ReDAC - Dynamic Reconfiguration of distributed component-based applications with cyclic dependencies

Andreas Rasche and Andreas Polze
Hasso-Plattner-Institute at University of Potsdam
14482 Potsdam, Germany
{andreas.rasche|andreas.polze}@hpi.uni-potsdam.de

Abstract

This paper introduces ReDAC, a new algorithm for dynamic reconfiguration of multi-threaded applications with cyclic dependencies. In order to achieve high reliability and availability, distributed component software has to support dynamic reconfiguration. Typical examples include the application of hot-fixes to deal with security vulnerabilities. ReDAC can be implemented on top of the modern component-platforms Java and .NET. We extend the statical term component, denoting a unit of deployment, to runtime by defining a capsule (runtime component instance) to be a set of interconnected objects. This allows us to apply dynamic updates at the level of components during runtime without stopping whole applications. Using system-wide unique identifiers for threads (logical thread IDs), we can detect and also bring capsules into a reconfigurable state by selectively blocking threads, relying on data structures maintained by additional logic integrated into the capsules using aspect-oriented programming. An important contribution of this paper is that ReDAC supports the dynamic reconfiguration of distributed multi-threaded and re-entrant applications with cyclic call graphs.

1 Introduction

In order to operate software with high reliability and availability, applications must support reconfiguration during runtime in order to avoid downtimes caused by rebooting the system. Dynamic reconfiguration allows applying hot-fixes during runtime in order to remove bugs potentially causing security vulnerabilities. We developed a new algorithm: ReDAC (Reconfiguration of Distributed Application with Cyclic dependencies), allowing for dynamic reconfiguration of component software during runtime.

We use ReDAC to implement security and fault-tolerance in the Distributed Control Lab (DCL), a remote laboratory environment. The DCL enables physical experiments to be programmed via a Web-interface, allowing sharing expensive equipment among students. Using our algorithm we can observe the behavior of students' control algorithms, and replace faulty control components with a verified safety controller. Also components containing malicious logic can be replaced without stopping the whole control application.

ReDAC can be implemented on top the modern component-platforms Java and .NET. Within the component-oriented programming paradigm, applications are composed out of a number of components, possibly produced by different vendors independently [15]. In order to be able to apply dynamic updates on the component level, a way must be found to change the components and their configuration during runtime. Since the term component is commonly defined for the deployment phase only [15], we introduce the concept of components at runtime, which we call **capsules**. A capsule logically groups a number of objects during runtime, which stem from a common set of components. The capsule abstraction allows for the dynamic reconfiguration on the level of components without stopping/involving whole applications. Changes to a component-based application during runtime include the addition of new capsules, changing capsule parameters, changing connections between capsules or activating updated versions of a capsule (dynamic update).

When changing application configurations during runtime, the consistency of application data must be ensured. Moazami-Goudarzi [8] defines consistency by the correctness of data. A component-based application is correct despite a reconfiguration if the following three conditions hold:

- The structural integrity is remained.
- Capsules are in mutually consistent states.
- Application invariants hold.

The purpose of our research was to find an algorithm which ensures consistency while applying configuration changes at runtime. We developed an algorithm which can be implemented on modern component-platforms, which

typically include a virtual execution platform. The central idea of the algorithm is to execute reconfiguration actions while there is no active method call on the involved capsules, because in this situation there is no state on the stack (which cannot be easily access on virtual execution platforms) and no mutual inconsistent capsule state due to on-going method calls. This means that the algorithm must synchronize applications threads executing functional methods and a thread executing reconfiguration commands. We already developed an algorithm and a corresponding programming model which supported acyclic interactions among capsules [9].

A second approach [10] used readers and writers locks (rw-locks) [6, S. 80ff.] for synchronization. For each functional method call, a read-lock has been acquired and for each reconfiguration request a write lock. Using an rw-lock implementation with a recursive locking feature, this approach supports multi-threaded applications with cyclic dependencies. Due to the nature of rw-locks this approach only works for local applications - in the distributed case, we developed a new algorithm introduced within this paper.

The remainder of the paper is structured as follows. Within the following section we introduce the state of the art in dynamic reconfiguration. Then we introduce our new algorithm and in a separate chapter a few implementation details. Finally, we present a performance evaluation of our algorithm and conclude the paper.

2 Related Work

The area of dynamic reconfiguration has been heavily influenced by Kramer and Magee with the Conic platform [5]. Applications in Conic consist of a number of processing entities called nodes and connections among them. A node initiates and services transactions. Kramer and Magee introduce the concept of a transaction which logically combines a number of interactions between two nodes. A transaction t_1 **depends** on a second transaction t_2 if t_1 can only complete if t_2 can be executed successfully. In their approach, a reconfigurable state is reached by making nodes passive. Having independent transactions, a node is passive if it is not involved in a transaction initiated by itself and does not initiate new transactions. If all nodes that potentially initiate a transaction on a node are passive, this node is said to be quiescent - a synonym for a reconfigurable state.

