Self-Adaptive Multithreaded Applications - A Case for Dynamic Aspect Weaving

Andreas Rasche
Hasso-Plattner-Institute at
University of Potsdam
Prof.-Dr.-Helmert Str. 2-3
14482 Potsdam, Germany
andreas.rasche@hpi.uni-
potsdam.de

Wolfgang Schult
Hasso-Plattner-Institute at
University of Potsdam
Prof.-Dr.-Helmert Str. 2-3
14482 Potsdam, Germany
wolfgang.schult@hpi.uni-
potsdam.de

Andreas Polze
Hasso-Plattner-Institute at
University of Potsdam
Prof.-Dr.-Helmert Str. 2-3
14482 Potsdam, Germany
andreas.polze@hpi.uni-
potsdam.de

ABSTRACT

Shorter product cycles, the requirement for immediate reaction to cyber-attacks and the need for the adaptation to changing environmental conditions demand software reconfigurations to be performed at runtime, in order to reduce downtime. Especially long running applications, which have to provide continuous service should not be restarted for maintenance. They must be updated dynamically.

We have developed a reconfiguration strategy allowing to identify valid reconfiguration points even in multithreaded environments, enabling dynamic application updates. The usage of dynamic aspect weaving enables us to transparently create self-adaptive applications without additional compilation steps or programming constraints in the software development process.

We demonstrate how our approach can be applied to a real-world retail application of a large logistics company. We will describe the implementation of the reconfiguration aspect and our dynamic aspect weaving tool Rapier LOOM.NET.

1. INTRODUCTION

The adaptation to changing environmental conditions, new product requirements, or software errors demand the reconfiguration of software. Often, long running applications can not be restarted for reconfiguration - but has to be performed during application runtime.

Today's software system often use threads to handle many tasks simultaneously. This makes the reconfiguration of such software a challange. Inter-component interactions must be tracked in order to identify valid reconfiguration points. Within this paper we will show how reconfiguration concerns for multithreaded component-based applications are identified and separated using aspect-oriented programming (AOP)[2] techniques. This allows for the update of components without knowledge of the components functionality and its interactions with other components.
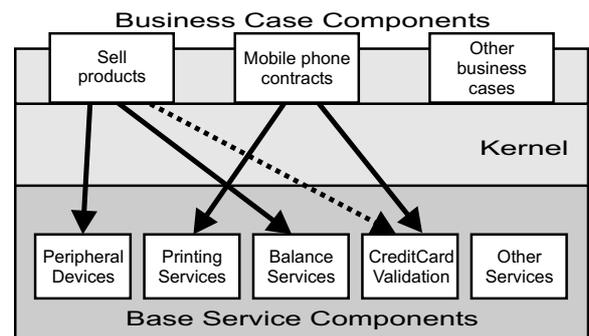

Figure 1: Architectural sketch of the updatable retail application on a frontend computer

Within this paper, we discuss how our solution can be used to improve an existing retail application. This application is installed on many thousand computers distributed over many different locations. One of the requirement for the application is that business logic can be exchanged dynamically. I.e. if the marketing department plans special offers for certain products, the according business case component must be deployed as soon as possible. As a second example, imagine that the operations control has detected a critical security issue within the credit card component. This should be fixed immediately.

Figure 1 shows a skeleton of the application described. Every business case (i.e. selling a certain product or making a cell phone contract etc.) is implemented inside a business case component (BCC). These components are shown in the upper part of the picture. At the moment there are approximately 350 business case components available. Depending on their requirements a business case component can use several base services. These can be seen in the bottom of figure 1. A base service typically implements common tasks. I.e. it provides functionality to print invoices, to gain access to a product scanner or authorizes credit card payments. Base services can use other base services. A typical configuration includes about 50 base services.

The kernel, displayed in the middle of figure 1, is the coordinator of the application. It instantiates appropriate components, establishes connections among business case components and base service components. In addition the kernel is responsible for displaying a graphical user interface, and a variety of other tasks. The application, including business

case components, base service components and the kernel is executed within is single operating system process on a PC at a retailstore. The application has been implemented using the Microsoft .NET platform.

The backend system (not displayed in figure 1) takes all data produced by business case components and routes them to backend services such as the companies ERP (Enterprise Resource Planning) system or to third party services. The connection to the backend system is managed by different base services. Each backend service is represented by a base service component in the frontend.

