

Dynamic Updates of Graphical Components in the .NET Framework

Andreas Rasche and Wolfgang Schult

Hasso-Plattner-Institute at University of Potsdam
Prof.-Dr.-Helmert Str. 2-3
14482 Potsdam, Germany

Abstract. Modern software has to operate in highly dynamic environments. In order to provide acceptable application-level quality-of-service parameters, applications must be adapted to underlying resource availabilities and other properties of their environment. Dynamic reconfiguration provides a powerful mechanism to realize these adaptation. Besides changes in the application structure and used algorithms, adaptation strategies must also face changes in the graphical representation of user interfaces. In this paper we introduce our Adapt.Net framework for building adaptive applications, focusing on dynamic updates of user interface components in the .NET framework. We use aspect-oriented programming to integrate synchronization logic - required for dynamic reconfiguration - transparently for the developer into application components. State transfer between old and new component version is implemented by cloning objects of the component's object graph member-wise.

1 Introduction

Modern software has to deal with varying resource availability during runtime. In order to provide acceptable quality of application-level quality-of-service parameters, applications must be adapted to changing environmental conditions. Dynamic reconfiguration provides a powerful mechanism to realize the adaptation even during runtime of an application. Within our framework *Adapt.Net*, we are able to change the behavior of component-based applications by:

- adjusting component parameters (e.g. compression rate)
- adding/removing components
- changing connections among components
- migrating components to other execution hosts
- dynamically updating component implementations

Besides tools for building distributed component-based applications, our adaptation framework *Adapt.Net* includes a monitoring infrastructure and a runtime environment able to execute dynamic reconfiguration commands. Adaptation to changes in the application's environment is realized by monitoring and loading new application configurations if requested by a pre-defined adaptation policy.

An *application configuration* denotes a set of parameterized components and the connections among them as well as a mapping to execution hosts in case of distributed applications.

Application configurations are described declaratively. An application developer easily specifies a number of configurations that perform best in a given category of environmental properties and our framework is able to detect what configuration has to be loaded at what time. The usage of categories limits the number of required configurations, by discretizing continuous ranges of environmental properties (including available resources). This allows for example to differentiate network connections over Bluetooth, LAN or GSM and to apply different configurations. For typical mobile applications only a small number of configurations is required. More fine-grained adaptations are possible on the level of component parameters, by adding additional closed-loop control schemes, to the general approach of exchanging whole configurations.

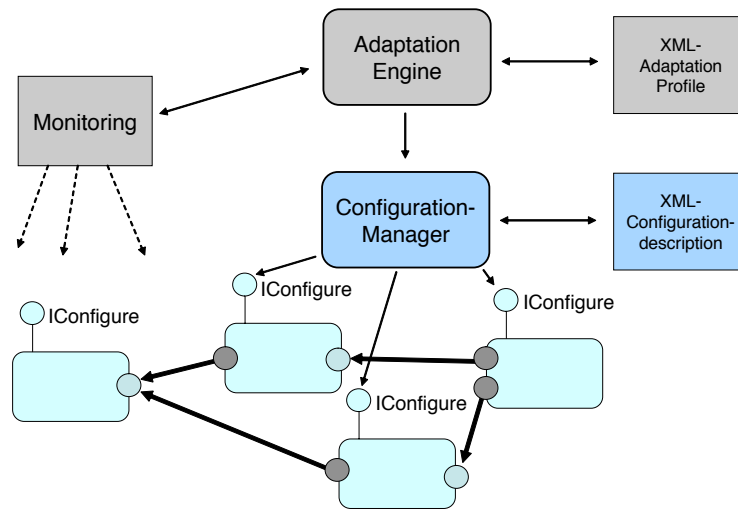


Fig. 1. The Adapt.Net framework runtime infrastructure

Fig. 1 shows the architecture of the *Adapt.Net* framework. The adaptation engine evaluates environmental conditions using a monitoring infrastructure, which relies on observer component's, which are able to measure certain environmental properties (network bandwidth, available battery). Observers are platform-specific, but can be reused among different projects. Based on an XML-based adaptation profile, a dynamic reconfiguration can be triggered via the configuration manager, which is able to exchange an application configuration using XML-based configuration descriptions. Adaptation profiles map a range of environmental properties to application configurations.