In the case of dependent transactions the algorithm was extended. In the generalized passive state, nodes are not involved in transactions they initiated and do not initialize new transactions for themselves, but they handle dependent transactions from other nodes - potentially initiating new transactions. The reconfigurable state is reached if all nodes are in the generalized passive state that potentially initiate transactions on nodes involved in the reconfiguration. A disadvantage of Kramer and Magee's approach is that the whole processing of nodes is stopped by a reconfiguration. In addition it is not possible in the Java- and .NET component platform to differentiate individual sources of method calls.

Moazami-Goudarzi [8] proposed an approach based on the assumption that a node is in an inconsistent state only during a transaction. A passive state of nodes is reached by sending a block command to each node involved in a reconfiguration, causing the nodes not to handle transactions. In order to support dependent transactions nodes have to be potentially unblocked during the blocking phase. In some cases, nodes have to be blocked and unblocked several times during one reconfiguration. Moazami-Goudarzi assumes that there is only one transaction active in a node at a time. The approach does not support multi-threaded applications.

Wermelinger [7] extended Kramer/Magee's algorithm allowing for reduced blackout times. He introduces ports as communication endpoints between nodes. Transactions are initiated via initiator-ports and executed by recipient-ports. Transaction dependencies are modelled on the base of port dependencies. In contrast to Kramer/Magee's algorithm only connections between nodes are blocked. A connection is blocked by preventing new transactions to start and wait for ongoing transactions over that connection to complete. Dependent transactions must be blocked orderly to avoid deadlocks. Dependencies between transactions are analysed on the basis of port-pairs. In comparison to Kramer and Magee who rely on the static structure of connections to decide what nodes must be blocked, Wermelinger uses individual transactions during runtime. This minimizes the disruption of applications but requires the connections among nodes to be acyclic.

Bidan et al. [2] describe a dynamic reconfiguration service for CORBA. Application consistency during a reconfiguration is preserved by maintaining the integrity of remote procedure calls (RPC). This means that no initiated RPC must be unprocessed. Also this approach relies on blocking of connections. Within this approach the blocking of dependent transactions must be ordered - this also means that the connection graph must be acyclic.

Almeida et al. [1] introduce the concept of selective queuing, which is a mechanism to decide for each message exchange between two nodes if a message must be queued or processed. By forwarding implicit parameters (node ID's) with each message it is possible to decide if a message stems from a node included in the set of nodes affected by a dynamic reconfiguration. Although our approach uses related techniques, the complexity of the overhead caused by our algorithm (O(1)) related to the call depth is less since we are not relying on additional parameters passed with each interaction queuing up for each method on the call stack (O(n)). In addition Almeida et al. require all interactions to be executed via the CORBA middleware, while our approach also supports efficient in-process method invocations.

There are other approaches for dynamic reconfiguration which use different execution models. Our solution targets on dynamically reconfigure component-based applications. OSA+ [12] relies on a service-based execution model allowing for dynamic reconfiguration of service implementations. In OSA+ reconfigurations can be executed in bounded time. Also, Port-based Objects (PBO) [14] provide an execution model allowing dynamic reconfiguration in bounded time. In contrast, our approach can be applied to standard component platforms such as Java and .NET without a manipulation of the used virtual execution environment.

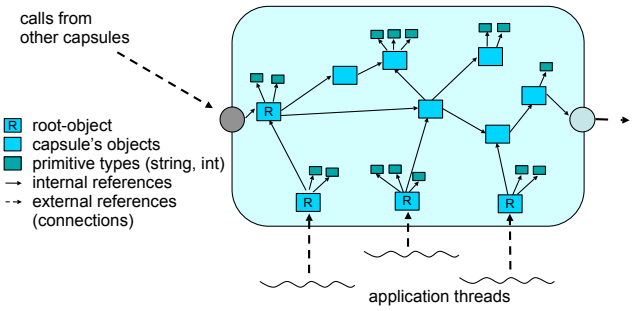

Figure 1. Capsules and Root-Objects

3 Application model

Making dynamic reconfiguration applicable to modern component platforms, we formulate an appropriate application model first. The component technology proposes applications to be composed out of a number of components. In [15], the term component denotes binary units of deployment that contain prototypes for the runtime entities of an application. We want to make the abstraction 'component' available at runtime. Having this option available, it will be possible to apply dynamic updates at the level of components during runtime.

To avoid confusions, we call component instances at runtime capsules. Our capsule concept can be compared to this introduced in Real-time UML [13]. Capsules are runtime entities, which are composed of a set of objects. We define several properties for these objects to form a capsule. At first we introduce the reachability relation for objects.

Definition 3.1 (Object reachability) *An object x_2 is said to be reachable from an object x_1 ($x_1 \rightsquigarrow x_2$) if:*

- *x_1 directly references x_2*
- *$\exists x_3 : x_1 \rightsquigarrow x_3, x_3 \rightsquigarrow x_2$.*

An object x_1 references another object x_2 if there is an attribue of x_1, which has a reference type and has the target x_2.