A typical use-case of the retail application is that a customer wants to buy a product: The clerk identifies the business case "sell product" by entering a numerical code whereon the kernel starts the appropriate BCC. The BCC displays a form within the user interface and asks the kernel for base service components including a balance service, a product scanner service, a credit card payment service, and an invoice service. Afterwards the BCC invokes the product scanner component to determine which products have been selected by the customer. In a subsequent step the clerk starts the payment (by pressing the button in the form) whereon the BCC reports the sales to the balance service component. In case of a credit card payment, the credit card component is involved as well.

Now lets take a closer look at the reconfiguration of this application. Shorter product cycles, new offerings and the requirement to react almost immediately on cyber-attacks makes an architecture that supports dynamic updates highly desirable.

The algorithm described herein supersedes a pull-based update scheme implemented in an earlier version of the retail application. With the pull model, the backend component used to ship new and/or updated components to each computer in the frontend. In addition a new configuration file for the application is transmitted. After restarting the application, the new configuration is activated and the kernel uses the new components henceforth. The problem is, that usually the application will be restarted only once a day. Even worse a store operating around the clock restarts the application days later only. This means that after the deployment of an updated component it could take some days until an updated component goes in production.

Within this paper we present an approach how dynamic updates of the described application can be performed almost immediately and discuss challanges on the way.

The remainder of the paper is structured as follows: In the next section we introduce our approach for dynamic reconfiguration of multithreaded component-based applications. Afterwards we present details of our dynamic aspect weaver library and explain implementation details of the separated configuration concern. Finally after the presentation of related and future work we conclude the paper.

2. TOWARDS A SOLUTION

First we introduce our model of multithreaded component-based applications: An application is a collection of components interacting via *public interfaces*. Within this paper we consider components to be software activities that are dynamically instantiated at runtime (i.e. objects). Inside a component, objects of different classes may exist. Com-

ponents are instantiated by the instantiation of one class, which is called *main-class*. The main-class may export one or more interfaces, which are considered as the components public interfaces. A reference to a component is always a reference to one of its public interfaces. A public interface method may be called from multiple threads, which execute concurrently.

The *reconfiguration* of component-based applications includes the addition, removal and update of components. While public interface signatures must not change, an *update* of a component includes:

- the addition, removal and change of member variables

- the addition and removal of methods (if no public interface method)

- change of method signatures (if no public interface method)

- code changes (addition, removal, change of instructions)

In order to preserve consistent component state, reconfiguration activities are allowed at certain points in time only: We are not able to restore execution state within a method after reconfiguration. That is why all methods of a component must complete ongoing method execution. For example, if a credit card service method starts a transaction and waits for a transaction commit, a reconfiguration within these steps leads to an incomplete transaction.

As a second example why reconfiguration can not be execute at any point in time consider our retail application using a balance service, booking all incomes. In case of an update two version of the balance service could exist, because some BCCs are still booking their incomes with the old service, while other BCCs are already using the new component. In this case consistency of the income record might be damaged.

We have developed a new approach to reconfiguration of multithreaded component-based application, matching the described requirements, which will be presented now.

2.1 The Dynamic Proxy Approach

Our approach is to dynamically create proxies for all public interfaces of a component at its load-time. Within this proxy we forward all method calls to the currently active implementation, represented by a reference within the proxy. Proxies are generated by our dynamic aspect weaver Rapier LOOM.NET, which is described later. Our dynamic proxy approach adds only small overhead on calls to public interface methods.

Figure 2 illustrates our approach. Within the picture you can see a service component referenced by three clients. Each of the client components is executed within a separate thread. Furthermore, each client component is interwoven with our reconfiguration aspect. Our aspect implementation adds code to each public interface implementation, which allows for controlling method execution. After a reconfiguration the aspect code acts like a proxy forwarding method calls as described. In addition the aspect adds synchronization code needed to catch valid reconfiguration points, as
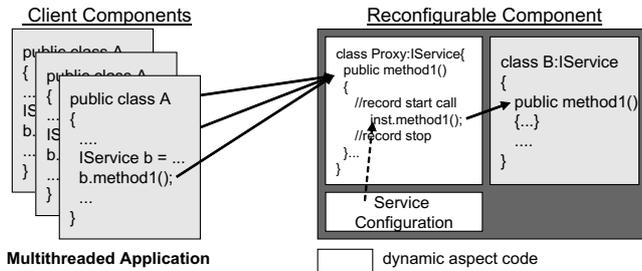
Figure 2: Dynamic Proxies

explained within the next section. In figure 2 the proxy represents the public interface `IService`, while the interface implementation is given by a main-class B, referenced through the variable `target`. A reconfiguration of the component is realized by the reassignment of the `target` variable.