Each component of an adaptive application implements a configuration interface which is used by the configuration manager to execute configuration commands. In Fig. 1 four components are connected by four connections. Components communicate by invoking methods at other components (RPCs).

The used dynamic reconfiguration algorithm is based on the blocking of connections between components. The algorithm has been described in detail in [2]. In short, during reconfiguration, the application will be transferred into a reconfigurable state by blocking all involved connections according to a calculated order. Basically, the algorithm ensures that there is no on-going method execution on components involved in reconfiguration. Within a short blackout-phase reconfiguration actions can be performed. We use aspect-oriented programming (AOP) [5] to interweave configuration-specific logic with the functional components of the application, including synchronization between application logic and reconfiguration commands, which is required to guarantee consistency despite dynamic reconfiguration. We have lately integrated the dynamic aspect weaver Rapiers-Loom.Net [12] into our framework.

Dynamic updates of component implementations are an important reconfiguration option. Algorithms can be adapted on a more fine-grained level - on the level of components itself. The update of component implementations typically includes the transfer of state between an old and a new version. We have implemented an algorithm that iterates through a component's object graph and updates all objects, whose types have been specified in a update mapping.

Besides adaptation of the application structure, adaptation must also be reflected in the graphical user interface. For example a reduced functionality of a mobile information system, caused by a reduced bandwidth, should be displayed differently than the full-featured service. Within this paper we will describe steps necessary to update graphical components in the .NET framework.

The remainder of the paper is structured as follows: In the next section we will present our approach for implementing configuration-specific concerns separated from the application logic using AOP. Followed by that we will introduce our algorithm for dynamic updates of component implementations and afterwards, focus on dynamic updates of graphical components. Finally related work and conclusions are presented in section 4 and 5.

2 Configurable Components

In *Adapt.Net*, the term **component** is not only defined as binary deployment time unit, but also forms a virtual boundary during runtime. During runtime we consider a component as a graph of interconnected objects with a main object as its root. Fields of objects can contain references to other objects. Components are instantiated by creating an instance of a main class.

Components interact via public interfaces. In *Adapt.Net*, connections are represented by one element of a component's main object. The configuration manager accesses components via their configuration interface *IConfigure*. We use AOP to introduce this configuration-specific interface to functional components,

transparent for the application developer. The interface contains methods, for setting component parameters, establishing and blocking connections, and to activate component processing after reconfiguration. The interface *IConfigure* and its implementation is added to a component's main class. Our generic implementation of the configuration aspect interacts via reflection with the functional code. Connections are established by injecting new values into fields representing connections. Parameters of components are adjusted with the same algorithm.

Here our solution differs from other approaches such as TRAP/J [11] or Adaptive Java [4]. These approaches use reflection to invoke alternative methods based on adaptation profiles. Reflective binding of methods is performed during runtime, using a framework *Invoke* method, causing high overhead for method invocations in the functional logic. In our approach functional logic and configuration-specific concerns are totally separated from each other. While configuration is handled on a meta-level, there is no code for selecting alternatives for adaptive behavior integrated into the method calls of the functional code. There are only references exchanged during dynamic reconfiguration. During runtime method invocations are performed by inexpensive virtual method calls. A small overhead - in the magnitude of 1-2 instructions - is only caused by synchronization mechanisms.

2.1 Synchronization

The synchronization between functional method calls and the configuration manager is implemented using reader-writer-locks [8], which are typically used for synchronizing requests to shared data structures. In our case, we acquire a read-lock when a method call is initiated. If a connection has to be blocked a write-lock for all method calls, on-going over that connection, will be acquired, causing the required behavior for blocking a connection. The advantage of using reader-writer-locks is a low overhead for functional application code. If there is no pending reconfiguration request only a single atomic instruction is added to inter-component method calls, compared to a method call this overhead is negligible.

Fig. 2 shows the implementation of our *ReconfigurationAspect* for the dynamic aspect weaver Rapier-Loom.Net - used for generating a dynamic proxy for the functional components. In Rapier-Loom.Net, joint-points are defined using .NET attributes¹. The presented aspect code introduces a new interface *IUpdate* and its implementation. The attribute *[Call(Invoke.Instead)]* above the method *InvokeAll* indicates that this code will replace all methods (*[IncludeAll]*) in the target class. The aspect weaver creates an inherited class containing aspect logic that will be used instead of the target class.

