Dynamic Reconfiguration of Component-based Real-time Software

Andreas Rasche and Andreas Polze

Hasso-Plattner-Institute at University of Potsdam

14482 Potsdam, Germany

{andreas.rasche|andreas.polze}@hpi.uni-potsdam.de

Abstract

Increasing capabilities of modern microcontrollers greatly increase their applicability to more and more complex scenarios. However, unstable and ever-changing environmental settings require embedded systems permanently to adapt to new situations. Dynamic reconfiguration provides a powerful mechanism to execute such adaptation strategies. The implementation of dynamic reconfiguration is still challenging for embedded real-time control software.

Within earlier work we have presented our framework Adapt.NET for runtime adaption of component-based applications, including a runtime infrastructure for dynamic reconfiguration and monitoring, targeted for mobile and desktop environments. New experiments in our web-based remote laboratory - the Distributed Control Lab - require the reconfiguration to complete in bounded time. In the remote lab we use dynamic reconfiguration to adapt experiment control to failures in user control components.

Within this paper we will analyze the timing behavior of the implemented dynamic reconfiguration algorithm in order to allow for predictable execution times. We describe how complex component-based real-time applications can be adapted to changing environmental conditions, continuously meeting all tasks deadlines during dynamic reconfiguration.

1 Introduction

Embedded systems are increasingly facing changing environmental conditions. During their lifetime, adaptation to changing parameters such as available battery power, varying communication bandwidth, available memory or faults in software components must be considered in order to preserve desired application level quality of service.

Nowadays embedded systems commonly are reconfigured during an offline phase. For doing that, running control software is stopped when new software configurations are installed onto embedded devices. The controlled physical process has to be stopped, causing high operational overheads. Even worse, in some cases stopping control systems could cause fatal system failures and even harm people.

Online changes to real-time control software require a reconfiguration to preserve application consistency and meet timing constraints of all real-time tasks.

Within earlier work [10] we have presented the adaptation framework *Adapt.NET* including tools for building distributed component-based applications, a monitoring infrastructure and a runtime environment capable of executing dynamic reconfiguration commands. Adaptation to changes in the application's environment is conducted by monitoring it and loading new application configurations, if requested by a pre-defined adaptation policy. An *application configuration* denotes the set of its parameterized components and the connections among them as well as a mapping onto execution hosts in case of distributed applications.

After implementing the reconfiguration infrastructure we investigated the automatic generation of reconfiguration specific code to relieve the application developers from implementing adaptation details. Tools provide support for developing adaptation profiles, which are mappings of environmental conditions to corresponding application configurations. Developers can choose from a variety of architectural styles that support application adaptation. Examples are filter, voter, and encryption components that minimize response time, improve reliability and security respectively. Code skeletons for these components can be generated automatically.

The Adapt.NET framework is currently being used in the Distributed Control Lab (DCL) [11] operated at the Hasso-Plattner-Institute. The DCL provides an open infrastructure for conducting robotics and control experiments via the Internet. In the lab, we use analytic redundancy and dynamic reconfiguration to ensure the safety of the lab's experiments. Potentially malicious code, downloaded from the Internet, could damage expensive physical hardware or manipulate involved computer systems. During the runtime of user control components, experiment parameters and the behavior components are monitored by observer components. If any

faults of user control components are detected, the control application can be reconfigured to hand over control to a verified safety controller.

Among others, we have applied our reconfiguration strategy to the Foucault's Pendulum experiment - without guaranteeing hard deadlines for reconfiguration. The experiment is controlled by a x86-PC running Windows 2000 as operating system. Real-time requirements of the physical experiment are being met by using the real-time scheduling class of the Windows 2000 scheduler in conjunction with external custom hardware. This powerful execution environment allows us to rely on the concept of .NET's *code access security* to implement a secure runtime environment *Sandbox* for user control components.