Our aspect code ensures that all inter-component references are tracked transparently, without any knowledge of the distribution of references to this component. Furthermore, during load-time weaving an additional configuration interface is added to each component. This interface includes methods to trigger reconfiguration by an external configuration manager, which is not focus of this paper.

The technique of dynamic weaving offers a novel approach for the development of reconfigurable applications. In contrast to existing implementations using bytecode rewriting [4], our approach does not require additional compilation steps. The seamless integration into the software development process is implemented by the transparent activation of the reconfiguration aspect during component instantiation through our framework using the factory pattern (In the described retail application, the kernel instantiates new components and activates dynamic weaving).

2.2 Our Reconfiguration Algorithm

Existing reconfiguration approaches are based on actor-like models, which assume single threaded method invocation. Our approach supports the reconfiguration of multi-threaded applications: In order to identify valid reconfiguration points we record all method calls to methods of public interfaces. In case of a reconfiguration request for a component we have to prohibit new threads from entering the component through a public interface method call. In addition all method calls in progress must be completed. This means that there must not be any method call to that component on any thread's call stack.

The described reconfiguration algorithm is implemented within the aspect code by using the *readers and writers lock* (`rw-lock`) algorithm[1, 80 pp.]. `Rw-locks` can be used to synchronize access to a resources, allowing concurrent read access but single write access. Within the reconfiguration aspect we use one `rw-lock` per component to protect the `target` variable, residing in the interwoven aspect code. All method calls on public interfaces acquire a read-lock at the component's `rw-lock`, while reconfiguration requests have to acquire a write-lock. This write-lock will be granted when all read-locks owned by other threads in progress, have been released. Method calls of threads in progress can still acquire read-locks, although there has been a write-lock request (from a configuration interface method) due to the recursive read-lock feature of the used `rw-lock` implementa-

tion. A recursive lock is acquired if a thread already owning a read lock acquires the read lock again. A recursive read-lock is released when a lock counter reaches zero. Within the reconfiguration aspect code the reassignment of the `target` variable is executed exactly when all read-locks of all pending thread's method calls have been released. The usage of `rw-locks` adds almost no overhead to normal application execution, because the acquisition of a read-lock requires no synchronization if there is no pending write-lock request.

2.3 State Transfer

In order to preserve state of objects during reconfiguration, we have to transfer state of all updated components from the old to the new versoin. We implement this either by *implicit* or *explicit state transfer*. On the one hand implicit state transfer means that the content of the main-class object's is copied member-wise to the new version. On the other hand we support the definition of a state transfer method, which can be used by programmers to define application specific update logic for a component update (explicit state transfer).

Implicit state transfers are applicable for *hot-fixes*. We define hot-fixes to introduce only minimal code changes and require no changes to the set of member variables. In this case, a state transfer can be implemented transparently without application-specific knowledge. Hot-fixes can be used in order to fix severe program errors (bugs) after major updates, with very short reaction times, because no additional update logic must be implemented for dynamic updates.

In order to describe the integration of state transfer into the reconfiguration process we will explain the steps necessary for reconfiguration: We will illustrate an update of Component A_{V1} running in version V1 to a new version A_{V2}:

1. Ensure that there is no pending method call to A_{V1} and prevent other threads from calling methods of A_{V1}

2. Create a new instance of A_{V2}.

3. Transfer the state of A_{V1} to A_{V2}.

   (a) Member-wise clone A_{V1} to A_{V2} - or -

   (b) Call state transfer method of A_{V2} with A_{V1} as parameter

4. Update target reference in the proxy to A_{V2}.

5. Enable method calls.

2.4 Application Reconfiguration

The reconfiguration of a whole application can involve the update of a set of components. Commonly expected the reconfiguration has to be performed in transitive fashion, however our approach traces object references at access and triggers reconfiguration operation on all objects involved explicitly. This allows for the independent reconfiguration of each single component.

Within our retail application, the kernel keeps a list of all instantiated components. In case of a reconfiguration the kernel simply sends a reconfiguration request to each component's configuration interface. The reconfiguration process is completed when the reconfiguration of all components was successful.

