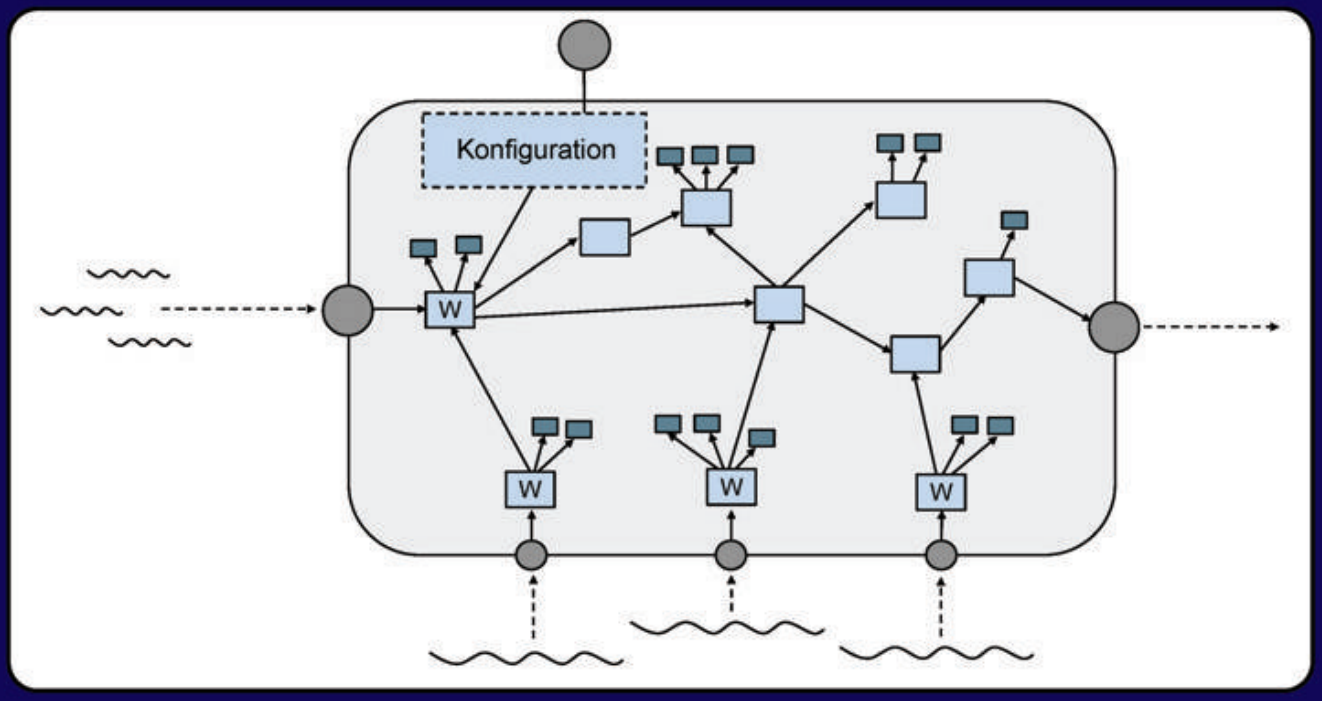
Ausführung und Entwicklung adaptiver komponenten-basierter Anwendungen

Dipl.-Inf. Andreas Rasche



Cuvillier Verlag Göttingen

Ausführung und Entwicklung adaptiver komponentenbasierter Anwendungen

Dipl.-Inf. Andreas Rasche

Cuvillier Verlag Göttingen

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.
1. Aufl. - Göttingen : Cuvillier, 2008
Zugl.: Potsdam, Univ., Diss., 2008

978-3-86727-698-6

Nonnenstieg 8, 37075 Göttingen
Telefon: 0551-54724-0
Telefax: 0551-54724-21
www.cuvillier.de

1. Auflage, 2008
Gedruckt auf säurefreiem Papier

978-3-86727-698-6

Zusammenfassung

Adaptive Anwendungen können sich auf Grund von Änderungen ihrer Umgebung anpassen, u.a. um von den Nutzern vorgegebene Anwendungseigenschaften zu gewährleisten, die durch die variierende Verfügbarkeit von Ressourcen unterliegender Infrastrukturen verletzt werden. Eine Lösungsstrategie für die Anpassung ist die Auswahl entsprechender Softwarekomponenten, die im Rahmen der Komponentenorientierung, einer Methode zur Konstruktion komplexer Software aus Einheiten unabhängiger Herstellung, Beschaffung und Deployment, in alternativen Varianten mit unterschiedlichen Eigenschaften vorliegen. Während die funktionalen Anforderungen einer Anwendung oft von mehreren gültigen Zusammenstellungen erfüllt werden können, variieren die resultierenden Anwendungseigenschaften in unterschiedlichen Umgebungssituationen. Die Ausführung der besten Alternative für eine Menge verfügbarer Ressourcen ist damit die Grundlage für die Gewährleistung gewünschter Anwendungseigenschaften. Bei Änderungen der Umgebung kann die Anpassung durch den Austausch mit einer besseren Alternative realisiert werden.

Die Abstraktion der Softwarekomponente ist in der Java- und .NET-Komponentenplattform zunächst für den Zeitraum des Deployment definiert. Um die Austauschbarkeit von Komponenten auch während der Laufzeit zu ermöglichen, führt die vorliegende Arbeit das Konzept der Kapseln ein, die eine Menge von Objekten gleicher Deployment-Einheiten zusammenfassen. Während der Laufzeit wird eine komponentenbasierte Anwendung als gerichteter Graph verbundener Kapseln verstanden, die von vernetzten Rechnern ausgeführt werden. Veränderungen der Konfiguration der Kapseln einer Anwendung werden als dynamische Rekonfiguration bezeichnet, die für die Nutzung der Komponentenorientierung als Basis der Adaption in aktuelle Komponentenplattformen integriert werden muss und bildet den Schwerpunkt dieser Arbeit.

Bei der dynamischen Rekonfiguration müssen Änderungsoperationen und Anwendungsaktivitäten synchronisiert werden. Hierfür stellt diese Arbeit einen neuen Algorithmus vor, der mit Hinblick auf die Komplexität der Zusatzkosten existierende Ansätze verbessert und dabei Anwendungen mit multiplen Threads und zyklischen Verbindungsstrukturen unterstützt. Für einen Spezialfall der dynamischen Rekonfiguration, der dynamischen Aktualisierung, die das Einspielen neuer Versionen von Softwarekomponenten während der Laufzeit ermöglicht, beschreibt die vorliegende Arbeit ein neues Verfahren für den notwendigen Zustandstransfer, das durch den Einsatz von identifizierten Basistechnologien ohne eine Manipulation der virtuellen Laufzeitumgebung auskommt. Das entwickelte Verfahren kann auch für das Einspielen fehlerbereinigter Softwareversionen ohne Neustart der Anwendung verwendet werden und ist dabei leicht in existierende Anwendungen integrierbar, was diese Arbeit im Rahmen komplexer Fallstudien nachweist.

Ein weiterer wichtiger Beitrag der Arbeit ist der Einsatz der aspektorientierten Programmierung für die wiederverwendbare Generierung komplexer (re-)konfigurationsspezifischer Logik, welche die Entwicklung adaptiver komponentenbasierter Anwendungen vereinfacht. Die entwickelten Techniken wurden unter anderem im Distributed Control Lab, einer Infrastruktur zur entfernten Ausführung von Steuerungsexperimenten über das Internet, für den Schutz vor bösartiger Software verwendet. Dabei werden Steuerungen während ihrer Laufzeit überwacht und im Fehlerfall die Steuerungsanwendung dynamisch rekonfiguriert.