New hard real-time control experiments in the DCL require the reconfiguration of control software to be performed in bounded time. Within this paper, we will present the dynamic reconfiguration approach used, and describe how real-time control applications can be reconfigured meeting all tasks' deadlines. We will discuss, which steps in the reconfiguration process exert influence on final reconfiguration time.

The remainder of the paper is structured as follows: The next Section 2 describes our Adapt.NET framework for distributed application adaptation. Followed by that, a general model of computation for applications is given. Dynamic reconfiguration of applications, following the model ensuring application consistency, will be described in Section 3. In Section 4, the real-time characteristics of the presented algorithm and a timing analysis are presented. Strategies for reconfiguration with state transfer are discussed in Section 6. In Section 7, related work is presented. Finally, we will present our conclusions and an outline for future work.

2 The Adapt.NET Adaptation Framework

Within earlier work we have implemented a framework for dynamic reconfiguration of middleware objects. Using an XML-based configuration description language [9], describing components, their parameters, connections and a host mapping, our implementation is able to identify components and connections involved in the reconfiguration process and perform the appropriate reconfiguration commands.

Figure 1 gives an overview of our reconfiguration infrastructure called *CoFRA*. When an application is started, a CoFRA daemon at the initiating host evaluates the application's adaptation profile, initializes the monitoring infrastructure by instantiating observer components for requested environmental properties, and, finally, loads the application configuration matching the current environmental conditions.
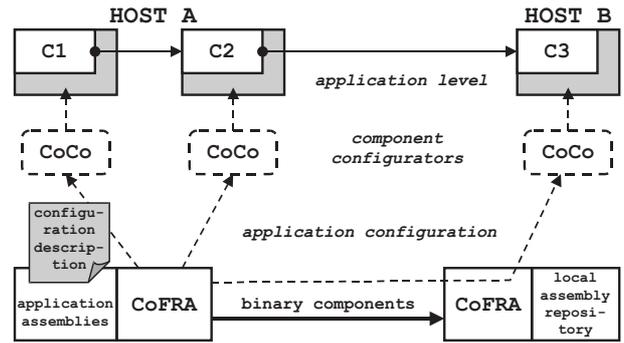

Figure 1. Reconfiguration Infrastructure

In case of distributed applications binary .NET components are transmitted via the CoFRA infrastructure onto a distributed computer network and instantiated as described by the configuration description document. This approach simplifies the deployment of distributed applications, because all dependencies among binary software components are resolved by the configuration framework using one central component repository. We have also implemented versioning facilities and are able to upgrade component instances at runtime.

Each component has to implement a reconfiguration-specific interface containing methods for connecting components, setting component parameters or transferring component state. Our tools are able to generate this interface and other configuration details for a given application component. Component developers merely will have to use simple hooks to indicate transactions (see Section 4). In figure 1 the gray parts of the application components indicate added configuration code.

Components within our framework can be instantiated in different ways. We distinguish several *component types* which allow for the instantiation of components either within separate processes, or as light-weight objects running in the context of our CoFRAs, or register them as remote objects. Component configurators hide complexity when accessing different component types. For example, if a component runs within a separate process, a communication channel has to be established in order to access the configuration interface of the component instance. This additional code will be provided by our framework, a developer will simply implement a class as usual and instantiation, distribution and configuration will be handled by our configuration framework.

Our current implementation was originally based on Microsoft's .NET platform. However, we have already introduced new component types to be able to integrate Java and CORBA objects into applications configured by our framework. Communication among .NET, CORBA and Java

components is realized using the IIOP.NET [2] interoperation framework. Instances of other component frameworks can easily be integrated.

Typically, adaptive applications involve the monitoring of environmental settings and require a strategy for actual adaptation. During runtime of an application, environmental properties are observed and in case of changes the application will be reconfigured.

We distinguish three categories of monitored parameters:

- environmental conditions - parameter outside the application (e.g. available memory, CPU power, network bandwidth);

- state of application components (e.g. crashed components, can be detected by flipping a variable each period of execution, if a no-flip is detected component has crashed);

- attributes of components - used to observe internal component state (e.g. internal counter, a flip-bit for state observation).