2.5 Limiting Reconfiguration Time

Reconfiguration time is heavily influenced by the time it takes to complete ongoing method calls. Especially in high load situations many client requests will be processes simultaneously. In general using our approach it takes a maximum amount of time:

$$t_{worstreconf} = \sum_{\#clients} t_m$$

to reconfigure a component. This worst case reconfiguration time $t_{worstreconf}$ is reached when all clients just started to call the method with the longest execution time t_m available. In order to cope with high load situations, we have added a timeout parameter into the reconfiguration method of our configuration interface. If a reconfiguration cannot be finished within the specified time we simply stop blocking all threads with pending calls and do not update the component. Because the reconfiguration depends on the actual context of calling clients it is possible that a reinitiation of the reconfiguration right after a first try could be executed in the time specified, because the clients have advanced in method execution.

Another possible algorithm would be to defer the reconfiguration until load has decreased or another algorithm could decide to increase the permitted reconfiguration timeout. Depending on the application context these algorithms have their advantages or disadvantages. We are going to investigate timely behavior of reconfiguration in more detail within future work.

3. THE RECONFIGURATION ASPECT

This section gives a short overview about Rapier LOOM.NET, the aspect weaver we used for our experiments. The aspect weaver arises from the LOOM.NET project [7, 8]. Rapier LOOM.NET is a dynamic aspect weaver and provides its functionality through a library. In the second part of the section we present implementation details of the aspect code.

3.1 Aspects in Rapier LOOM.NET

In Rapier LOOM.NET, an aspect can easily be defined through an *aspect class*. An aspect class is a special .NET class with methods constructors and fields as well. At defined *join points*, an aspect class becomes interwoven with a target class. Interweaving, strictly speaking, means that either code will be added at a specific point to a target method or an aspect introduces additional methods to a target class by adding a new interface and its implementation to the target class. In both cases, the additional code is implemented by *aspect methods*.

An aspect method is defined within an aspect class. An advice attribute[1] is used to mark a method as aspect method. Advice attributes can be `Call` or `Create` to identify aspect methods which should add code to target methods, and `Introduces` to declare an interface implementation to be added to a target class. Not necessarily every method in an aspect class is an aspect method.

In addition to advice attributes, Rapier LOOM.NET defines several join point attributes. These attributes are used to

<hr>

[1]Attributes are metadata information in .NET which can be used to annotate programming elements such as types and methods
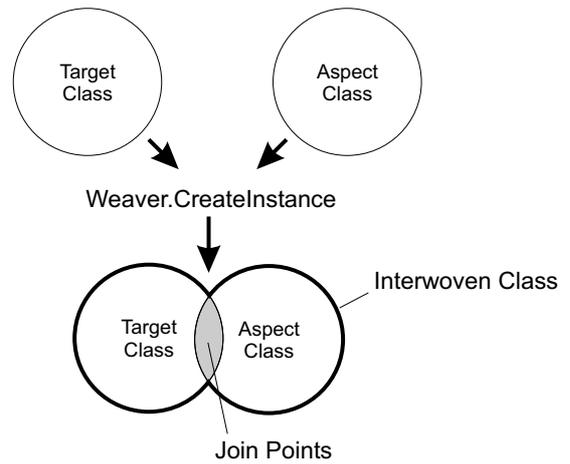

Figure 3: The weaving process

describe which methods should become interwoven with the aspect method. Examples for these attributes are `Include`, `Exclude` and `IncludeAll`.

A target class is a normal .NET class. The one and only limitation is that target class methods (which should become interwoven) either have to be virtual or to be defined via an interface. Regarding to our application model this no restriction, because all communication is based on public interface method calls. The weaving process will be initiated during runtime within a factory. Instead of using the *new* operator, the weaver's factory methods are used to produce interwoven objects. Figure 3 depicts this in detail.

3.2 Rapier LOOM.NET vs. AspectJ

AspectJ is pioneering work and has implemented many AOP ideas for Java. Our implementation Rapier LOOM.NET picks up on AspectJ and implements similar features for Mono and .NET (table 1). An advantage of Rapier LOOM.NET is that it works for all .NET languages without requiring a special compiler.

3.3 Implementing the Reconfiguration Aspect

In figure 4 you can find the implementation of the reconfiguration aspect. The aspect works as follows: At first it introduces a new interface (`IConfigure`) (line 7) to every class which becomes interwoven with the aspect. Secondly it controls the execution of every method and possibly redirects the method call to a different target object. This happens within the `Proxy` method, interwoven with all public interface methods, via a `Context.Invoke` or `Context.InvokeOn` call. In Rapier LOOM.NET the `Context` provides access to the interwoven target class. The first case (line 20) addresses the situation that the initial object is still active (no reconfiguration has been performed yet) and the method call will be proceeded to its original implementation. The second case (line 24) represents the proxy if the object has already been reconfigured.