Abstract

Adaptive applications adapt to changing context conditions, among other reasons to cope with varying resources of underlying infrastructures, which violate application-level quality properties desired by the user. One solution to implement the adaptation is the selection of appropriate software components. The component-oriented programming paradigm introduces software components as units of independent construction, acquisition and deployment, which may exist in several variants with different properties. The functional requirements of an application can be achieved by various valid compositions, which may have different application-level quality properties in environmental situations given. The selection and activation of the best alternative for a set of available resources provide the basis to keep the application-level quality properties in a required range.

In the Java and .NET component platforms, the abstraction component is defined at deployment time only. In order to extend the exchangeability given by components to runtime, this thesis introduces capsules which logically combine a number of objects created from common deployment units. During runtime, applications are modeled as graphs of interconnected capsules which are executed by a set of networked computers. Adaptive component-based applications change the configuration of capsules during runtime. In order to make the concepts introduced by the component-oriented programming paradigm available for the development of adaptive applications, this thesis focuses on the integration of dynamic reconfiguration into modern component platforms.

During a dynamic reconfiguration application activities and change operations have to be synchronized. For this reason this thesis introduces a new reconfiguration algorithm, which improves existing approaches in terms of the complexity of runtime overhead, also supporting applications with multiple threads and cyclic connections. Dynamic updates are a special case of dynamic reconfiguration and allow the installation of new component versions during runtime. This thesis describes a new approach for the required state transfer, which relies on base technologies such as reflection to implement dynamic updates without the manipulation of the virtual machine. The new approach can also be used to install bug-fixed versions without a restart of applications. In addition the introduced approach can easily be used with existing software. This has been proven in various complex case studies.

Finally, an important contribution of this thesis is the usage of aspect-oriented programming for the reusable generation of complex (re-)configuration-specific logic, which simplifies the development of adaptive applications. Among other, the developed techniques have been used in the Distributed Control Lab, an e-learning infrastructure for remote experiment execution, in order to prevent damage from expensive experiment hardware by malicious code potentially uploaded from the Internet. In the lab, control applications are observed during runtime and in case of abnormal behavior user control algorithms are replaced with a dynamic reconfiguration.

Danksagung

Die vorliegende Dissertation wäre ohne die Unterstützung vieler Menschen nur schwer zustande gekommen, denen ich an dieser Stelle danken möchte.
In erster Linie bedanke ich mich bei meinem Mentor Prof. Dr. Andreas Polze für die Ermöglichung dieser Dissertation. Seine Ratschläge und Führung sowie die inspirierende Atomsphäre in seiner Arbeitsgruppe waren die wichtigste Vorraussetzung für das Gelingen dieser Arbeit.
Besonderer Dank gilt meinen Kollegen aus dem Fachgebiet "Betriebssysteme und Middleware", die wesentlich zum Gelingen dieser Arbeit beigetragen haben. Für unsere gemeinsamen Diskussionen, die zahllosen lehrreichen Stunden und die enge Zusammenarbeit und Unterstützung danke ich Dr. Martin von Löwis, Wolfgang Schult, Peter Tröger und Bernhard Rabe. Michael Dirska schulde ich großen Dank für seinen Einsatz und sein technisches und physikalisches Know-How beim Aufbau der Experimente im Distributed Control Lab.
Prof. Lars Lundberg möchte ich für die Möglichkeit eines Forschungsaufenthalt an der Blekinge Techniska Högskola danken. Er und viele Mitarbeiter haben für eine unvergessliche, produktive aber auch erlebnisreiche Zeit in Schweden gesorgt.
Dank schulde ich auch meiner Familie und meinen lieben Freunden, die mir immer mit Rat und Tat zur Seite standen. Besonders danke ich Christa für ihren grammatikalisch-orthographischen Durchblick. Meiner Frau Claudia danke für ihren Rückhalt und ihre Ermutigung und dafür, dass sie immer für mich da ist.

Inhaltsverzeichnis

1 Einleitung

Adaptive Anwendungen können sich auf Grund von Änderungen ihrer Umgebung dynamisch rekonfigurieren, u.a. um von den Nutzern vorgegebene Anwendungseigenschaften (*quality of service*) zu gewährleisten, die durch die variierende Verfügbarkeit von Ressourcen unterliegender Infrastrukturen verletzt werden [Kon 2000]. Das Phänomen der variierenden Verfügbarkeit von Ressourcen wird nach [Dini u. a. 2004] als *partial ressources* bezeichnet und kann z.B. durch die Verwendung mobiler Kommunikationsnetze verursacht werden, die durch physikalische Gegebenheiten wie die Entfernung eines Nutzers zu einer Basisstation oder die Anzahl von Nutzern in einer Funkzelle variierende Übertragungsbandbreiten oder Fehlerraten besitzen. Auch die Nutzung des Internets über wechselnde Kommunikationsverbindungen wie drahtgebundene und drahtlose Netzwerke verursachen Änderungen in den verfügbaren Ressourcen, auf die adaptive Anwendungen entsprechend reagieren müssen. Eine dynamische Rekonfiguration kann dabei einfache Parameteranpassungen, die Aktualisierung von Anwendungslogik während der Laufzeit oder Änderungen der Softwarearchitektur beinhalten.
Neben der Verfügbarkeit von Ressourcen können Änderungen der Umgebung auch geänderte Anforderungen durch einen Nutzer oder das Bekanntwerden von Sicherheitslücken umfassen. In diesem Fall ist die dynamische Rekonfiguration eine wichtige Voraussetzung für die Erhöhung der Verfügbarkeit einer Anwendung durch die Möglichkeit des Einspielens aktualisierter Softwareversionen (*hot-fix*). Wird die Rekonfiguration während der Laufzeit durchgeführt, können Ausfallzeiten durch einen Neustart vermieden werden. Während das Einspielen von aktualisierten Softwareversionen eine Maßnahme zur Fehlervermeidung darstellt, kann die dynamische Rekonfiguration auch eingesetzt werden, um bei einem Ausfall von Teilen einer Anwendung, Fehler zu beheben [Laprie 1998].

1.1 Konfiguration verteilter komponentenbasierter Anwendungen

Verteilte Anwendungen werden auf einer Reihe von Rechnern ausgeführt, die über ein Netzwerk miteinander verbunden sind (verteiltes System). Für die Entwicklung von Software hat in den letzten Jahren der Ansatz der Objektorientierung eine weite Verbreitung gefunden. In der Objektorientierung wird die betrachtete Welt in Objekte mit ihren Eigenschaften und Operationen gekapselt. Die Struktur eines Objekts wird durch seine Attribute (Eigenschaften) beschrieben. Das Verhalten eines Objekts wird durch seine Methoden (Operationen) festgelegt. Eine **objektorientierte verteilte Anwendung** besteht aus einer Reihe von Objekten, die auf über ein Netzwerk verbundenen Rechnern existieren und über entfernte Methodenaufrufe (*Object Remote Procedure Call* (ORPC)) miteinander kommunizieren.
Eine wichtige Technologie zur Strukturierung komplexer Software ist die Verwendung von Softwarekomponenten. In Analogie zu Hardwarebausteinen werden Softwarekomponenten als austauschbare, unabhängig erworbene Elemente der Wiederverwendung

aufgefasst. Nach [Szyperski 1999, S. 36 ff.] ist eine Softwarekomponente eine ausführbare Einheit unabhängiger Herstellung, Beschaffung und Deployment, die durch Dritte in ein funktionierendes System komponiert werden kann. Unter **Deployment** wird in der Informatik die Auslieferung und Installation von Software in einem verteilten System verstanden [Object Management Group 2006, S. 1]. Eine Softwarekomponente besitzt wohl definierte Schnittstellen, über die auf ihre Funktionalität zugegriffen werden kann. Softwarekomponenten sind eine Hülle für zusammenhängende Software in binärer Form und enthalten Vorlagen für die Laufzeitstrukturen einer Anwendung. Eine **komponentenbasierte Anwendung** wird aus einer Reihe von Softwarekomponenten erzeugt, die im Falle der Objektorientierung Vorlagen für die Objekte einer Anwendung enthalten.