Finally an update of a component is triggered via the method *UpdateReference*, which replaces the *target* member with a reference to an updated component implementation - downgrading the original component to a proxy. The

¹ Attributes are meta-data annotations for language constructs, such as classes, methods, assemblies or method parameters.

```

[Introduces(typeof(IUpdate))]
[AttributeUsage(AttributeTargets.Class)]
public class ReconfigurationAspect : Aspect, IUpdate
{
    private object target;
    public ReaderWriterLock rwlock=new ReaderWriterLock();

    [Call(Invoke.Instead)]
    [IncludeAll]
    public object InvokeAll(object[] args)
    {
        rwlock.AcquireReaderLock(-1);
        try {
            if (target == null)
                return Context.Invoke(args);
            else
                return Context.InvokeOn(target, args);
        }
        finally {
            rwlock.ReleaseReaderLock();
        }
    }

    public bool UpdateReference(Hashtable assemblyMapping, int millisecondsTimeout)
    {
        try {
            rwlock.AcquireWriterLock(millisecondsTimeout);
            try {
                if(target==null)
                    target = ComponentUpdater.UpdateObjectGraph(Context.Instance, assemblyMapping);
                else
                    target = ComponentUpdater.UpdateObjectGraph(target, assemblyMapping);
                return true;
            }
            finally {
                rwlock.ReleaseWriterLock();
            }
        }
    }
}

```

Fig. 2. Synchronization: The Reconfiguration Aspect

original component only accepts new requests and forwards them via the *target* member to latest component implementation. The update process will be described in the next section. The reconfiguration request is synchronized with the functional logic by the acquisition of a write-lock as described. Note that the forwarding method call of the proxy does not use reflection. The aspect weaver generates only one additional virtual method call - which causes again low overhead to normal method calls.

One important advantage of using reader-writer-locks for synchronization is, that we can also support nested and recursive method calls. Another advantage of our dynamic proxy approach is that we must not know any client of the components, because only references to the proxy exist in an application. Clients are allowed to create arbitrary copies of the proxy and pass them to other code as well.

In *Adapt.Net* we use this approach for active components. These components have their own control flow (threads). There could exist references on the stack of these threads, that can not be manipulated from the meta-level. That's why we have to use the dynamic proxy approach for these methods.

3 Dynamic Updates

We are now going to focus on updates of component implementations only. Other reconfiguration actions have been described in more detail in other publications. When a component is in a reconfigurable state (there are no on-going methods) the update of the component's object graph can be performed. Besides activating a new component version, state must be transferred.

3.1 Graph-traversal Algorithm

An update is specified as an assembly mapping. Assemblies are units of deployment in .NET that contain the intermediate language instructions (Byte code) for classes and their meta-data. An assembly mapping defines assemblies that have to be updated and assigns new versions to be activated. During the update phase we scan the component's object graph. Each object will be tested for an update against the assembly mapping.

```
object UpdateObjectGraph(object node Hashtable updateMapping)
{
    Type newType = GetUpdate(type.GetType(), updateMapping);
    if(newType!=null)
        object updatedNode = FormatterServices.GetUninitializedObject(newType);

    foreach(FieldInfo _field in type.GetFields())
    {
        if(LeafNode(_field.GetValue(node), _field.GetValue(node).GetType()) {
            if(updatedNode!=null)
                _updatedField.SetValue(updatedNode, _field.GetValue(node));
        }
        else
            if(updatedNode!=null) {
                _updatedField.SetValue(updatedNode,
                    UpdateObjectGraph(_field.GetValue(node), _field.GetValue(node).GetType()));
            }
        else {
            _field.SetValue(node,
                UpdateObjectGraph(_field.GetValue(node), _field.GetValue(node).GetType()));
        }
    }
    return (updatedNode==null):node?updatedNode;
}
```

Fig. 3. Field-wise Traversal

Figure 3 shows excerpted code of the traversal algorithm. Starting at the component's main object we investigate every object's fields and recursively traverse object references. The method *UpdateObjectGraph* implements the traversal algorithm. At first, in line 3, each node (object) is checked for an update according to the given assembly mapping. In case of an update a new version of the object is created. New versions of objects are NOT created using the *new* keyword of the framework, because this would involve the execution of a constructor, which potentially could cause side effects. The method *FormatterServices.GetUninitializedObject*, available in the .NET libraries, creates a new object without running a constructor. It initializes the object's meta-data, method tables and allocates (zero-initialized) memory for the object's fields.