Using the reachability relation we are able to define a capsule as the set of objects, which can be reached from a set of root-objects, specified by an application developer. A capsule can be seen as an object graph, starting from the set of root objects. Threads invoke methods at objects of the object graph. During the execution of a method other methods could be invoked. A method of a non root-object can only be invoked by a thread having an open method on a root-object of the capsule. This means a thread enters a capsule always via a root-object. Root-objects can be used to guard the execution of application threads while dynamically updating a capsule.

Capsules provide a way to group a set of objects together which have been defined in the same binary component. A capsule is created by building an instance of a type defined in a component. The initial object is called **main-object**. Starting from the main-object, a capsule can contain other objects which are all reachable from the main-object, or potentially other root-objects, which cannot be reached by the main-object anymore. Capsules form a logical boundary for a set of objects which we used as the unit of change during runtime.

Capsules communicate via remote object invocations (here also called method calls) over a connection. A connection is always directed and interconnects two capsules. A connection is represented by an attribute of the main-object of a capsule. This means, that a connection is a reference between two capsules. The target of a connection is always a main-object of another capsule. Threads execute instructions of the objects' methods and can be seen as ingoing connections of a capsule. A thread can enter a capsule directly via a method call by a thread or via a remote object invocation from another capsule. A thread always enters a capsule via a root-object. Figure 1 gives an overview of a capsule and connections.

Within this paper we consider a transaction between two capsules to consist of single method calls only. A method invocation between two capsules can cause a dependent invocation on a third capsule. Having dependent method invocations, the first invocation returns only when the dependent invocation returned before. Dependent invocations can be re-entrant. This means, that a dependent invocation can enter a capsule again, although an invocation opened by the same thread in this capsule has not yet returned. Re-entrant method invocations are also called to be cyclic dependencies, since the call graph in this case contains a cycle. There can be multiple threads working on the objects of a capsule.

We define the **state of a capsule** as the data of all objects (attributes) and all object identities if there is no on-going method call on the capsule. Having on-going method calls, the state could also reside on the stack, in registers, or on the network, which cannot (easily) be accessed in the used virtual execution platforms. Access to capsule state is required to perform dynamic updates of capsules.

4 Problem specification

Using our application model, reconfiguration actions can include adding or removing capsules, or performing a dynamic update. Reminding the consistency requirements given before, it is not possible to reconfigure an application at any point in time. The structural integrity can be corrupted if a capsule is removed while another capsule tries to call a method at this point in time on that capsule. A mutually consistent state between capsules can be corrupted if there is an on-going method call during a reconfiguration. This is because capsule state is potentially not valid during a method call because there is data passed between the capsules. Since an update operation of a capsule must be able to work on valid state of a capsule and also because of the other consistency requirements, an application can only be reconfigured if there is no on-going method call on any involved capsule. We call the set of capsules, involved in a reconfiguration, the **block-set**. The block-set depends on the reconfiguration actions that must be executed. Although we could apply single reconfiguration actions to one capsule at a time, we allow the application of multiple reconfiguration actions at a time, because this can speed up the whole reconfiguration process.

The reconfiguration algorithm must ensure that there is no on-going method call on any capsule of the block-set - a reconfigurable state of all these capsules must be established. For reaching a reconfigurable state we rely on blocking application threads. Using aspect-oriented programming (AOP) [4] we are able to integrate new logic to root-objects of all capsules, transparently for the developer. Within the aspect logic, the blocking of connections can be implemented. Threads are directly blocked via root-objects. The usage of AOP is not a focus of this paper and was described in previous work [9].

In order to execute a reconfiguration, all activities in the capsules of the block-set must be blocked. Existing solutions rely on blocking whole connections. This means they block all threads invoking methods via an inter-capsule reference. In the case of acyclic connection graph it is possible to order the blocking of connections to reach a reconfigurable state without deadlocks [7]. For cyclic connection graphs the blocking of connections is not sufficient any more, because blocking of connectors can lead to deadlocks. One example of a potential deadlock is illustrated in Figure 2.

The figure shows three cyclically connected capsules. Two threads (ID1 and ID2) are working directly on the capsules K_1 and K_3. In Figure 2, consider capsule K_1 and K_2 to be updated. When the reconfiguration is triggered, thread ID1 has on-going method calls in all capsules, thread ID2 only in K_3. If the connection between K_3 and K_1 is blocked, one can see, that there is a potential deadlock. The execution of thread ID1 (indicated by the striped line) would be blocked before entering K_3. But this thread still

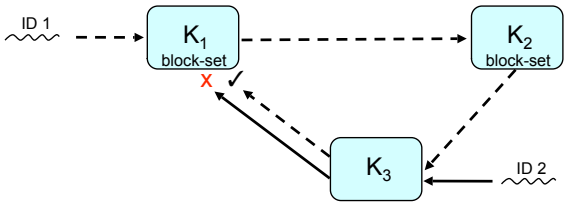

Figure 2. Cyclic Connections

has an on-going method calls on K_1 and K_2. Thread ID2 also tries to call a new method via the connection between K_3 and K_1. Blocking thread ID2 is no problem since there are no on-going method calls in K_1 or K_2. This example demonstrates that a reconfiguration algorithms supporting cyclic structures must selectively permit the initiation of new method calls by some threads still having on-going method calls on capsules of the block-set, while others can be blocked directly.