Furthermore, the `Proxy` method takes care that there is no method call in progress during the reconfiguration phase. This is done by using a `System.Threading.ReaderWriterLock`. At any given time, the `ReaderWriterLock` allows either concurrent execution of methods of an interwoven instance (read lock), or it allows one single thread to reconfigure the object (write lock).

	AspectJ	Rapier LOOM.NET
Interweaving	at compile time	at run time
aspect definition	`aspect MyAspect` `{ ... }`	`public class MyAspect:Aspect` `{ ... }`

Definition of interweaving points	With pointcuts:
	`pointcut TraceMethod(): execution(* *.`
	`    CalculatorClass.*(..));`
		With attributes on the aspect method and the method's signature:
		`[IncludeAll]`
		`[Call(Invoke.After)]`
		`object MyCode(object[] )`

Interweaving Aspects	Implicit through pointcuts:
	`pointcut TraceMethod(): execution(* *.`
	`    CalculatorClass.*(..));`
		Explicit with class attributes or at instantiation:
		`[MyAspect]`
		`public class CalculatorClass`
		`{ ... }`
		`or:`
		`Weaver.CreateInstance(typeof(CalculatorClass), null,`
		`    new MyAspect());`

Definition of aspect code	In advices
	`after():TraceMethod()`
	`{`
	`    ...`
	`}`
		In aspect methods
		`...`
		`[Call(Invoke.After)]`
		`object MyCode(object[])`
		`{ ... }`

Table 1: AspectJ vs. Rapier Loom .NET

```
1  public interface IConfigure
2  {
3    void ReplaceReference(object target);
4    bool ReplaceReference(object target, int milisecondsTimeout);
5  }
6
7  [Introduces(typeof(IConfigure))]
8  public class ReconfigurationAspect:Aspect,IConfigure
9  {
10   private object target;
11   private ReaderWriterLock rwlock=new ReaderWriterLock();
12
13   [Call(Invoke.Instead)]
14   [IncludeAll]
15   public object Proxy(object[] args)
16   {
17     rwlock.AcquireReaderLock(-1);
18     try
19     {
20       if(target==null)
21       {
22         return Context.Invoke(args);
23       }
24       else
25       {
26         return Context.InvokeOn(target,args);
27       }
28     }
29     finally
30     {
31       rwlock.ReleaseLock();
32     }
33   }
34
35   // IConfiure Implementation
36   public void ReplaceReference(object target)
37   {
38     ReplaceReference(target, -1);
39   }
40
41   public bool ReplaceReference(object target, int milisecondsTimeout)
42   {
43     if(rwlock.IsReaderLockHeld) throw new ApplicationException("
          ReplaceReference calls must not be in the control flow of the
          object itself.");
44     try
45     {
46       rwlock.AcquireWriterLock(milisecondsTimeout);
47       this.target=target;
48       rwlock.ReleaseWriterLock();
49       return true;
50     }
51     catch(Exception)
52     {
53       return false;
54     }
55   }
56   // IConfigure Implementation End
57 }
```

Figure 4: The Reconfiguration Aspect

The `ReplaceReference` methods are responsible for the reconfiguration of an object. At the beginning (line 43), we have to ensure that the actual thread is not holding a reader lock. Holding a reader lock indicates that the `ReplaceReference` method has been called within the control flow of one public interface method of the component, currently under reconfiguration. But this would outrage our reconfiguration rules, because it is impossible to assure the completion of pending public interface interface method calls. If this happens, the method throws an exception. Otherwise, we try to acquire a writer lock. If we gain this lock then we set up the new target for all method calls. If there was a timeout specified and we could not acquire the lock within the given period then we return without the reassignment of the target variable.

3.4 Applying the Aspect

In the section above we have shown, how the reconfiguration concern can be described as an aspect. Now we want to demonstrate how it could be applied to our retail application. The solution is straightforward: Since the kernel is responsible for creating new components, we simply have to modify instantiation code. Generally speaking we use the aspect weaver to create new components. This means that if a new component is requested by the kernel, an interwoven component with reconfiguration capabilities is created. This is transparent to the component's user. The generated component implements the same interface as the original, still preserving black-box semantic.

Using our approach updates of the described retail application are performed as follows. If a new configuration has been deployed, our configuration manager intersects the new configuration description with the currently running one, resulting in a list of components to be updated. The configuration manager sends a reconfiguration to each of these components via the introduced configuration interface `IConfigure`. In contrast to the prior update mechanism, new configurations are activated almost instantaneously.