Die Komponentenorientierung ist dabei nicht nur technischer Natur, sondern hat auch einen wirtschaftlichen Hintergrund. Durch die Schaffung von durch Konkurrenz belebte Märkte können die Qualität der aus Komponenten zusammengesetzten Endprodukte gesteigert sowie die Entwicklungskosten reduziert werden [Szyperski 1999, S. 4 ff.]. Aus der Definition nach Szyperski folgt, dass eine Komponentenart (Menge von Komponenten gleicher Schnittstelle) von mehreren unabhängigen Herstellern erstellt werden kann und dabei mit unterschiedlichen Qualitätsmerkmalen bzw. Eigenschaften vorliegen kann. Ein Anwendungsentwickler kann bei der Komposition einer komponentenbasierten Anwendung aus einer Menge funktionsgleicher Komponenten wählen, um z.B. die Einhaltung eines bestimmten finanziellen Budgets einzuplanen, indem entsprechende Komponenten ausgewählt werden. Ein wichtiges Ziel der Komponentenorientierung ist es, einen Markt alternativer Komponenten zu erstellen.

Genau an dieser Stelle sind die Konzepte der Komponentenorientierung für die Entwicklung adaptiver Software von zentraler Bedeutung; Komponenten sind unabhängige Einheiten, die nach ihrer Komposition Systeme mit unterschiedlichen Eigenschaften ergeben, die sich in bestimmten Umgebungen besser eignen als andere Alternativen. Mit der Auswahl entsprechender Komponenten bei der Komposition einer Anwendung kann also die Anwendung an die Eigenschaften ihrer Umgebung angepasst werden, indem die besten Alternativen aus einer Reihe funktionsgleicher Komponenten ausgewählt werden.

Szyperski definiert den Begriff der Softwarekomponente als Hülle zusammenhängender Software in binärer Form. In den verbreiteten Java- und .NET-Komponentenplattformen werden Komponenten während der Anwendungslaufzeit durch Objekte zum Leben erweckt und die Kapselung, die zunächst logisch durch die Hülle der Komponenten geschaffen wurde, geht verloren. Für die Entwicklung adaptiver Software ist es nun wichtig, die funktionale Kapselung von Komponenten auf die Laufzeit von Anwendungen zu erweitern. Im Rahmen dieser Arbeit fasst eine Komponente während der Laufzeit die Menge aller Objekte zusammen, deren Klassen bzw. Objektprototypen in einer Deployment-Einheit „Softwarekomponente“ im Sinne der Szyperski-Definition enthalten sind. Die Austauschbarkeit von Softwarekomponenten kann so auf die Laufzeit einer Anwendung ausgeweitet werden. Um die Komponentenabstraktion während der Laufzeit zu differenzieren wird der Begriff der **Kapsel** eingeführt, die die beschriebene Zusammenfassung einer Objektmenge während der Laufzeit realisiert.

Komponentenbasierte, verteilte Anwendungen können als gerichtete Graphen mit Kapseln als Knoten und Verbindungen als Kanten betrachtet werden [Kramer und Magee 1990]. Kapseln werden auf Rechnern in einem verteilten System ausgeführt und interagieren in Form von Objekten über ihre Schnittstellen. Das Verhalten ein-

zelner Kapseln kann über Parameter konfiguriert werden. Die Verbindung zwischen den Objekten verschiedener Kapseln wird über Konnektoren realisiert, die als Interaktionsmedium dienen, über das entfernte Methodenaufrufe ausgeführt werden. Eine gültige Zusammenstellung parametrierter Kapseln, Konnektoren und einer Zuordnung der Kapseln auf die Rechner eines verteilten Systems wird als **Anwendungskonfiguration** bezeichnet. Der Konfigurationsbegriff wird in Kapitel 3 formal eingeführt.
Eine Anwendungskonfiguration geht aus einer Komposition von Komponenten aus einer Menge verfügbarer Alternativen hervor. Oft existieren verschiedene Möglichkeiten, aus einer Menge von Komponenten ein System zur Lösung einer bestimmten Aufgabe zu komponieren. Abbildung 1.1 zeigt die Auswahl von Komponenten und die Zusammenstellung zu gültigen Anwendungskonfigurationen. Die Komposition von Anwendungen wird nur am Rande dieser Arbeit diskutiert. Lösungsansätze wurden u.a. im Rahmen von Forschungsprojekten zum Thema „Ressourcenorientiertes Konfigurieren“ [Heinrich 1993] (einem Teilgebiet der Künstlichen Intelligenz) und seit kurzer Zeit im Zusammenhang mit der Komposition von Web-Diensten [Milanovic und Malek 2004] untersucht, sind aber auch Gegenstand aktueller Forschungsarbeiten [Richling 2006].

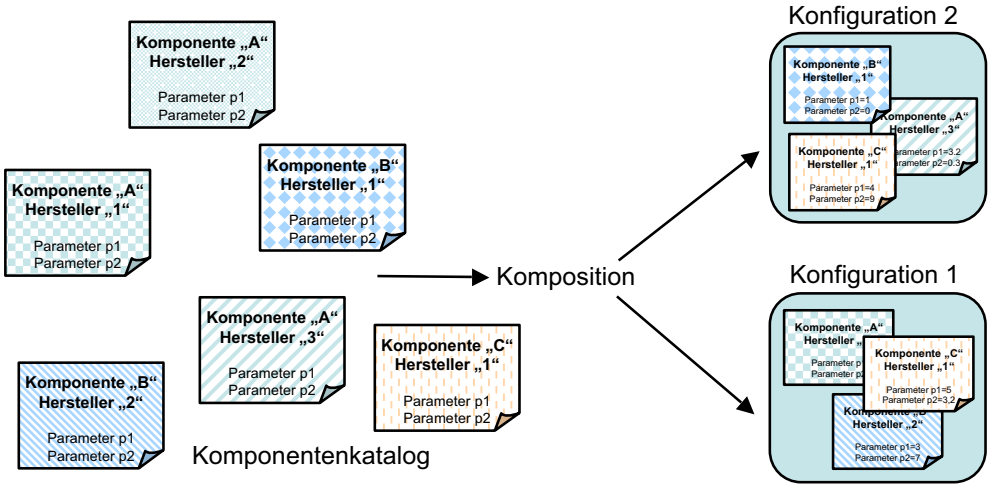


Abbildung 1.1: Komposition: Erstellung alternativer Konfigurationen

Neben den verfügbaren Ressourcen bestimmt die Anwendungskonfiguration einer verteilten komponentenbasierten Anwendung wesentlich ihre Eigenschaften, die adaptive Anwendungen in einem durch den Nutzer definierten Bereich halten sollen. Ändern sich die einer Anwendung zur Verfügung stehenden Ressourcen, können durch die **Auswahl einer alternativen Anwendungskonfiguration**, die für die gegebene Verfügbarkeit von Ressourcen optimiert ist, die geforderten Eigenschaften gewährleistet werden. Die Auswahl und Aktivierung einer neuen Anwendungskonfiguration während der Laufzeit wird als dynamische Rekonfiguration bezeichnet.
Die Verwendung der Konfigurationsabstraktion bietet eine hohe Flexibilität bei der Realisierung des adaptiven Verhaltens. Anpassungen an einer verteilten Anwendung können auf einer breiten Ebene vorgenommen werden. Angefangen bei der Parameteranpassung, die das Verhalten auf der lokalen Ebene einer Kapsel verändert, über den Einsatz zusätzlicher Kapseln (z.B. einer Datenkompressionskapsel zwischen zwei über ein Netzwerk kommunizierende Kapseln) bis hin zur Migration der Objekte einer Kapsel auf einen anderen Rechner, falls auf diesem z.B. mehr Rechenleistung zur Verfügung steht, existieren vielfältige Möglichkeiten.