5 The Algorithm

Our new algorithm ReDAC includes a mechanism providing information of each thread about its on-going method calls and the involved capsules. While developing the algorithm some constraints have been given by some usage scenarios. The usage of component platforms with virtual execution environments puts some restrictions on available techniques. A known technique for solving the problem of identifying on-going method calls are stack walks, which provide a way for analyzing the execution of a thread. On each method call, a stack frame is created which can be used for identifying on-going method calls. Since it is not (at least not efficiently) possible to access the stack in virtual execution environments found in Java and .NET and it is especially complicated for distributed applications we use a different technique.

In order to identify on-going method calls, we increment a counter for each thread when entering a capsule and decrement it when leaving. During blocking, this counter can be used to find out if a thread still has on-going method calls on a capsule of the block-set and to allow new method calls to be initiated for this thread.

In order to implement the algorithm, a system-wide unique identifier for threads is required, which is commonly known as logical thread-ID. Logical threads-IDs are available in existing middleware technologies such as CORBA and DCOM. We assume the existence of logical thread-IDs within this paper. They can be easily added to Java or .NET by transferring the unique thread-IDs as implicit parameters with remote object invocations and injecting them into the tread-local-storage (TLS) of the target host. System-wide unique IDs can be generated by using a unique host ID and an ID of a local thread on the host creating the thread.

The required counter variables can be efficiently managed in hash-tables using logical thread-IDs as key and the counter value as entry in the table.

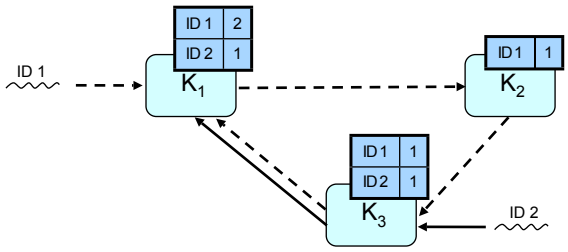

Figure 3. Algorithm: Data Structures

Figure 3 illustrates an example for the created data structures. For each thread, the current call depth is saved in a variable in each capsule indicating the number of on-going method calls. When blocking a thread, this data can be used to detect if the thread is active on a capsule of the block-set. An advantage of the proposed algorithm is that capsules can also - in the presence of dependent transactions - be blocked in parallel. This means that the blocking can be initiated on all involved capsules at the same time. When the last synchronously executed block request returns, the application is in a reconfigurable state. In the following, the algorithm will be described in more detail.

At first a helper function *ActiveThreadOnBlockSet* is required. This function indicates whether there is any thread with an on-going method call in any capsule of the block-set. The function checks for each capsule of the block-set if there is a positive counter value for any thread in the hash-tables of the capsules of the block-set. The hash-table (`ThreadList`) exists once per capsule. The function is shown as pseudo-code in Algorithm 1.

Algorithm 1 Function ActiveThreadOnBlockSet

```
1: for all c ∈ BlockSet do
2:   if ∃c.ThreadList[x] > 0, x ∈ ThreadIDs then
3:     RETURN true
4:   end if
5: end for
6: RETURN false
```

For a dynamic reconfiguration with our algorithm, two mechanisms have to be inserted into the application logic. On the one hand the described hash-tables must be updated during normal operation and on the other hand there must be appropriate synchronization constructs during the blocking phase. The following Algorithm 2 will be executed before every method call on a method of a root-object. Using the variable *blockMode*, the blocking phase can be started. The value *false* indicates normal operation, while *true* indicates an active block mode. The block mode can be initialized by setting the value of the variable *blockMode* simultaneously in all capsules of the block-set. This can be implemented by a central synchronization manager running in the context of a configuration manager, which is required for handling reconfiguration commands.

If there is no active block request (`blockMode=false`), the corresponding counter in *ThreadList* will be increased. The counter is saved in the hash-table (*ThreadList*), which exists once per capsule, using the ID of the thread invoking the method as key. In the block phase it is first checked if the reconfigurable state is already reached (line 2). In that case the event-object *quiescenceReached* is signalled (line 2,3), in order to inform the thread executing reconfiguration actions that the quiescent state has been reached. In every case a thread will be blocked if it has no on-going method calls on any capsule on the block-set (line 6). This part of the algorithm contains its central mechanism, which is the selective blocking of threads without on-going method calls on capsules of the block set and letting pass all threads with on-going method calls in order to complete the on-going method calls. When the blocking phase is finished all blocked threads are signalled via the event-object *blockEnd*.