We define a number of application level quality-of-service properties that are adjusted by our reconfiguration framework. Adaptation primarily aims at ensuring that these properties are in a pre-defined range by e.g. adjusting the frame-rate of a movie-player or the memory usage of a component. There is a relationship between an application configuration and particular values of these quality-of-service parameters. But this relationship is machine-specific and often not known a priori.

There are methods to calculate the influence of components, their connections, and their parameters as well as the component host mapping on application level parameters. For example, if we consider a client-server multimedia application, the insertion of a component that compresses the data stream may optimize the end-to-end response time in case of limited communication bandwidth between client and server. However, predicting the effects of particular configuration decisions on the resulting application level parameters is an interesting open research topic. We are going to investigate this issue in a future work.

Within our framework, *adaptation policies* define a mapping of monitored parameters to application configurations. Our tools support the definition of such adaptation profiles and we are also investigating the generation of adaptation policies based on defined application level quality of service parameter and component properties.

We provide a graphical tool to build application configurations out of a given set of functional components. The tool generates configuration interfaces for each component and, finally, creates a deployable package for release. Our graphical tool also supports the integration of architectural patterns into created application configurations,

which provides a powerful way to support developers in order to achieve the desired application level quality of service properties. Our tool supports the generation of "glue code" in order to support the above-mentioned architectural patterns. During runtime of an application, our monitoring infrastructure observes environmental settings using a pluggable component-based architecture. Observer components implement specific measurement code for selected environmental properties. These observer component are reusable between different applications. If significant changes are detected[*], whose importance can be specified for each observer, a reconfiguration request is sent to the underlying CoFRA infrastructure which executes the reconfiguration immediately.

3 Model of Computation

Our dynamic reconfiguration framework is based on an algorithm first introduced by Kramer and Magee [3] and improved by Wermelinger [15], who discussed the topic theoretically.

Following their original work, we model applications as interconnected computational entities, called components. Components provide interfaces, namely *in-ports*, and require other components connected to their *out-ports*. We distinguish between active components, including a thread of control, and passive components only activated on request of other components. Application topology can be depicted as a directed graph whose nodes are components and whose arcs are connections among them. Each component of the application must be connected to at least one other component, cycles in the application graph are not allowed - the graph must be acyclic.

A component's state changes through the interaction with other components. In order to preserve global application consistency during a reconfiguration, the communication among components must be tracked. To cope with ongoing interactions during a reconfiguration, the concept of a transaction is introduced. A transaction virtually combines a number of bidirectional interactions between components. A transaction completes in bounded time and its initiator will be informed about its completion. A transaction is said to be *dependent* on a subsequent transaction if its completion depends on the completion of the other transaction.

Figure 2 illustrates four connected components, including transactions flowing along the connections. Transaction T1 depends on the subsequent transaction T2 as indicated by T1/T2. Transaction T1 cannot be completed until T2 has finished. Considering a client-server application and an integrated proxy. Interactions between client and

---

[*]The application developer marks parameter tresholds by the definition of adaptation policies.
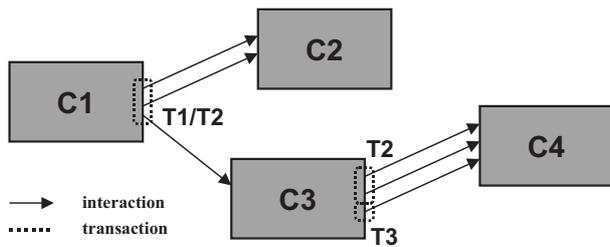
Figure 2. Application Model

proxy depend on the subsequent communication between proxy and server. The initial transaction can only be completed when the transaction between proxy and server is completed. Each arrow between two components indicates one interaction. A transaction is marked by a dotted line around a number of interactions.