1.2 Eine Ausführungsumgebung für adaptive Anwendungen

Ein wichtiges **Ziel dieser Arbeit** war die Entwicklung und Integration von Techniken zur Ausführung adaptiver komponentenbasierter verteilter Anwendungen in einer Komponentenplattform. Weit verbreitete Vertreter aktueller Komponentenplattformen sind Suns Java-Plattform und Microsofts .NET Plattform. Eine Besonderheit dieser Plattformen ist der Einsatz einer virtuellen Maschine für die Ausführung von objektorientierten Anwendungen, um die durch den Einsatz verschiedener Betriebssysteme und Hardware-Plattformen entstandene Heterogenität der Rechner in einem verteilten System zu verbergen, aber auch um die Portabilität von Software zu erhöhen. Die virtuellen Maschinen führen eine portable Zwischenrepräsentation (*Bytecode, Intermediate Language*) der Softwarekomponenten aus, mit der plattformspezifische Details wie z.B. der Zugriff auf Register und den Stack oder die Verwaltung von dynamischem Speicher gekapselt wird. Durch die Verwendung einer virtuellen Maschine für eine spezifische Plattform kann die Zwischenrepräsentation einer Softwarekomponente ohne Neuübersetzung des Quellcodes ausgeführt werden, indem die virtuelle Maschine die Zwischenrepräsentation direkt vor der Ausführung in Maschineninstruktionen (*just-in-time compilation*) übersetzt.

Bei der Integration von Mechanismen zur Ausführung adaptiver Anwendungen in eine Komponentenplattform ist es zum einen möglich, existierende virtuelle Maschinen um notwendige Mechanismen zu erweitern; im Wesentlichen um die fehlende Rekonfigurationsfunktionalität direkt zu integrieren. Dieser Ansatz ist problematisch, da erweiterte Versionen der virtuellen Maschinen dem Aktualisierungsprozess des Herstellers angepasst werden müssen, das heißt, dass die Änderungen in eine neue Version eingepflegt werden müssen. Zusätzlich kann die Stabilität der virtuellen Maschine gefährdet und damit der Einsatz in Produktivumgebungen erschwert werden. Deshalb wurden im Rahmen dieser Arbeit Techniken entwickelt, um die dynamische Rekonfiguration in den Java- und .NET-Plattformen zu ermöglichen, ohne die virtuelle Maschine zu erweitern.

Hierfür wurden existierende **Basistechnologien** für die Integration verwendet. Dies ermöglicht den herstellerunabhängigen Einsatz der entwickelten Techniken. Eine wichtige Basistechnologie und Grundlage der im Rahmen dieser Arbeit entwickelten Techniken ist die Verwendung der vorgestellten virtuellen Maschinen für die Ausführung von Softwarekomponenten in einem heterogenen verteilten System. Des Weiteren wird die Möglichkeit des dynamischen Ladens von Komponenten verwendet, um einer Konfiguration während der Laufzeit Objekte neuer Komponenten hinzufügen zu können. Beim Laden der Funktionalität einer neuen Komponente muss zunächst das Deployment, also die Verteilung und Installation, von Komponenten in einem verteilten System untersucht werden, bei dem die entsprechenden Komponenten und ihre Abhängigkeiten an einen entfernten Rechner übertragen werden müssen. Die Basistechnologie der Metaprogrammierung wird zur Manipulation von Objekten einer Anwendung aus einer Metaebene verwendet, über welche die Konfiguration der Anwendung erfolgt.

Eine weitere verwendete Basistechnologie ist die aspektorientierte Programmierung [Kiczales u. a. 1997], mit der nichtfunktionale Aspekte separiert von der eigentlichen Anwendungslogik entwickelt werden können. Im Rahmen dieser Arbeit wurde konfigurationsspezifische Logik identifiziert und als Aspekt implementiert. Mit Hilfe eines Aspektwebers kann diese Logik in die verwendeten Softwarekomponenten integriert werden und muss damit nicht vom Hersteller der Softwarekomponenten entwickelt

werden. Die entwickelten Aspekte ermöglichen damit die Wiederverwendbarkeit der konfigurationsspezifischen Logik bei der Erstellung adaptiver Anwendungen. Eine komplexe Anwendung fand die aspektorientierte Programmierung im Rahmen dieser Arbeit durch die Integration von Synchronisationslogik in die Anwendungskomponenten, die bei der dynamischen Rekonfiguration benötigt wird.

1.3 Dynamische Rekonfiguration

Der Wechsel von einer Anwendungskonfiguration zu einer anderen während der Laufzeit einer Anwendung wird als dynamische Rekonfiguration bezeichnet. Bei der dynamischen Rekonfiguration können verschiedene Konfigurationsänderungen unterschieden werden:

- Anpassung von Kapselparametern (z.B. Kompressionsrate)
- Hinzufügen/Entfernen von Kapseln
- Anpassung von Verbindungen zwischen Kapseln
- Austausch der Algorithmen einer Kapsel (Dynamische Aktualisierung)
- Änderung der Platzierung von Kapseln auf andere Rechner (Migration)

Eine dynamische Rekonfiguration kann komplexe Änderungen an der Struktur und der Logik einer Anwendung bedeuten, welche die Konsistenz der Daten und die Funktionalität der Anwendung insgesamt nicht beeinträchtigen dürfen. Hierfür müssen geeignete Algorithmen verwendet werden, um zu verhindern, dass Anwendungsaktivitäten auf Grund nicht abgeschlossener Rekonfigurationsoperationen Inkonsistenzen verursachen. Für die dynamische Rekonfiguration komponentenbasierter Anwendungen existieren Algorithmen [Bidan u. a. 1998; Kramer und Magee 1990; Moazami-Goudarzi 1999; Wermelinger 1997], die auf dem „Einfrieren“ der Anwendungen in einem konsistenten Zustand basieren. Dieser Zustand wird im Rahmen dieser Arbeit als rekonfigurierbarer Zustand bezeichnet.
Übertragen auf die Java- und .NET-Komponentenplattformen bedeutet das Erreichen eines rekonfigurierbaren Zustands, dass keine offenen Methodenaufrufe in den Objekten der betroffenen Kapseln existieren. Dies ist vor allem dadurch bedingt, dass ohne eine Manipulation der virtuellen Maschine kein Zugriff auf den Stack und die Register möglich ist, die potentiell Zustand in Form von lokalen Variablen enthalten können. Dies ist bei der dynamischen Aktualisierung problematisch, da auf dem Stack Referenzen auf Objekte existieren können, die aktualisiert wurden. Ein wichtiges Ziel bei der dynamischen Rekonfiguration ist das Erkennen eines rekonfigurierbaren Zustands, in dem kein Thread Instruktionen einer Methode der Objekte einer von der Rekonfiguration betroffenen Kapsel ausführt. Da dieser Zustand nicht immer erreicht werden kann, weil immer wieder eine neue Methode aufgerufen werden kann, während ein vorheriger Methodenaufruf noch verarbeitet wird, müssen neue Methodenaufrufe blockiert und die Beendigung aktiver Methodenaufrufe abgewartet werden, um einen rekonfigurierbaren Zustand herbeizuführen.
Im Falle verschachtelter Aufrufe muss beim Blockieren eine Reihenfolge beachtet werden, da sonst potentiell nicht alle aktiven Methodenaufrufe beendet werden können, weil diese auf die Ausführung verschachtelter Aufrufe warten. Im Falle azyklischer

Verbindungsgraphen kann das Erreichen des rekonfigurierbaren Zustands durch das geordnete Blockieren von Konnektoren implementiert werden [Wermelinger 1997], bei dem auf das Ende aktiver Methodenaufe über diesen Konnektor gewartet wird und neue Aufrufe blockiert werden. In diesem Zusammenhang untersucht die vorliegende Arbeit Techniken, die notwendige Synchronisationslogik automatisiert in die Komponenten der Anwendung zu integrieren. Mit Hilfe der aspektorientierten Programmierung konnte so die Synchronisationslogik wiederverwendbar für die Entwicklung adaptiver Anwendungen zur Verfügung gestellt werden und somit die Entwicklung vereinfacht werden.