Afterwards, each field of the object is investigated. In case of an update, the state of the old object is copied field-wise to the new instance using the method *SetValue* of the .NET Reflection API. Object references are traversed via a recursive call of *UpdateObjectGraph*. The return value of the method call (a potentially updated reference) will be injected into the newly created object's field. In case of a primitive typed field (int, string, byte ...) the value is copied directly into the new object. Because the state of an object is represented by its fields only, the state transfer is complete after the investigation of all fields.

```
long id = idGenerator.GetId(node, out firstTime);
if (!firstTime)
{
    return visitedNodes[id];
}
Type newType = GetUpdate(type.GetType(), updateMapping);
if (newType != null)
{
    object updatedNode = FormatterServices.GetUninitializedObject(newType);
    visitedNodes.Add(id, updatedNode);
}
else
    visitedNodes.Add(id, node);
...
// Field-wise traversal
...
```

Fig. 4. Cycle Recognition for cyclic dependencies in the object graph

Cycles in the object graph are another challenge during graph traversal. The described implementation would run into a live lock in case of cycles. Figure 4 shows the implementation of our cycle recognition algorithm. We save each traversed object in the hashtable *visitedNodes*. Each object is identified by a unique number generated by an *IDGenerator*. The value in the hashtable is either the object reference itself or a reference to an updated version of the object. If an object is visited a second time, the value from the hashtable will be returned. The caller installs the potentially new reference found in the *visitedNodes*-table into a field, in case of an update.

Finally, we'll describe the handling of delegates, because they are important for the update of graphical components. Delegates are function pointers in .NET. They can point to static or instance methods. Basically a delegate contains a *MethodInfo* identifying the target method and a *target* pointing to an object (or null in case of a static method). Both delegate members must be investigated for updates. Delegates are updated by creating a new delegate using *Delegate.Combine* with the new *target* and/or *MethodInfo* as parameter. The new delegate will be injected into the object graph, afterwards. Multicast-delegates are containers for multiple delegates - they are investigated as well. Due to the quite complex handling of delegates we are not able to display source code here.

3.2 Updating controls

Graphical user interfaces in .NET are created using the *Windows.Forms* library. This library uses the event subsystem of .NET. Building blocks for graphical user interfaces are derived from the class *Windows.Forms.Control*. Event handler can be defined for each control for a variety of events such as mouse clicks, mouse movements, key presses and so on. Event handler are saved as delegates that are invoked if an event is signaled.

When updating controls we want the updated version of the application to have the layout and behavior as defined in the configuration. Because at least the layout is part of the state that is transferred from the old version, the new layout must be activated separately. During the update process additional code can be executed for nodes of certain types. This code is called *update handler*. Depending on the type name *update handler* can be registered in the update infrastructure. After completion of the state transfer this code will be activated to post-process the object graph.

In case of graphical components (*UserControls*) the update handler creates an instance of the new version, also executing the default constructor. The result is called *reference control*. The updated state (layout) of the control is injected into the node (still containing the old state, but already running in a new version) of the object graph. For controls this means that location, font, colors, and size are adjusted. In addition all sub-controls, which are saved in fields of the control, are updated according to the sub-controls of the *reference control*. The content of all controls remains the same as in the old version, only the layout is updated. In addition new sub-controls are added and old ones are removed in accordance to the *reference control*. Behavior is already updated during the graph traversal phase, because all occurrences of references to the updated control are replaced by the new version - including delegates defining event handler for control events.

Beside the central idea of the algorithm there remain some additional items to discuss. In order to support the handling of new events and the addition/change of event handlers these must be processed specially.