Algorithm 2 Before every method call of root objects

```
Require: id is the ID of the thread executing the method
1: if blockMode then
2:   if not ActiveThreadOnBlockSet() then
3:     SIGNAL(quiescenceReached)
4:   end if
5:   if ∄c ∈ block − set : c.ThreadList[id] > 0 then
6:     WAIT(blockEnd)
7:   end if
8: end if
9: ThreadList[id] ← ThreadList[id] + 1
```

The following part of the algorithm (Algorithm 3) is executed when a method call is finished. During normal operation only the hash-table value will be decreased. During the blocking phase the quiescent state will be signalled via the event-object *quiescenceReached* if exactly this decrement operation caused the last positive value in a capsule of the block-set to be zero. This means that the last application thread with an on-going method call finally informs the configuration manager that the reconfiguration activities can be carried out.

Algorithm 3 After every method call of root objects

```
1: ThreadList[id] ← ThreadList[id] − 1
2: if blockMode then
3:   if not ActiveThreadOnBlockSet() then
4:     SIGNAL(quiescenceReached)
5:   end if
6: end if
```

Last but not least the final part of the algorithm (Algo-

rithm 4) shows the implementation of a block request issued by the configuration manager. At first the event-objects *blockEnd* and *quiescenceReached* are reset (line 1,2). Afterwards, the variable *blockEnd* will be set to *true* in all involved capsules to indicate the start of the blocking phase. The next step is a check if the quiescent state has already been reached. Otherwise the thread executing the block request waits for the on-going method call to complete (line 7,8). When the event-object is signalled, the reconfiguration thread will be released and the dynamic reconfiguration actions can be applied. Finally, the processing of the application threads is continued by signaling *blockEnd*, after the reconfiguration has finished.

Algorithm 4 Reconfiguration Request

```
1:  RESET(quiescenceReached)
2:  RESET(blockEnd)
3:  for all c ∈ BlockSet do
4:      c.BlockMode ← true
5:  end for
6:  if ActiveThreadOnBlockSet() then
7:      WAIT(quiescenceReached)
8:  end if
9:  Reconfiguration Actions
10: for all c ∈ BlockSet do
11:     c.BlockMode ← false
12: end for
13: SIGNAL(blockEnd)
```

After the description of the ReDAC, we'll in short discuss a small extension. Existing reconfiguration approaches do not yet consider synchronisation between application threads (in most cases because no multi-threading is supported). Almeida et al. [1] require that there are no dependent non-re-entrant threads in an application. In case of common synchronization objects among application threads our algorithm (and the other algorithms) could run into a deadlock. One example of such a deadlock is shown in Figure 4.

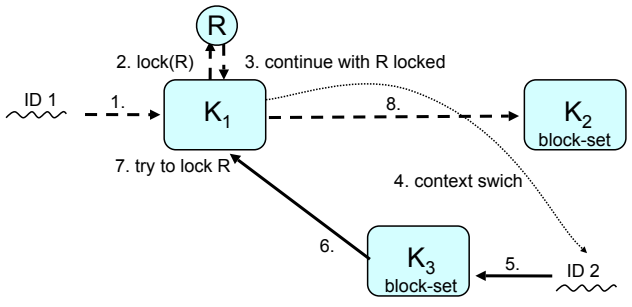

Figure 4. Reconfiguration and Resources

Within the figure, 3 capsules can be seen. The capsules K_2 and K_3 form the block-set. A first thread (ID 1) calls a method at K_1, locks the resource R and starts to initiate a method call at K_2 still having locked R. While the call has not yet reached K_2 a context switch activates a second thread, invoking a method at K_1 via K_3 and tries to lock R as well, but is blocked because thread ID1 owns R. If in that moment a reconfiguration is triggered, there is a potential deadlock. The call (8.) initiated by thread ID1 would be blocked in K_2 because this thread has no on-going method calls on capsules of the block-set. The resource R would not be released by thread ID1, forever blocking thread ID2, that still has an on-going method in K_3.

Fortunately this problem can be solved with our algorithm with a little extension only, not influencing runtime properties of the algorithm. The block-set must be extended with all capsules which are on a path between capsules of the block-set and do not belong to the block-set themselves. This are by far not all capsules of the applications. Our algorithm is the first, we know of, that solves the reconfiguration problem also in the existence of common ressources between application threads.

We used a Petri-net model to prove some important properties of the algorithm, such as the absence of deadlocks, progress and mutual exclusion of application and reconfiguration activities. We modelled a special application configuration together with the reconfiguration algorithm as Petri-net. Using model checking, the properties of the algorithm have been evaluated.