Es hat sich herausgestellt, dass viele der existierenden Rekonfigurationsalgorithmen nicht für die Integration in eine Komponentenplattform wie der Java- bzw. .NET-Plattform verwendet werden können, weil diese Verfahren keine Unterstützung multipler Threads oder re-entranter Kapseln bieten. Re-entrante Kapseln können zyklische Aufrufbeziehungen von verschachtelten Methodenaufrufen verursachen, die beim Erreichen des rekonfigurierbaren Zustands problematisch sind. Eine zyklische Aufrufbeziehung entsteht, wenn ein Thread aus dem Kontext einer Methode weitere Methoden aufruft und dabei gleichzeitig zwei laufende Methodenrufe in einer Kapsel besitzt. Dies kann genau dann entstehen wenn im Verbindungsgraphen einer Anwendungskonfiguration Zyklen existieren. In diesem Fall kann für das Blockieren von Verbindungen keine Reihenfolge ermittelt werden, um Verklemmungen zu vermeiden.

Für Anwendungskonfigurationen mit zyklischen Verbindungsgraphen wurde deshalb im Rahmen dieser Arbeit ein neuer Algorithmus entwickelt, bei dem selektiv einzelne Methodenaufrufe blockiert werden. Der entwickelte Algorithmus basiert auf threadspezifischen Datenstrukturen, die in jeder Kapsel bei Ein- und Austritt eines Threads aktualisiert werden. Bei einer dynamischen Rekonfiguration kann für jeden Thread ermittelt werden, ob dieser aktive Methodenaufrufe an einer von der Rekonfiguration betroffenen Kapseln besitzt und damit neue Methoden aufrufen darf; andernfalls kann der Thread blockiert werden. Durch den Einsatz logischer Thread-IDs, die Threads über Rechnergrenzen hinweg eindeutig identifizieren, unterstützt das entwickelte Verfahren auch die Rekonfiguration verteilter Anwendungen. Eine besondere Eigenschaft des neuen Rekonfigurationsverfahren ist, dass die notwendige Synchronisationslogik nur geringe Zusatzkosten (in Form von zusätzlichen Instruktionen) für normale Methodenaufrufe verursacht.

Ein Spezialfall der dynamischen Rekonfiguration stellt die **dynamische Aktualisierung** von Kapseln dar. In diesem Fall liegen die entsprechenden Softwarekomponenten in einer neuen Version vor oder alternative Implementierungen sollen ohne einen Neustart der Anwendung aktiviert werden. Bei der dynamischen Aktualisierung ergibt sich dabei das Problem des Zustandstransfers zwischen alter und neuer Version. Existierende Ansätze realisieren dynamische Aktualisierungen durch den Einsatz redundanter Hardware oder erweiterter Laufzeitumgebungen [Dmitriev 2001]. Auch der Einsatz der Serialisierungsfunktionalität aktueller Komponentenplattformen für den Zustandstransfer ist nicht möglich, da Versionsinformationen Teil der persistierten Daten sind und bei der Deserialisierung auf die Softwarekomponenten der alten Version zugegriffen wird. Im Rahmen dieser Arbeit wurde ein Verfahren entwickelt, das die komplexe Rekonfigurationsoperation „Dynamische Aktualisierung“ ohne die Manipulation der Laufzeitumgebung ermöglicht. Durch die Verwendung der Metaprogrammierung (Reflexion) kann durch das Traversieren aller Objekte einer Kapsel erkannt werden, welche Objekte durch neue Versionen ersetzt werden müssen. Der Zustand der alten Objekte

wird durch attributweises Kopieren auf die neuen Objekte, die aus der aktualisierten Softwarekomponente erzeugt werden, übertragen.
Während die dynamische Rekonfiguration in modernen Komponentenplattformen den zentralen Fokus dieser Arbeit darstellt, wurden einige weiterführende Aspekte untersucht, die für die Funktionsweise und Entwicklung adaptiver Anwendungen von Bedeutung sind.

1.4 Funktionsweise adaptiver Anwendungen

Adaptive Anwendungen müssen in der Lage sein, Änderungen der Umgebungseigenschaften und ihres internen Zustands zu erkennen und entsprechende Rekonfigurationsmaßnahmen durchzuführen. Die Diagnose von Änderungen kann entweder durch periodische Messung der Umgebungseigenschaften und der internen Struktur oder ereignisbasiert durch entsprechende Mechanismen der unterliegenden Infrastruktur erfolgen. Die Umgebungseigenschaften können dabei die verfügbaren Ressourcen aber auch weitere Parameter wie den Kontext eines Nutzers (z.B. seine Position) umfassen. Abbildung 1.2 zeigt die Funktionsweise einer adaptiven Anwendung unter Verwendung eines Regelkreises. Ausgehend von der periodischen Messung der Umgebungseigenschaften und dem Zustand der Anwendung in einer Analysephase werden Einstellungen an der Anwendung vorgenommen.

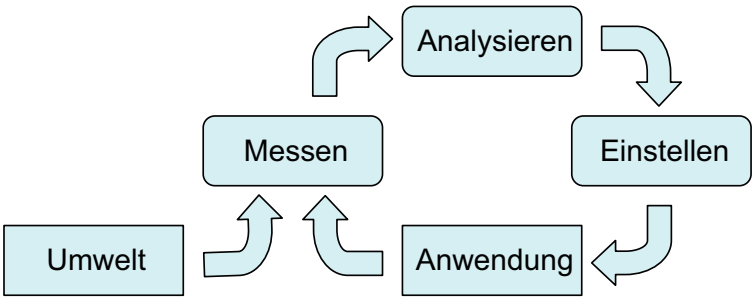


Abbildung 1.2: Zyklus einer adaptiven Anwendung

Die wesentlichen Grundmechanismen für die Realisierung adaptiver Anwendungen lassen sich in drei Bereiche unterteilen. Zunächst müssen die Umgebungseigenschaften ermittelt werden. Diese Funktionalität wird als **Monitoring** bezeichnet. Die Anpassung der Anwendung erfolgt durch die Auswahl einer geeigneten Anwendungskonfiguration und einer dynamischen Rekonfiguration der Anwendung. Schlussendlich muss die **Zuordnung** zwischen ermittelten Umgebungseigenschaften und den alternativen Konfigurationen beschrieben werden.
Neben der Implementierung einer Ausführungsplattform für adaptive Anwendungen, die das Deployment von Softwarekomponenten in verteilten Systemen, das Monitoring von Umgebungseigenschaften und Kapselausfällen sowie der Integration heterogener Java- und .NET-Kapseln untersucht, wurde im Rahmen der Arbeit ein graphisches Werkzeug zur Erstellung von Anwendungskonfigurationen sowie der Konfiguration des Monitoring und der Zuordnung von Konfiguration zu gemessenen Umgebungseigenschaften entwickelt. Das graphische Werkzeug unterstützt dabei auch die Entwicklung alternativer Konfigurationen mit Hilfe von Architekturmustern, mit denen auf ausgewiesene Änderungen der Umgebung reagiert werden kann. Die aspektorientierte

Programmierung spielt bei der Werkzeugunterstützung eine zentrale Rolle, denn sie ermöglicht, dass komplexe (re-)konfigurationsspezifische Logik generiert werden kann, und erleichtert damit die Entwicklung adaptiver Anwendungen. Hierfür wurde ein neues Programmiermodell für die Identifikation von Kommunikationsendpunkten und Parametern in den Komponenten entwickelt, die bei der graphischen Konfigurationserstellung für die Verbindung und Parametrierung von Kapseln verwendet werden.
Der entwickelte Ansatz zur Entwicklung adaptiver Anwendung wurde im Rahmen des **Distributed Control Lab** (DCL), einer entfernten Laborumgebebung für die Ausführung von Steuerungsexperimenten über das Internet, für die sichere Ausführung von studentischen Steuererungsprogrammen eingesetzt. Im DCL werden über einen Web-Browser eingegebene Programme, die Algorithmen für die Steuerung einer physikalischen Hardware enthalten, auf einem eingebetteten Rechner installiert und ausgeführt. Die über das Internet abgesendeten, potentiell fehlerhaften bzw. bösartigen Programme werden durch die entwickelte Ausführungsumgebung während der Laufzeit überwacht und beim Erkennen fehlerhaften Verhaltens oder dem Ausfall einer Nutzersteuerung (z.B. durch eine unbehandelte strukturierte Ausnahme) durch eine dynamische Rekonfiguration der Steuerungsanwendung ersetzt.