This application model allows for the definition of a reconfiguration algorithm on an abstract level. On the implementation level, high flexibility is possible. We have implemented a variety of connectors ranging from simple shared memory and TCP/IP sockets to .NET Remoting and CORBA connectors, all of them exchangeable during runtime using dynamic reconfiguration. Interactions among components are executed via connectors. Components can be implemented as separate processes, threads, CORBA objects, or component of any arbitrary component framework.

In the following Section 4 we are going to describe an algorithm to reconfigure distributed component-based applications that follow the model introduced.

4 Algorithm for Dynamic Reconfiguration

Dynamic Reconfiguration of component-based applications can involve several atomic operations, including the addition of components, the removal of components, and the adjustment of component parameters. These simple operations have the advantage that no state of component instances (i.e. state of the objects residing inside a component) has to be transferred. More complex operations include component updates and the migration component instances to other hosts in distributed applications at runtime, where state must be considered.

In order to guarantee continuous service after reconfiguration, the application must be kept in a consistent state (also called reconfigurable state). In literature [7], several levels of consistency are described. *Local component consistency* addresses state transfer. *Global consistency* defines application invariants that must not be violated during a reconfiguration. Finally, *structural consistency* deals with integrity among application component interfaces. All ports of the affected components must be connected and the types

of connected in- and out-ports must match. Structural integrity can be checked by tools statically, while local and global consistency must be ensured by the reconfiguration algorithm during runtime.

The original article by Kramer and Magee [4] describes a process called "freezing" of application components, which includes stopping of the whole component activity. Wermelinger improved the algorithm by blocking only connections between involved components ([15]). An advantage of this approach is that interruption time is minimized while only affected connections must be blocked in contrast to whole components.

Wermelinger argues further that a reconfigurable state can be reached if all connections involved in a reconfiguration are blocked. A connection is blocked by blocking all ongoing transactions using this connection. Finally, a transaction is blocked by waiting for all ongoing interactions to complete and not allowing the initiation of new transactions. In order to guarantee that all transactions can be blocked, the blocking must be ordered. Otherwise application deadlocks could occur. Interdependent transactions must be blocked in the order of their dependency.

When the application is finally blocked, new components can be added, components can be reconnected and component parameters can be changed. In order to reduce the blackout time during a reconfiguration, we decided to move as many steps of the reconfiguration process as possible before and after the blocking phase resp.. Starting (i.e. loading and instantiating) new components can be performed before the blocking process is initiated. Removal of components is realized only after the new application configuration is effective.

5 Reconfiguration of Real-time Applications

In our introduction we have stated that reconfiguration of real-time control software must preserve application consistency and meet all deadlines of application tasks. The reconfiguration algorithm described before is able to realize the consistency requirement. Within this section we are aiming to demonstrate that all application tasks will meet their deadlines under our particular reconfiguration algorithm, if there is time left on CPU to schedule reconfiguration commands. (The reconfiguration is considered as an additional task which must be scheduled during runtime.) The execution of reconfiguration commands itself has bounded execution time.

Typically, real-time applications perform their work in a periodic way [5]. In order to perform a dynamic reconfiguration of an application, processor resources for potential reconfiguration commands must be reserved. We will demonstrate how the required resources can be calculated before runtime of the application and budgeted in appli-

cation design and implementation. Observance of certain rules reduces overhead in time to a minimum.

A reconfiguration, typically, also involves non real-time activities. We will demonstrate that our algorithm performs the central reconfiguration that involves a blocking of the application executes in bounded time. Activities such as loading of new components and removal of old components may require undefined execution time and will therefore be performed before and after the blackout phase respectively. There are some exceptions when these strategy cannot be applied - for example if there is a resource conflict between new and old components. We will discuss this issue later in the text.

Of course there is always a trade-off between reconfiguration and blackout time. In our approach we optimized blackout time, which only takes influence on real-time task's deadlines.

At first, we will separate the reconfiguration process into single steps, which can then be analyzed in detail.