Figure 5 shows our Petri-net model representing an application with one capsule. For the model checking, we used the tool TimeNET [3] developed at the *Technische Universität Berlin*. TimeNET supports a quantitative analysis of stochastic Petri-nets and can be used for analysing invariants. The top of Figure 5 shows the only thread of the modelled application recursively looping through the places *Thread1Start*, *CallMethod*, *InMethod*, *RecursionPool* and *AftAdvice*. The place *InMethod* symbolizes the execution of a functional method of the application. The recursion depth is configured by the number of initial tokens in place *RecursionPool*. The other parts of the Petri-net represent the reconfiguration logic. The Petri-net has been created by systematically formalizing the introduced algorithm. The place *Reconfiguration* symbolizes the dynamic reconfiguration of the modeled application configuration. For the mutual exclusion of application activities and reconfiguration logic, the following invariant must hold:

$$P\{\#InMethod + \#Reconfiguration > 1\} = 0 \quad (1)$$

In TimeNET syntax this means, that the probability of more than one token in the places *InMethod* and *Reconfiguration* is zero. The invariant proves that there is never a marking in both places at a time. Using the TimeNET feature "'Stationary Analysis"' the property mutual exclusion has been proven by the invariant.

In addition we proved the properties progress and ab-

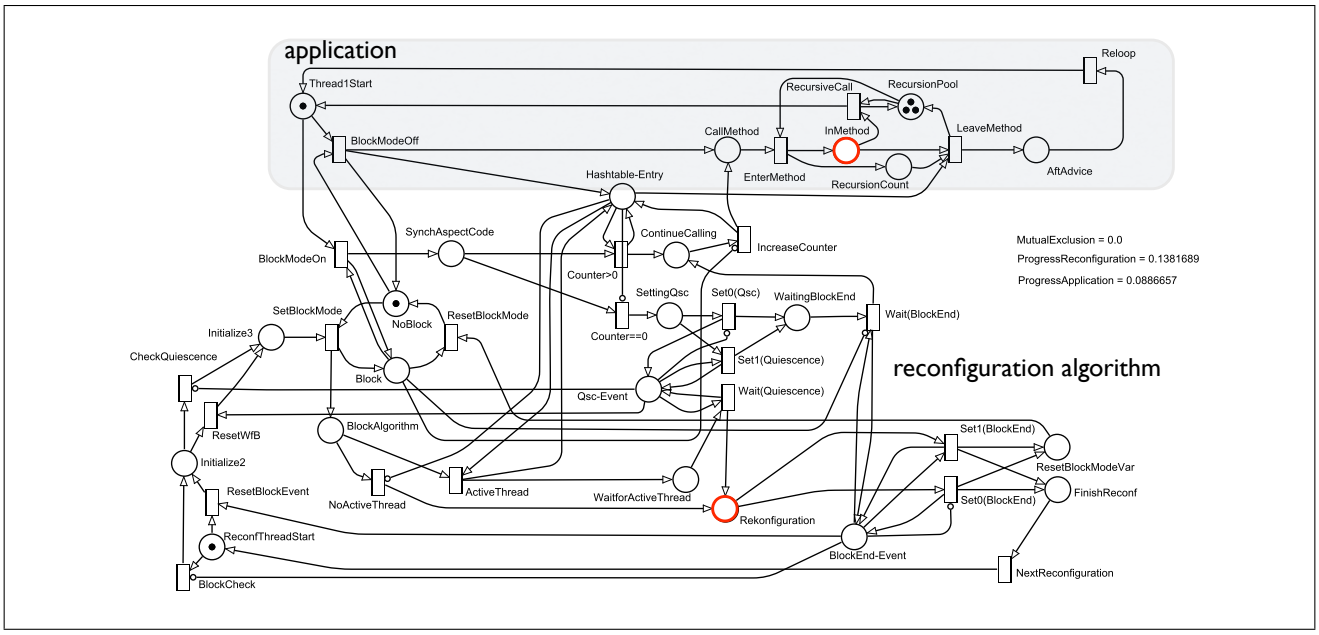

Figure 5. Petri-net of an application with one capsule including reconfiguration logic

sence of deadlocks with the following invariants:

$$E\{\#InMethod\} > 0 \quad (2)$$

$$E\{\#Reconfiguration\} > 0 \quad (3)$$

With the probability of a token in the place *Reconfiguration* distinct from zero progress has been proven because in that case there will always be a token in this place from time to time. We also consider more complex configurations. Due to space limitations, those will not be explained in this paper.

6 Implementation

We have implemented the described algorithm within our Adapt.Net framework [9] using the Microsoft .NET Framework. We developed tools for transparently integrating the described reconfiguration logic into functional components using aspect-oriented programming. Our tools add instructions to all methods of root-objects of a capsule in addition to a configuration-specific interface used by a configuration manager for executing reconfiguration commands. Due to space limitations we only describe a short excerpt of our implementation. Figure 6 shows the aspect logic executed for each root-object method of a capsule, if the thread already entered the capsule at least one time before (otherwise additional logic is executed to add a new entry into the used hash-table, which is not shown here).