1.5 Struktur der Arbeit

Im Rahmen dieses Abschnitts wird die Struktur der vorliegenden Arbeit und der Zusammenhang der einzelnen Projekte, die im Rahmen der Arbeit beschrieben werden, vorgestellt. Eine Übersicht ist in Abbildung 1.3 dargestellt. Im Mittelpunkt der Arbeit steht dabei die Entwicklung adaptiver Anwendungen durch die dynamische Rekonfiguration komponentenbasierter Anwendungen, für die alternative Anwendungskonfigurationen für verschiedene Umgebungssituationen vorliegen. Die Grundpfeiler der Arbeit stellen Algorithmen zur dynamischen Rekonfiguration und dem Spezialfall der dynamischen Aktualisierung dar, die in den zentralen Kapiteln 3 und 4 beschrieben werden. Basierend auf den entwickelten Techniken wurde eine komplexe Fallstudie im Rahmen des Distributed Control Lab realisiert, die im 6. Kapitel der Arbeit vorgestellt wird.

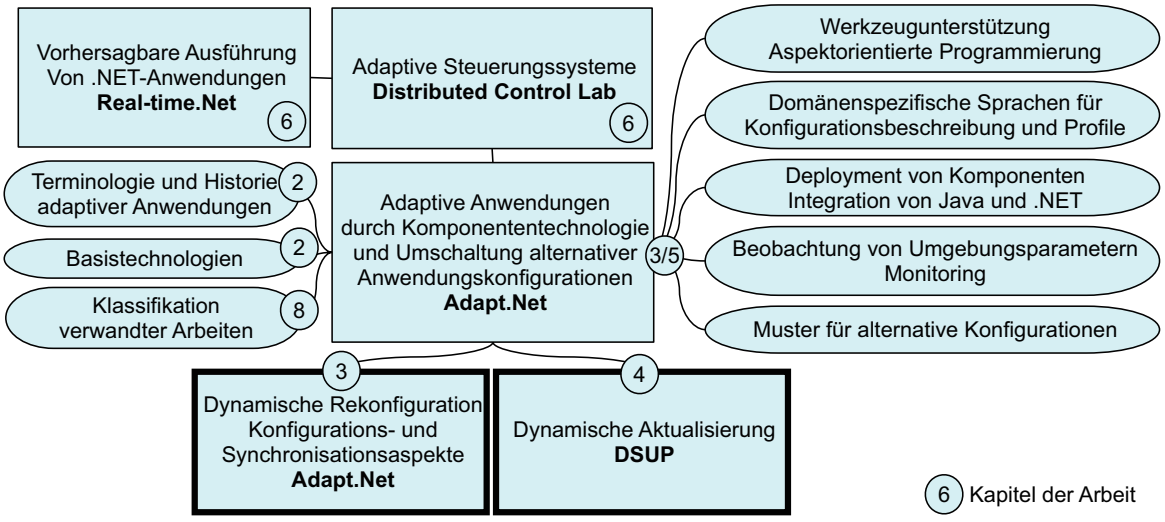


Abbildung 1.3: Strukur and Themengebiete der Arbeit

Im der Einleitung anschließenden 2. Kapitel wird die historische Entwicklung und Bestrebungen im Bereich adaptiver Software nach der Einführung der grundlegenden Terminologie in diesem Umfeld vorgestellt. Zusätzlich werden bedeutende Vertreter von Komponentenplattformen vorgestellt und wichtige Basistechnologien diskutiert.
Den Kern der Arbeit bildet die Entwicklung einer Ausführungsplattform für adaptive komponentenbasierte verteilte Anwendungen und entsprechende Algorithmen für die dynamische Rekonfiguration. Im ersten Teil des zentralen 3. Kapitels der Arbeit wird ein Modell der eingesetzten Komponentenplattformen eingeführt, auf dessen Basis ein neuer Rekonfigurationsalgorithmus zunächst abstrakt vorgestellt wird. Im zweiten Teil des 3. Kapitels wird die Implementierung der im Rahmen des Adapt.Net-Projekts entwickelten Ausführungsplattform für adaptive Anwendungen beschrieben und dabei die für die dynamische Rekonfiguration notwendigen Synchronisationsverfahren diskutiert. Hierbei wird auch das Deployment von Softwarekomponenten in verteilten Systemen, die Separierung konfigurationsspezifischer Aspekte sowie eine domänenspezifische Sprache zur Beschreibung von Anwendungskonfigurationen diskutiert. Die dynamische Aktualisierung steht im Mittelpunkt des 4. Kapitels. Im Rahmen des Projekts DSUP (*Dynamic Software Update Platform*) wurde die dynamische Aktualisierung entkoppelt untersucht und die Ergebnisse in die Adapt.Net-Ausführungsplattform integriert. DSUP kann dabei auch separat für das Einspielen fehlerbereinigter Softwarekomponenten in verteilte Anwendungen verwendet werden.
Im 5. Kapitel der Arbeit werden weiterführende Aspekte adaptiver Anwendungen diskutiert. Darunter ist die profilbasierte Zuordnung zwischen gemessenen Umgebungseigenschaften und Anwendungskonfigurationen, die Implementierung einer Monitoring-Infrastruktur sowie ein graphisches Werkzeug für die Erstellung adaptiver Anwendungen, mit dem auslieferbare Pakete einer adaptiven Anwendung erstellt werden können. Zusätzlich werden in diesem Kapitel Architekturmuster für die Erstellung alternativer Anwendungskonfigurationen diskutiert, mit denen auf typische Änderungen der verfügbaren Ressourcen sowie Ausfälle von Kapseln reagiert werden kann.
In den anschließenden Kapiteln 6 und 7 werden Fallstudien des in dieser Arbeit vorgestellten Ansatzes beschrieben. Im Rahmen des *Distributed Control Lab* wird die Adaption in Steuerungsanwendungen diskutiert und ein Verfahren zur sicheren Ausführung von Steuerungen in einem entfernten Labor vorgestellt. In diesem Umfeld wurde auch die vorhersagbare Ausführung von .NET-Anwendungen im Rahmen des Real-Time.Net-Projekts untersucht, einer Voraussetzung für die Nutzung der Komponentenplattformen in Echtzeitsystemen. In diesem Zusammenhang wurde weiterführend in Form eines Ausblicks die Vorhersagbarkeit der eingesetzten Rekonfigurationsalgorithmen untersucht. Inhalt des 7. Kapitels ist eine Leistungsbewertung im Rahmen weiterer Fallstudien.
Anschließend werden im 8. Kapitel verwandte Arbeiten auf dem Gebiet der adaptiven Software sowie der dynamischen Rekonfiguration beschrieben und die Abgrenzung zur vorliegenden Arbeit vorgenommen. Die Zusammenfassung und die Vorstellung zukünftiger Arbeiten bilden den Abschluss der Arbeit.