Fig. 5 shows parts the graphical sub-system of .NET. On the left you can see the main object of a graphical component - a *UserControl*. This control has been interwoven with the *ReconfigurationAspect*. In addition to the functional methods, the proxy methods are shown. The control contains one sub-control - a button. Each control is attached to a native window in the Win32-subsystem, that executes the main message dispatch loop². Window messages arrive at the native window and are forwarded through the *WndProc* each control implements. Within this method messages are dispatched to .NET events handled by each control. Each event contains a list of event handlers which are invoked if an event is signaled. The native window holds a reference to the interwoven control. Each method invocation from the Win32-subsystem is caught by the reconfiguration aspect, allowing for synchronization with the update logic.

² The Win32-subsystem works by exchanging messages. A process can implement a main message loop, that reads messages one after another and executes according actions (e.g. redrawing a window)

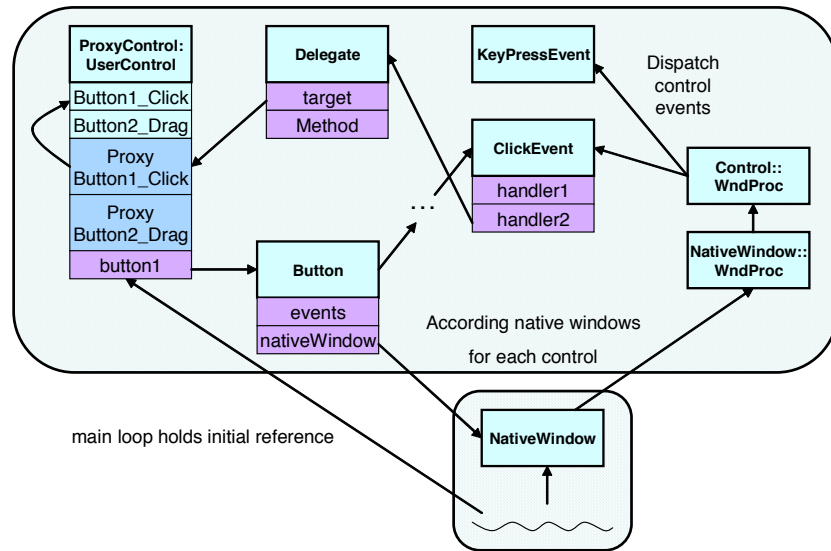


Fig. 5. Update of Windows.Forms.Control - Win32 sub-system and .NET

During the update process a new version of the control is created including the reconfiguration aspect. While the native window always holds a reference to the original control proxy all delegates are updated to point to the newly created proxy. If a delegate is invoked, the invocation goes through the aspect code again, allowing for synchronization with the update logic. This means, that there are always two instances of the proxy running after the first update. The initial one, processes calls from native threads, while all event handlers directly point to the latest version of the control proxy. Method calls to the initial proxy are also forwarded to the second, latest proxy (that includes the latest implementation) - containing new behavior for handling control events.

Event handlers are saved in a linked list in the *event* member of each control. In order to support changes in the set of event handlers, this list must be updated. We use .NET Reflection to iterate through the event list within the update handler code. Each entry contains a *key* that is used by *WndProc* to dispatch events. In addition each entry contains a delegate, that represents the event handler. The lists contained in the object graph on the one hand and in the newly created *reference control* on the other hand must be balanced. New event handlers in the reference control are transferred into the object of the object graph, while missing entries in the reference control are removed in the object graph's object. The delegate targets of new event entries in the list are adjusted to point to the new proxy. In addition the content of the (multicast)-delegates itself must be adjusted according to the *reference control*.

Another challenge for updating controls is, that they can only be manipulated in the context of the thread they have been created in. This is due to the Win32-

subsystem, that saves information in the thread's context. The configuration manager is typically executed in another thread than the application components themselves. That's why the update of graphical components is executed deferred. During the reconfiguration phase a flag is set in the reconfiguration aspect code. The update will be executed just before the execution of the next functional method call - in the context of the thread that created the control. The update is deferred until the control flow enters the component. Because there are always two proxies after the first update, a member of the reconfiguration aspect always points to the initial proxy, because this is the root of the object graph to be updated.

4 Related Work

A number of papers on dynamic reconfiguration of software has been published within the last years. Our solution has advantages over these according to ease of use, runtime overhead and the support of multi-threaded applications.