- t_r - time between the reconfiguration request and the new configuration to run

- t_b - blackout time - time the application is interrupted

- t_l - time to load and initialize new components

- t_d - time to delete components not contained in new configuration

The reconfiguration time of the whole process can be calculated in order of the single steps' occurrence as follows:

$$t_r = t_l + t_b + t_d$$

As loading of new components is realized in free slots during normal operation of the real-time code, we have to consider the blackout time in more detail. During this time the application will not be able to process any data. That's why this time must be integrated into the schedule when designing the application.

The blackout time is made up of the following steps:

- t_e - time required to execute reconfiguration commands;

- t_i - maximum timespan to complete transaction i;

- n - number of transactions in the application;

- t_{state} - time to transfer state;

- t_{ci} - time to initialize and start connectors;

- t_s - time to restart the application.

The blackout time is calculated as follows:

$$t_b = t_e + max(t_1, ..., t_n) + t_{state} + t_{ci} + t_s$$

In our initial calculation, we will not consider the transfer of component state. The reader is referred to section 6. It can easily be seen that the blackout time with remaining terms is bounded if each of term is bounded in execution time.

The execution time of reconfiguration commands depends on the number of involved components, the number of their attributes, and connections. Each of these commands involve a restricted number of variable assignments, whose execution is bounded in time. We do not consider caches or memory access latency because these have to be analyzed in the base application design phase.

The calculation of the number of involved components and connections/transactions to be blocked is performed before the blocking phase. A schedule of reconfiguration commands is generated in a free CPU slot before a reconfiguration is initialized. If all application configurations are known before runtime, all these schedules can be created offline.

Blocking connections among components involved in a reconfiguration includes the completion of all ongoing transactions over those connections. Configuration-specific code in each component is able to wait for ongoing transaction to end, and to prevent new transactions from starting. In case all transaction are requested to finish at the same time (parallel)[*], the blocking of transactions is bounded by the time the longest transaction requires to complete which is indicated by $max(t_1, ..., t_n)$. This time depends on interaction times of involved components. Upper bounds for interaction times must be found by analysis of application configurations before runtime. If the application developer adheres to certain programming rules, the time to stop transactions can be reduced to a minimum. In general only applications with short interaction times (bursts) in contrast to long computation and sleep phases - a behavior which is typical for real-time software - can be dynamically reconfigured with our approach.

Finally, the restart of components and the recreation of connections involves a single assignment per component and connector respectively.

On a task level, there must be enough CPU resources to perform reconfiguration commands, whose execution time is bounded, as discussed above. It is possible to use mode change protocols [14] to integrate reconfiguration specific tasks with a given task set. Additional tasks to load new components and remove old ones have to be budgeted as well as tasks added by new components.

---

[*]In fact not all transactions can be blocked in parallel, because of transaction dependencies, which must be ordered, but blocking a chain of dependent transactions takes as long as the blocking the longest transaction in the chain.
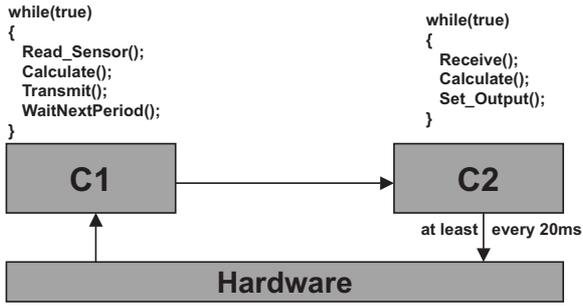
```
while(true)
{
  Read_Sensor();
  Calculate();
  Transmit();
  WaitNextPeriod();
}
```

```
while(true)
{
  Receive();
  Calculate();
  Set_Output();
}
```