1.6 Beitrag der Arbeit

Der Autor beschreibt einen neuartigen Ansatz zur Entwicklung adaptiver komponentenbasierter Software. Die Ausdehnung der Komponentenabstraktion der Java- und .NET-Komponentenplattformen auf die Laufzeit eröffnet die Möglichkeit der Realisie-

rung adaptiven Verhaltens mit Hilfe der Komponentenorientierung. Eine im Rahmen der Arbeit entwickelte Ausführungsplattform unterstützt die Adaption durch Änderungen der beteiligten Komponenten, ihrer Parameter und der Verbindungsstruktur, was durch die Vorstellung von Architekturmustern für alternative Anwendungskonfigurationen untermauert wird. Anwendungskonfigurationen können dabei unabhängig entwickelt und in für sie bestimmten Einsatzumgebungen getestet werden.

Der vom Autor entwickelte Rekonfigurationsalgorithmus verbessert existierende Lösungen bzgl. der Komplexität der Zusatzkosten durch die notwendige Synchronisationslogik und unterstützt dabei *Multithreading* und zyklische Verbindungsstrukturen. Ausgehend von einem theoretischen Modell komponentenbasierter Anwendungen und der Formalisierung einer Laufzeitumgebung, kompatibel zu aktuellen Komponentenplattformen, wurden wichtige Eigenschaften des neuen Algorithmus u.a. unter Einsatz formaler Methoden evaluiert. Die ausführliche Analyse möglicher Synchronisationsstrategien für die Implementierung dynamischer Rekonfiguration geht über existierende Untersuchungen hinaus.

Das vom Autor präsentierte Verfahren zur dynamischen Aktualisierung von Kapseln ermöglicht die Aktivierung alternativer Algorithmen während der Laufzeit, kann aber auch für das Einspielen neuer Softwareversionen zur Fehlerbehebung verwendet werden. Durch die Nutzung von Basistechnologien kann das Verfahren auf einer Vielzahl von Plattformen eingesetzt werden, ohne dass Manipulationen der Laufzeitumgebung notwendig sind. Das entwickelte Verfahren wurde mit einer leistungsfähigen Implementierung untermauert, wobei komplexe Details bei der Traversierung von Objektgraphen untersucht wurden. Zusätzlich konnte mit Hilfe der entwickelten Techniken die Anpassung von graphischen Nutzerschnittstellen realisiert werden.

Besonders hervorzuheben ist der Einsatz der aspektorierentierten Programmierung (AOP) zur Integration konfigurationsspezifischer Logik in die funktionalen Komponenten. Die komplexe Implementierung von Konfigurationsschnittstellen für die Erstellung von Kommunikationsverbindungen und den Zugriff auf Kapselparameter, aber vor allem die Integration der für die dynamische Rekonfiguration nötigen Synchronisationslogik konnte im Rahmen dieser Arbeit wiederverwendbar als Aspekt implementiert werden und kann so die Entwicklung adaptiver Anwendungen beschleunigen. Zusätzlich ist mit den im Rahmen dieser Arbeit entwickelten Aspekte ein komplexes Einsatzszenario für die aspektorientierte Programmierung gefunden.

Schlussendlich leistet diese Arbeit einen wichtigen Beitrag für die Bereitstellung von Steuerungsexperimenten über das Internet. Im Rahmen des Distributed Control Lab kann die Beschädigung von Experiment-Hardware durch den Einsatz der entwickelten Rekonfigurationsalgorithmen verhindert werden. Das Distributed Control Lab wurde vom Autor dieser Arbeit in zahlreichen Lehrveranstaltungen unter anderem an der Blekinge Techniska Högskola in Ronneby, Schweden während eines 3-montigen Forschungsaufenthalt erfolgreich eingesetzt. Im Leonardo da Vinci Projekt der Europäischen Union werden im Vet-Trend-Projekt die entwickelten Techniken sowie die Laborinfrastruktur auf transnationaler Ebene vernetzt.

Die vorliegende Arbeit beschreibt einen Ansatz zur Entwicklung adaptiver komponentenbasierter Anwendungen. Diese Anwendungen umfassen dabei interaktive verteilte Anwendungen mit einer begrenzten Anzahl an Komponenten. Zusätzlich wird die Anwendbarkeit des Ansatzes im Bereich der Steuerungsanwendungen demonstriert. Als mobile Geräte wurden zunächst Notebook-Rechner betrachtet, die über verschiedene Netzwerke wie WLAN, LAN oder GSM kommunizieren. Kleinere mobile Geräte

wurden bei der Implementierung zunächst ausgeschlossen. Bei der Implementierung werden explizit die Java- und .NET-Plattformen betrachtet. Obwohl die entwickelten Konzepte anwendbar sind, werden aus Gründen der eingeschränkten Performance keine dynamischen Sprachen wie CLOS, Smalltalk oder Python und wegen der fehlenden Typsicherheit kein C++ betrachtet. In diesen Fällen wären einige Implementierungsdetails verschieden zu lösen.
Die Arbeit resultierte in zahlreichen Veröffentlichungen auf internationalen Konferenzen und Workshops. Darunter sind [von Löwis und Rasche 2006; Rasche 2006; Rasche und Polze 2002, 2003, 2005, 2008; Rasche u. a. 2005a, 2004; Rasche und Schult 2007; Rasche u. a. 2005c, 2003; Richter u. a. 2007; Tröger u. a. 2008] sowie ein Buch- [Polze und Rasche 2005] und ein Journalbeitrag [Rasche u. a. 2005b].
Einige weiterführende Teilaspekte wurden in Master- und Diplomarbeiten, die vom Autor dieser Arbeit mitbetreut wurden, untersucht. Darunter befinden sich die Arbeiten [Barthel 2008; Borchert 2005; Issel 2006; Leidner 2006; Müller und Widmer 2005; Puhlmann 2005; Richter 2006; Schöbel 2005; Senf 2006; Tiedt 2007; Widmer 2007]. Zusätzlich co-betreute der Autor die Bachelor-Projekte „Hochverfügbare Web-Schnittstellen für Steuerungssysteme“, „Steuerung einer Fertigungsstraße mit Industrie-PCs“, „Entwicklung einer transnationalen experimentbasierten Lernumgebung im Leonardo Da Vinci Programm der Europäischen Union“ und „Eingebettete Steuerungssysteme und Echtzeitkommunikation in der Firma Beckhoff Automation GmbH“. Ein solches Bachelor-Projekt an dem 4-6 Studenten beteiligt sind, wird am Ende des Bachelor-Studiums für die Dauer von 6 Monaten durchgeführt.

1.7 Hinweise zur Schreibweise

In der Informatik finden sehr oft englische Fachbegriffe Verwendung, für die versucht wurde, eine deutsche Entsprechung zu finden. In einigen Fällen ließ sich die Verwendung der Originalbegriffe nicht vermeiden. In diesem Fall werden englische Begriffe kursiv hervorgehoben. Oft gebrauchte englische Begriffe werden normal gesetzt, um einen besseren Textfluss zu gewährleisten (z.B. Thread, Deployment, Software, Middleware, Client, Server, Proxy, Framework). Neu eingeführte Begriffe und wichtige Inhalte bestimmter Absätze werden fett gedruckt hervorgehoben.

2 Terminologie und Grundlagen

In diesem Kapitel sollen zunächst einige im Rahmen der Arbeit verwendete Begriffe eingeführt werden. Grundlegende Projekte und Bestrebungen sowie wichtige Meilensteine auf dem Weg zu adaptiver Software werden im Anschluss beschrieben, währenddessen verwandte Arbeiten zum Abschluss der Arbeit in einem separaten Kapitel diskutiert werden, um eine vergleichende Analyse zu den in der Arbeit vorgestellten Techniken vornehmen zu können. Schließlich wird eine Auswahl an Komponentenplattformen als Ausführungsumgebung für adaptive Anwendungen untersucht und in der Arbeit eingesetzte Basistechnologien beschrieben.