Another use-case for our approach is the dynamic activation of additional aspect. For instance we can use this to activate logging and tracing facilities in the retail application

if it is required. The dynamic activation aspect, a powerful technique supported by our approach will be discussed in a future publication in more detail.

4. RELATED WORK

A number of papers on dynamic reconfiguration of software has been published within the last years. However, in contrast to most related work, our approach supports the reconfiguration of multithreaded applications. In addition it offers the seamless integration into the software development process using dynamic aspect weaving.

M. Wermelinger presents a theoretical approach [10] to reconfiguration extending an algorithm of J. Maggee and J. Kramer [3]. In his approach a reconfigurable state is reached by blocking connections between components. The introduction of a transaction concept, combining method calls, ensures consistent state despite reconfiguration. In contrast to our approach, no concept of threads is supported, applications are modeled actor-like including only one thread of control.

JAsCo [9] is an aspect-oriented implementation language developed for the Java Beans component model. Connectors introduce hooks into the execution flow of methods, which allow for the deployment of aspects code. A trap in a method's execution flow triggers a lookup in the central connector registry, where reconfiguration operations are performed. In contrast to our solution, this approach introduces new programming features for the definition of connectors and hooks, which prohibits a seamless integration into the software development process.

M.J. Rutherford et. al. introduce an approach [6] for the update of Java Enterprise Beans by rewriting inter-component reference names within the JNDI naming services, which enables the exchange of component connections and updates during runtime. This approach uses an application server model, which is applicable to our retail application. In our opinion, the usage of a name service has a significant performance overhead.

An interesting approach is the usage of Java bytecode rewriting e.g. [4] in order to prepare an application during compile time for runtime updates. Orso et. al. [4] inject proxy classes which counts active method calls. A reconfiguration request checks the counter and in case of a non-zero counter value, reconfiguration fails. In contrast to our solution there is no possibility to force the occurence of a reconfiguration point. In addition this approach adds an additional compilation step into the software development process.

5. CONCLUSIONS AND FUTURE WORK

Within this paper, we have presented a novel approach to reconfigure multithreaded component-based applications. Starting from an existing retail application as a use-case scenario, we have developed an update algorithm enabling an almost instantaneous component update at runtime. This technique also offers the adaptation of running applications to changing environmental conditions without causing restart delays. In contrast to existing solutions, our approach supports multiple threads accessing a component.

The usage of aspect-oriented programming techniques facilitates a seamless integration into the software development process of reconfigurable applications. Our dynamic aspect weaver Rapier LOOM.NET provides a transparent integra-

tion of our reconfiguration aspect into existing applications. There is no need for additional compile steps and component programmer do not have to care for reconfiguration details.

First measurements of execution overhead of a first implementation have demonstrated that the overhead is minimal and short reconfiguration times can be achieved. A detailed evaluation is planned in the future.

Within future work we will investigate timing behavior of our reconfiguration algorithm and its usage within real-time environments. State transfer will be investigated in more detail for possible optimizations.

We are investigating the development of adaptive application within our *Adaptive.Net* framework [5], which includes tools for building application configurations graphically, monitoring support including a set of standard monitors, and a reconfiguration and deployment infrastructure for complex distributed component-based applications.

6. REFERENCES

[1] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.

[2] Gregor Kiczales and John Lamping and Anurag Menhdhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John 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.

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

[4] A. Orso, A. Rao, and M. Harrold. A technique for dynamic updating of java software, 2002.

[5] A. Rasche and A. Polze. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 164–171, Hakodate, Japan, May 2003.

[6] M. J. Rutherford, K. Anderson, A. Carzaniga, D. M. Heimbigner, and A. L. Wolf. Reconfiguration in the Enterprise Java Beans Component Model. In J. Bishop, editor, *Proc. of the 1st IFIP/ACM Working Conference on Component Deployment, Berlin, Germany*, LNCS. Springer, 2002. To appear.

[7] W. Schult and A. Polze. Aspect-Oriented Programming with C# and .Net. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 241–248, Crystal City, VA, USA, April 29 - May 1 2002.

[8] W. Schult and A. Polze. Speed vs. memory usage - an approach to deal with contrary aspects. In *The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) at the International Conference on Aspect-Oriented Software Development*, Boston, Massachusetts, 17.-21. Mar. 2003.

[9] D. Suve, W. Vanderperren, and V. Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM Press.

[10] M. Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 243–254, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1997. IEEE.