C1 → **C2**

C2 at least | every 20ms →

Hardware

Figure 3. Simple Real-Time Application

```
while(true)
{
  Read_Sensor();
  Calculate();
  Transmit();
  WaitNextPeriod();
}
```

log data

```
while(true)
{
  Receive();
  Calculate();
  Set_Output();
}
```

C1 → **C3** → **C2**

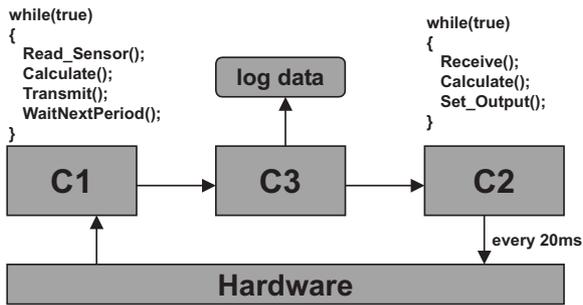**C2** every 20ms →

Hardware

Figure 4. Added Debug Component

In fact, the reconfiguration approach matches mode change theory, if we assume tasks as components and the synchronization among them as connections. Transactions introduce additional synchronization delays as they are described in mode change protocols. The advantage of our algorithm is that we are independent of the actual implementation; we are able to handle any kind of connections among components and are able to reconfigure their usage at runtime.

Figure 3 shows a typical real-time control application. One component processes data of an external sensor and transmits results to a second component, which must generate output signals each 20ms. Figure 4 shows a third component being inserted. This component generates logging data for debugging purposes. The application has to be reconfigured dynamically, because it may presumably not be shut down for the insertion of additional debug components. If we assume a worst-case execution time of 5ms for the first component and another 5ms for the second, we have 10ms time for reconfiguration if the first component started reading sensor data immediately after the second component produced an output signal. Of course, the logging component is not allowed to use more than 5.6ms if we assume a rate monotonic scheduler, which requests an upper bound for CPU utilization.

6   Reconfiguration and State

We have not yet considered state transfer to new component instances that are affected by migration or component updates. State transfer must be considered to preserve local component consistency. The state transfer has to be performed in the blackout period, thus increasing reconfiguration-related disruption. During timing analysis of the reconfiguration process, we have already considered state transfer time, but, not yet discussed its implication on the blackout phase.

When analyzing reconfigurations involving state, we have to distinguish between reconfiguration trigger through the fault of a component, and an adaptation request for a healthy application. There are several methods to handle state transfer in these scenarios.
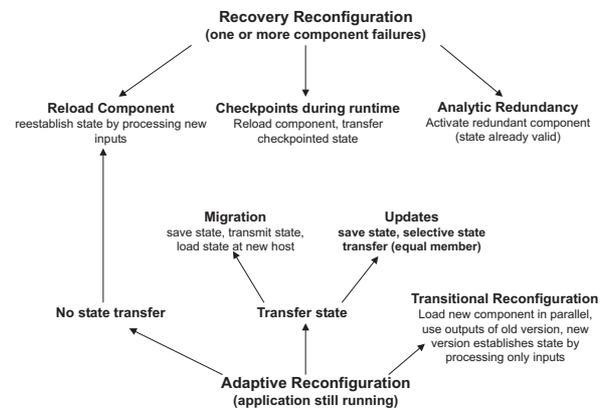
Recovery Reconfiguration
(one or more component failures)

Reload Component
reestablish state by processing new inputs

Checkpoints during runtime
Reload component, transfer checkpointed state

Analytic Redundancy
Activate redundant component
(state already valid)

Migration
save state, transmit state, load state at new host

Updates
save state, selective state transfer (equal member)

No state transfer

Transfer state

Transitional Reconfiguration
Load new component in parallel, use outputs of old version, new version establishes state by processing only inputs

Adaptive Reconfiguration
(application still running)

Figure 5. Reconfiguration and State

Figure 5 gives a general overview of handling state during a reconfiguration. If component instances crash, all runtime state of that component is lost. A reconfiguration could be triggered to reestablish a valid configuration of the application. Real-time software typically periodically reads input from external sensors, processes data and generates according output signals. If there is no persistent component state, the crashed components simply can be reloaded and integrated into the remaining application structure.