The upper part of the logic (lines 4-19) is only executed in the case of an active block request. Line 20 shows the increment of the counter and afterwards in line 25 the functional method is invoked (not shown). After the invocation of the functional method the counter is decremented. In line 21 it is checked if between the test of the `blockMode`-Variable and the increment of the counter a block request has arrived. In this case a jump is made back to the block logic, because if the current thread would be preempted just after the check of the `blockMode`-Variable there would be a thread without an increased counter - potentially invoking a method during a reconfiguration. This approach for realizing thread-safety can be found in many places within the implementation of the .NET libraries.

In the case of a block request the variable `blockMode` is set to `true`. In this case the logic between line 4 and 19 is executed. The basic structure has already been described before in the form of pseudo-code. An important point to mention is that with an active block request multiple threads could signal the accomplishment of the reconfigurable state (line 12). If a second thread is preempted in line 6, while a first thread just signalled `quiescenseReached`, the second thread could trigger a second reconfiguration although a third thread still has on-going method calls, because the application could have already been restarted after the first reconfiguration. For this reason a reconfiguration counter (`reconfNr`) is introduced, realizing that a thread can never trigger a reconfiguration while not all capsules are in a reconfigurable state.

The implementation of the reconfiguration actions itself has been described elsewhere [9]. During the dynamic update of a capsule, state must be transfered from an old to a new version. We implemented this state transfer by a recursive traversal of the object graphs of a capsule, details can be found in [10]. Within this paper we focused on reaching a reconfigurable state only.

```
1  reprocess:
2    bool initialBlockMode = blockMode;
3
4    if (blockMode){
5      int initialReconfNr = reconfNr;
6      lock (globalLock){
7        if (NoActiveThreadOnBlockSet()){
8          if (!blockMode)
9            goto reprocess;
10         if (initialReconfNr != reconfNr)
11           goto reprocess;
12         quiescenseReached.Set();
13         reconfNr++;
14       }
15     }
16     if (!isThreadActiveOnBlockSet(curThread)){
17       blockEnd.WaitOne();
18     }
19   }
20   countRef.count++;
21   if (initialBlockMode != blockMode){
22     countRef.count--;
23       goto reprocess;
24   }
25   // Invoke target method
26   countRef.count--;
```

Figure 6. Implementation of block logic

7 Evaluation

We have measured the additional execution time of method calls caused by the integrated synchronization logic. In addition, we compared our algorithm to our previous implementation which used reader-writer-locks. To contrast we show measurements for normal method calls and `Reflection.Invoke`-calls, which are used by related approaches to perform a reconfiguration by forwarding method calls into a meta level [11]. The measurements have been performed on a 2,8 GHz Intel Xeon processor with one active core, 2 GB RAM and Windows XP SP2 as the operating system. We used our .NET implementation running on .NET Framework 2.0. All timings have been taken by using the CPU tick counter of the x86-CPU (`rdtsc`). All measurements have been taken for 1000 method invocations. The following table presents our measurements, which show mean values and standart deviation for 1000 separate measurements.

Type of method invocations	duration (1000 calls)
interface method call	5,96 $\pm$ 0,14 μs
`Reflection.Invoke`	5,05 $\pm$ 0,21 ms
.NET Remoting-call	346,92 $\pm$ 2,11 ms
ReDAC	151,68 $\pm$ 2,09 μs
RW-Lock synchronization	324,19 $\pm$ 4,28 μs

In average 1000 normal interface method calls take about 6 μs. In contrast 1000 calls with rw-lock-based synchronization lasts about 324 μs. 1000 method calls with synchronization realized with ReDAC takes about 152 μs. Although there is some overhead for normal method calls these few nano seconds can be neglected for normal applications since the method logic itself requires much more time. In addition not all method calls are interwoven with synchronization primitives, only the root-objects which are used for synchronization cause performance overheads. Method calls to root-objects sum up to a low percentage compared to all method calls. Details about this can be found in [10], which have been presented with our initial dynamic reconfiguration approach using rw-locks.

Compared to other approaches, which have been introduced in the related work section, our new algorithm has the advantage that there is very low overhead when performing in-process method calls. Since reconfiguration-specific data is saved within the capsules, no data must be passed along method calls [1], which typically use features of an underlying middleware-layer. To demonstrate the difference to this approach, we show method invocation times for the .NET Remoting middleware. In addition the approach proposed by [1] requires each interaction between components to be performed via the middleware-level, while our approach also supports efficient in-process calls.

The blackout-time, caused by our algorithm, has been evaluated using a test environment, with randomly generated configurations with a configurable number of capsules, threads, call depth, and call dependencies. This allows for the evaluation of complex application scenarios with complex dependencies and multiple threads. The blackout-time is also influenced by the duration of normal method calls (because the algorithm waits for on-going methods to complete). In order to determine the part of the blackout-time caused by our algorithm the used test application methods just executed a few arithmetical operations.