M. Wermelinger presents a theoretical approach [9] to reconfiguration extending an algorithm of J. Maggee and J. Kramer [6] that influenced our work. 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 [3] 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.

Trap/J [11] uses a meta-level for guiding component configuration similar to our approach. In contrast to ours this solution integrates reflective method binding into the functional application logic, causing high overhead. During compile time, applications are prepared for adaptation by making them adapt-ready. During runtime an adaptation manager provides information about environmental conditions, which are used by decision logic integrated into the application logic. These logic selects among alternative method to implement adaptive behavior using the reflection mechanism of the Java language. In contrast, our approach uses reflection only during the reconfiguration phase.

The JBoss application server [7] includes a framework for aspect-oriented development called JBoss-AOP. Dynamic AOP of JBoss-AOP supports the instrumentation of Java classes at load-time through the manipulation of used class-loaders. This techniques allows for the implementation of an configuration aspect similar to our introduced approach. The JBoss-AOP hot-swap mode [1]

allows to enable aspects during runtime of an applications. Application classes must be prepared before runtime. The hot-swap mode is implemented using the *java.lang.instrument* instrumentation features of Java 5. The addition of new aspects is trigger through a singleton component - the Aspect Manager.

There are other approaches that in contrast to our solution extent existing languages or manipulate virtual machine implementations. Adaptive Java [4] introduces new keywords to the Java language to implement adaptive behavior. QuO [10] is a middleware for developing distributed applications. The developer defines alternative methods, invoked in different situations.

5 Conclusions and Future Work

We have described our framework *Adapt.Net* for building adaptive, component-based applications. The handling of configuration-specific details using aspect-oriented programming relieves the application developer from concerning complex synchronization details and the manipulation of configuration-specific data. Our framework also provides an adaptation engine - together with a monitoring infrastructure adaptation decisions are triggered automatically based on XML-based adaption profiles. Our framework provides a very convenient way to build adaptive applications.

In this paper we focused on an important reconfiguration option: dynamic updates of component implementations. Besides logic for synchronization we introduced a graph-based algorithm for field-wise state transfer, both handled on a meta-level. This causes only minimal overhead to functional application logic. The implemented synchronization logic is able to handle multi-threaded applications an important step for future multi-core processor architectures.

The adaptation of graphical user interfaces allows to reflect changes in the resource availability to the user. We provide a generic algorithm that can be used by programmers out-of-the-box to implement updatable applications including updates of graphical components. We have made no manipulations to the virtual machine. Our approach can be used on the commercial implementations of the .NET framework and can also be applied to Java, without restrictions.

Due to on-going performance optimizations we will present an evaluation of our approach in a future publication. We are also working on a case study for our approach evaluating the effects of changes of the user interface at the user itself. Currently we are investigating runtime updates and the dynamic activation of aspect implementations - another important option for building adaptive applications.

References

1. Alex Vasseur. dynamic AOP and HotSwap. http://blogs.codehaus.org /people/avasseur/archives/ 000615_dynamic_aop_and_hotswap.html, 2004.

2. Andreas Rasche and Andreas 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.
3. Davy Suvée and Wim Vanderperren and Viviane 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.
4. E. Kasten and P. K. McKinley and S. Sadjadi and R. Stirewalt. Separating Introspection and Intercession in Metamorphic Distributed Systems. In *Proceedings of the IEEE Workshop on Aspect-Oriented Programming for Distributed Computing (with ICDCS'02)*, Vienna, Austria, July 2002.
5. 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 Mehmet Akşit and Satoshi Matsuoka, editor, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
6. J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
7. JBoss Community. JBOSS Website. <http://www.jboss.org>, 2005.
8. M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982.
9. Michel 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.
10. Rodrigo Vanegas and John A. Zinky and Joseph P. Loyall and David Karr and Richard E. Schantz and David E. Bakken. QuO's Runtime Support for Quality of Service in Distributed Objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 15–18, The Lake District, England, September 1998.
11. S. Masoud Sadjadi and Philip K. McKinley and Betty H.C. Cheng and R.E. Kurt Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. In *Proceedings of the International Symposium on Distributed Objects and Applications (DOA'04)*, Agia Napa, Cyprus, October 2004.
12. Wolfgang Schult and Andreas 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.