One method to deal with state of crashed components is to save it periodically. Such checkpoints can be used to initialize new component instances in case of failures. This method has drawbacks in overhead and global application consistency is difficult to maintain. We have not been considered this method for our analysis yet.

A much better method to handle component faults is the usage of analytic redundancy. With analytic redundancy, variants of components implement different control algorithms that all produce valid outputs. If one of the redundant components fails, the control application can be reconfigured to use a backup component, which then is able to

produce correct output signals immediately. State does not have to be transferred because the redundant components keep track of control state during parallel execution. We have currently been using this approach for our experiments in the Distributed Control Lab.

If reconfiguration is requested by an adaptation policy due to changes in the environment, the state of component instances is still available. If the reconfiguration requires components to be migrated or updated, state of old component instances must be transferred to new ones.

In our existing implementation, state transfer was realized by using a reflection interface for component instances and simply copying values of all instance members recursively to migrated and updated instances respectively. In case of new members in new component versions, these fields are initialized with default values. Missing members are not considered any more and their state is left over.

Time required for transferring state from one component instance to another depends proportional on the state's size. E. Schneider et. al. [12] present an interesting approach to deal with this problem. With their approach, the state of services is transferred to updated service instances during a reconfiguration. If there are requests that must be handled before state transfer completes, these requests will be processed first and the state transfer is restarted afterwards. This approach works fine if work load in single periods varies significantly.

Otherwise the maximum instance size of a component must be specified by component programmers. State size is always bounded by available memory in the target system. All our considerations assume that there is always enough memory to load new components and transfer state among them.

In *transitional reconfiguration* new and old versions of components run in parallel for a specified amount of time. While the old instance produces correct output signals, a new version establishes runtime state needed for control. After a specified time, a second reconfiguration step activates the target configuration and removes old component instances.

In some cases no state transfer is necessary at all. In such cases new components can be instantiated prior to the blackout phase; old component instances are remove when the new configuration already runs.

7   Related Work

The original work by M.Wermelinger has been on a theoretical basis and was not targeted at real-time control software. Initially we focused on the implementation of Wermelinger's algorithm, which later on has been adapted to our problem specification.

There is a body of related work that deal with dynamic reconfiguration of real-time software as well. We will discuss the most prominent concepts within this Section.

The Swift toolkit [1] enables dynamic reconfiguration of feedback control applications by limiting component interaction to a simple input/output model. Reconfiguration is performed by changing component parameters and re-plugging of controllers. Adaptive behavior is reached by guarding system parameters.

E. Scheider et. al. [12] describe the dynamic reconfiguration real-time service-based software through the OSA+ middleware. With OSA+, service reconfiguration is scheduled as a normal task. This allows the specification of deadlines and priorities for reconfiguration requests. During reconfiguration of a service, the old services' state is transferred to a newly loaded instance. If any service requests have to be scheduled during this phase, because of a close deadlines, state will get inconsistent. That is why the reconfiguration task restarts the state-transfer to preserve service consistency. If the state has been transferred without interruption, the new service instance will be activated - new service request then are forwarded to the new service instance. This is a very powerful and general approach for service-based architectures, however, the reconfiguration of services is very coarse grain in contrast to a reconfiguration based on application objects, as presented in this paper.

M. Pfeffer and T. Ungerer present an approach for dynamic reconfiguration on a multithreaded Java-microcontroller, Komodo [8]. The Komodo microcontroller is able to execute Java bytecode directly and run multiple threads parallel in hardware. Single Java objects can be updated using a separate thread. Updates are realized by overwriting an object's code memory section and updating method invocation tables and object data if required. The authors give a worst case switching time of 246 clock cycles. With their approach, it is only possible to update one object at a time.