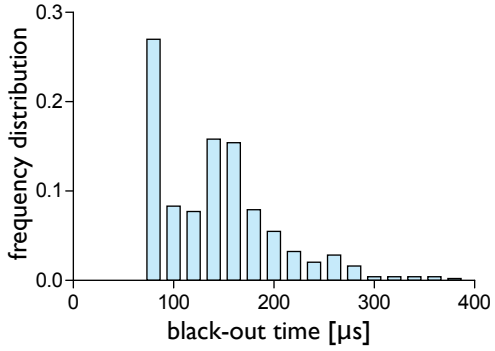

Figure 7. Blackout times

Figure 7 shows a histogram of the blackout-time for 500 reconfigurations. The test configuration contained 20 capsules, 10 threads and a maximum call depth of 20. Within the histogram one can see that the blackout-time ranges between 80 and 400 μs. Most measurements can be found below 300 μs.

Finally, one can say that all measurements confirm that our new algorithm is perfectly suitable for realizing dynamically reconfigurable applications. The overhead for functional method calls is very low, since our capsule definition requires only a small amount of method calls to be synchronized with the reconfiguration activities. Even considering todays faster hardware, ReDAC performs an order of magnitude faster than our previous approach presented in [9].

8 Conclusion

The contributions of the paper are: At first, the abstraction component has been made available at runtime (named capsule), allowing dynamic updates to be applied on the level of components, without stopping whole applications. Secondly, a new reconfiguration algorithm (ReDAC) has been proposed, supporting multi-threaded, re-entrant applications with cyclic call dependencies. Using the concept of logical thread-IDs, ReDAC works for distributed applications as well. In addition our algorithm supports the usage of common resources among application threads.

ReDAC uses counter variables to count on-going method calls for threads in order to detect a reconfigurable state of capsules. By selectively blocking threads in capsules that are involved in the dynamic reconfiguration, ReDAC can also actively establish a reconfigurable state. The required synchronization logic is inserted using aspect-oriented programming to all methods, directly called by threads or other capsules, allowing to guard the objects of a capsule during a reconfiguration.

We already applied the described algorithm while updating complex off-the-shelf applications. We have been able to dynamically update PaintDotNet, an open source application with more that 133.000 lines of code. While executing PaintDotNet no overhead caused by the reconfiguration logic could be noticed.

In a second case study we used our approach for dynamic reconfiguration to implement security and fault-tolerance in a remote laboratory environment. The Distributed Control Lab enables physical experiments to be programmed via a Web-interface, allowing sharing expensive equipment among students. Using our Adapt.Net framework we can observe environmental properties, while executing students' control algorithms and replace faulty controllers with a verified safety controller using dynamic reconfiguration.

9 Acknowledgments

Part of this work has been sponsored by the Phoenix Direct Funding Award 2007 Number 15899. We would like to thank Armin Zimmermann and Michael Knoke for their support with the TimeNET tool and Martin von Löwis and Alexander Schmidt for helpful discussions of the algorithm and reviewing this paper.

References

[1] J. P. A. Almeida, M. V. Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for corba. In *DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications*, page 197, Washington, DC, USA, 2001. IEEE Computer Society.

[2] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A Dynamic Reconfiguration Service for CORBA. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 35, Washington, DC, USA, 1998. IEEE Computer Society.

[3] R. German, C. Kelling, A. Zimmermann, and G. Hommel. Timenet: a toolkit for evaluating non-markovian stochastic petri nets. *Performance Evaluation*, 24(1-2):69–87, 1995.

[4] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. 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.

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

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

[7] 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.

[8] K. Moazami-Goudarzi. *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.

[9] A. Rasche and A. Polze. Configuration and Dynamic Reconfiguration of Component-based Applications with Microsoft .NET. In *Proceedings of ISORC'03*, pages 164–171, Hakodate, Japan, May 2003.

[10] A. Rasche and W. Schult. Dynamic Updates of Graphical Components in the .NET Framework. In *Proceedings of SAKS'07 Workshop*, pages 219 – 230. VDE Verlag, 2007.

[11] S. M. Sadjadi, P. K. McKinley, B. H. Cheng, and R. K. 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] E. Schneider, F. Picioroagă, and U. Brinkschulte. Dynamic reconfiguration through osa+, a real-time middleware. In *DSM '04: Proceedings of the 1st international doctoral symposium on Middleware*, pages 319–323, New York, NY, USA, 2004. ACM Press.

[13] B. Selic. Using uml for modeling complex real-time systems. In *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260, London, UK, 1998. Springer-Verlag.

[14] D. B. Stewart, R. A. Volpe, and P. Khosla. Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects. Technical Report CMU-RI-TR-93-11, Carnegie Mellon University, Pittsburgh, PA, July 1993.

[15] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999.