered and tangled code.

In the CoBRA prototype the OSGi Service Platform provides the dynamic lifecycle management and a lightweight service-oriented component model. Based on this platform, the management facilities for consistency and state preservation are incorporated in a transparent matter. The realisation of the prototype based on the OSGi Service Platform with its lightweight design enables the appliance in manifold application areas, ranging from embedded systems to cluster environments, which are also currently under examination. The integration of the CoBRA runtime adaptation into an OSGi based *Fragment component model* is another field under development. A possible model for the integration is proposed by [8].

We are currently developing a database management system which uses the CoBRA prototype to test our approach in a large scale evaluation, as well as studying the influence of persistent data stores on the state migration and developing a model to support transactions in CoBRA.

In conclusion the CoBRA framework can be used to build self-managing systems by adding a layer to manage the adaptation of services with respect to changing requirements.

7. REFERENCES

[1] K. Arnold. The Jini architecture: dynamic services in a flexible network. In *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 157–162, New York, NY, USA, 1999. ACM Press.

[2] J. Balasubramanian, B. Natarajan, D. C. Schmidt, A. S. Gokhale, J. Parsons, and G. Deng. Middleware support for dynamic component updating. In R. Meersman, Z. Tari, M.-S. Hacid, J. Mylopoulos, B. Pernici, Özalp Babaoglu, H.-A. Jacobsen, J. P. Loyall, M. Kifer, and S. Spaccapietra, editors, *OTM Conferences (2)*, volume 3761 of *Lecture Notes in Computer Science*, pages 978–996. Springer, 2005.

[3] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for corba, 1998.

[4] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System.* PhD thesis, Massachusetts Institute of Technology, 1983.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36:1257–1284, 2006.

[6] H. Cervantes and R. S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 614–623, Washington, DC, USA, 2004. IEEE Computer Society.

[7] R. Chitchyan and I. Sommerville. Comparing dynamic AO systems. Technical report, Dynamic Aspects Workshop (held with AOSD 2004). Technical Report No. 04.01, Research Institute for Advanced Computer Science (RIACS),California, USA. Pages 23-36, 2004.

[8] M. Desertot, H. Cervantes, and D. Donsez. FROGi: Fractal components deployment over OSGi. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 275–290. Springer, 2006.

[9] E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.

[10] M. N. Huhns and M. P. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 9(1):75–81, 2005.

[11] T. Illmann, F. Kargl, M. Weber, and T. Krüger. Migration of mobile agents in Java: Problems, classification and solutions. In *Proceedings of the MAMA'00, Wollogong, Australia*, 2000.

[12] F. Irmert, Meyerhöfer, and M. Weiten. Towards Runtime Adaptation in a SOA Environment. In W. Cazzola, S. Chiba, Y. Coady, S. Ducasse, G. Kniesel, M. Oriol, and G. Saake, editors, *Proceedings of ECOOP'2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'07)*, pages 17–26, Berlin, Germany, 2007.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[14] G. Kniesel. Type-safe delegation for run-time component adaptation. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 351–366, London, UK, 1999. Springer-Verlag.

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

[16] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Proc. of 18th International Conference on Architecture of Computing Systems*, 2005.

[17] Organization for the Advancement of Structured Information Standards. Reference model for service oriented architecture 1.0, commitee specification, August 2006.

[18] OSGi Alliance. OSGi Service Platform core specification, release 4, August 2005.

[19] M. P. Papazoglou and W.-J. Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[20] A. Popovici, G. Alonso, and T. Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In *AOSD '03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 100–109, New York, NY, USA, 2003. ACM Press.

[21] S. C. Previtali. Dynamic updates: Another middleware service? In *Proceedings of the 1st Workshop on Middleware-Application Interaction (MAI'07)*, pages 49–54. ACM Digital Library, 20 March 2007.

[22] S. M. Sadjadi and P. K. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Department of Computer Science, Michigan State University, East Lansing, Michigan, December 2003.

[23] I. Sommerville. *Software engineering (5th ed.).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[24] C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[25] E. Thomas. *Service-Oriented Architecture.* Prentice Hall PTR, Upper Saddle River, 2005.

[26] A. Vasseur. Dynamic AOP and Runtime Weaving for

Java - How does AspectWerkz Address It? AOSD 2004 International Conference on Aspect-Oriented Software Development, Invited Industry Talk, March 2004.