The Simplex [6, 13] approach introduced by Lui Sha et. al. describes a special case for dynamic reconfiguration, which has already found its way into highly responsive systems like the Boing 777 flight controller. The idea is to use analytic redundancy for control. With Simple, faults in a high performance controller trigger a dynamic reconfiguration that activates a more reliable controller instance implementing a simpler control algorithm, but still producing valid output signals.

Mode Changes [14] for priority-driven scheduling algorithms describe task set reconfiguration for real-time systems. Tasks can be added, modified or removed from an existing task set. Mode change protocols define rules when reconfiguration can take place and how, including the adjustment of task synchronization (e.g. ceiling value adaptation in the priority ceiling protocol). The reconfiguration

from one task set to another is split into single operations (add, modify, remove), which are executed sequentially. A reconfiguration can last several task periods. Our approach is equivalent to mode changes if we consider tasks as concrete component instances and synchronization as their interaction. Our approach is more general.

8 Conclusions and Future Work

Within this paper, we have presented our framework for dynamic reconfiguration that allows applications to adapt to changing environmental conditions. Based on prior work we have here described the steps necessary for reconfiguration. Furthermore, we have analyzed their timing behavior, which can be calculated for many application configurations.

Using our approach it is possible to reconfigure complex component-based applications with complex component dependencies. Dynamic reconfiguration is not limited to the simple exchange of a single components, but instead allows for dynamic modification to complex parts of an application. Reconfiguration and state transfer have been discussed for different scenarios. We have analyzed the impact of these procedures on the time bounds of black-out periods.

We have demonstrated that the blackout time during reconfiguration is bounded and can be calculated for given applications. This allows for the construction of real-time software that is reconfigurable dynamically, while all task's deadlines are met. It is possible to show that the used reconfiguration algorithm a generalization of the mode change theory, which has been theoretically proven.

The presented adaptation strategy has been integrated into an experiment controller within the Distributed Control Lab, a remote laboratory, to ensure safety against malicious code downloaded from the Internet.

We have implemented and evaluated the described approach based on standard middleware components. We are now going to implement a new version of our framework for real-time control software. We plan to report on measurements in a separate publication and will describe our experiences made in context of our remote lab.

References

[1] A. Goel, D. Steere, C. Pu, and J. Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. In *Proceedings of the 2nd Usenix Windows NT Symposium*, pages 172–172, September 1998.

[2] IIOP.NET Homepage. .NET, CORBA and J2EE Interoperation. http://iiop-net.sourceforge.net, 2004.

[3] S. E. J. Magee, N. Dulay and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989, pages 137–153, Sitges, Spain, 1995. Springer-Verlag, Berlin.

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

[5] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[6] M.Bodson, J. Lehoczky, R. Rajkumar, L. Sha, M. Smith, D. Soh, and J. Stephan. Control reconfiguration in the presence of software failures. In *Proceedings of the 32nd IEEE Conference on Decision and Control*, volume 3, pages 2284–2289, 1993.

[7] N. D. Palma, P. Laumay, and L. Bellissard. Ensuring dynamic reconfiguration consistency. In *6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*, pages 18–24, Budapest,Hungary, June 2001.

[8] M. Pfeffer and T. Ungerer. Dynamic real-time reconfiguration on a multithreaded java-microcontroller. In *International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, pages 86–92, Vienna, Austria, 2004. IEEE.

[9] A. Rasche and A. Polze. Configurable Services for mobile Users. In *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*, pages 163–171, San Diego, CA, Januar 2002.

[10] 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.

[11] A. Rasche, P. Tröger, M. Dirska, and A. Polze. Foucault's Pendulum in the Distributed Control Lab. In *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*, pages 299–306, Capri Island, Italy, October 2003.

[12] E. Schneider, F. Picioraga, and U. Brinkschulte. Dynamic Reconfiguration through OSA+, a Real-Time Middleware. In *Middleware 04, 1st Middleware Doctoral Symposium, Toronto, Canada*, 2004.

[13] L. Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[14] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1989.

[15] 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.