2.1 Terminologie

Der Begriff der **Adaption** ist vom lateinischen Verb „adaptare“ gebildet, das mit „anpassen“ zu übersetzen ist. Der Adaptionsbegriff in der Informatik wurde der Bedeutung in der Biologie entlehnt. Unter der evolutionären Adaption versteht man die Veränderung von Körperbau und Verhalten als Reaktion auf spezielle Umweltbedingungen [Darwin 1859]. Die Organismen stellen sich dabei auf die jeweiligen Umwelt- oder Reizbedingungen ein. In dieser Arbeit wird der Begriff Adaption für die automatisierte Anpassung von Software an die aktuellen Umgebungsbedingungen gebraucht. Ein in diesem Zusammenhang oft verwendeter Begriff ist der der selbst-adaptiven Systeme, der im Rahmen der Selbst-Eigenschaften aus dem *Autonomic Computing* [Kephart und Chess 2003] gebildet wurde. Da der Adaptionsbegriff schon einen Automatismus (also ein selbst) der Anpassung einschließt, scheint dieser Begriff zunächst wenig sinnvoll. Allerdings wird seit kurzer Zeit unter Selbst-Adaptivität die eigenständige Planung bzw. auch die Intelligenz von Software verstanden, selbstständig die Einhaltung von Zielvorgaben (*goals*) zu erfüllen auch wenn unvorhergesehene Änderungen der Umgebung eintreten [Kramer und Magee 2007]. Im Gegensatz dazu umfassen adaptive Anwendungen als Abgrenzung auch Anpassungen an vorhersehbare Änderungen, sind also eine Obermenge selbst-adaptiver Anwendungen. Mary Shaw beschrieb den Unterschied zwischen adaptiven und selbst-adaptiven Systemen im Rahmen eines Dagstuhl-Seminars über die Unsicherheit (uncertainty) der Umgebungssituation während der Anwendungserstellung. Im Unterschied zu adaptiven Anwendungen welche die Fähigkeit zu selbstständigen Anpassungen besitzen, besteht bei **adaptierbaren** (*adaptable*) Anwendungen die Möglichkeit zur Adaption von Außen.

Der Adaptionsbegriff wird in der Literatur mit den Attributen **statisch** und **dynamisch** verbunden, dabei bedeutet die statische Adaption die Anpassung vor der Anwendungslaufzeit. Dynamische Adaption hingegen findet während der Anwendungslaufzeit statt.

Die **Konfiguration** einer komponentenbasierten Anwendung beschreibt die Menge der beteiligten Bestandteile, die als Kapseln bezeichnet werden, deren Verbindungen sowie eine Abbildung, die Kapseln auf beteiligte Rechner eines verteilten Systems zuordnet.

Eine formale Spezifikation des Konfigurationsbegriffs befindet sich im 3. Kapitel dieser Arbeit.
Eine **Rekonfiguration** ist der Vorgang der Änderung einer Konfiguration. Eine Rekonfiguration kann während der Laufzeit stattfinden (**dynamisch**) oder vor dem Start einer Anwendung (**statisch**). Eine Rekonfiguration kann **automatisch** durch eine Software ausgelöst werden oder **manuell** durch einen Anwender. Findet eine Rekonfiguration mit dem Ziel einer Anpassung statt, wird sie als **adaptiv** bezeichnet. Der Begriff der dynamischen Rekonfiguration wird auch im Bereich der technischen Informatik für die Änderung und Optimierung von logischen Schaltungen (z.B. FPGAs) während des Betriebs verwendet. Es existiert eine Reihe von Veröffentlichungen (siehe [Vaidyanathan und Trahan 2004]) aus diesem Bereich, wobei sich die verwendeten Techniken zwischen Hard- und Software stark unterscheiden. Im Rahmen dieser Arbeit wird ausschließlich die dynamische Rekonfiguration von Software untersucht.
Ein im Zusammenhang mit der Anpassung von Software oft gebrauchter Begriff ist „Evolution“, welcher nach [Kramer und Magee 1990] die Weiterentwicklung von Software zur Anpassung an neue Technologien oder neuer Anforderungen der Nutzer umfasst. [Heineman 1998] versteht unter **Evolution** die Anpassung von Komponenten für den Einsatz in neuen Umgebungen durch den Anwendungsentwickler. Im Gegensatz zur Adaption, die sich mit der Anpassung von Software an eine sich wiederkehrend verändernde Umgebung beschäftigt, behandelt die *Evolution* den Fortschritt bei der Softwareentwicklung. Als **Portieren** wird in der Informatik die Anpassung einer Software an ein Betriebssystem oder eine neue Hardware bezeichnet, die während der Implementierungszeit stattfindet.
Im Entwicklungsprozess und Lebenszyklus von Software finden sich einige Aktivitäten, welche die Adaption tangieren. Die einzelnen Phasen und mögliche Adaptionsmechanismen in diesen Phasen sollen kurz vorgestellt werden. Diese Arbeit beschäftigt sich mit der (dynamischen) Adaption zur **Laufzeit**, die nach dem Start (**Startphase**) der Anwendung beginnt. Vor dem Start müssen Anwendungen verteilt und installiert werden; dies geschieht während der **Deployment-Phase**, in der Anpassungen durch die Verwendung alternativer Bibliotheken vorgenommen werden können. Zusätzlich können während der Deployment-Phase Konfigurationsdokumente erstellt werden, welche Anpassungen in der Startphase auslösen. Die klassischen Phasen im Softwareentwicklungsprozess der Spezifikations-, **Entwurfs**- und **Implementierungsphase** ordnen sich vor dem Deployment-Zeitraum ein. Von Interesse für diese Arbeit ist auch die **Übersetzungszeit**, die sich der Implementierungsphase anschließt. Auch hier können Anpassungen durch die Wahl alternativer Module konfiguriert werden.
Im Rahmen dieser Arbeit wird der Begriff Adaption synonym mit dynamischer Adaption verwendet, da Anpassungen an wechselnde Umgebungen während der Laufzeit erfolgen. Bei der Entwicklung adaptiver Anwendungen müssen aber in weiteren Phasen der Softwareentwicklung zusätzliche Verfahren integriert werden. Dies beginnt bei der Auswahl von Alternativen in der Entwurfsphase und führt bis zur Integration von Adaptationsmechanismen während der Implementierungs- bzw. Deployment-Phase.

2.2 Auf dem Weg zu adaptiver Software

Eine frühe Anwendung fand Adaption in den 60er-Jahren bei der Flugzeugentwicklung [Sastry und Bodson 1989]. Adaptive Regler, die auch noch heute oft Verwendung finden, wurden für die Fluglagekontrolle eingesetzt. Für die unterschiedlichen Phasen

während des Fluges (z.B. Start, Landung, Steigflug) mussten verschiedene Parameter für die eigentliche Regelung eingestellt werden, um z.B. unterschiedlichen Druck- und Windbedingungen gerecht zu werden. Gelöst wurde das Problem durch den Einsatz einer zweiten Regelung, welche die Hauptregelung der Fluglage parametriert. Diese zusätzliche Regelung ist in der Lage, die Umgebungssituation zu ermitteln, entsprechende Parameter für den Hauptregler zu konfigurieren und damit adaptives Verhalten zu realisieren.

Im Bereich der Softwareentwicklung wurde die Problematik von Laufzeitänderungen an Softwaremodulen erstmals im Jahre 1976 durch [Fabry 1976] beschrieben. Schon zuvor wurde mit Multics [Corbato und Vyssotsky 1965] im Jahre 1964 das dynamische Linken eingeführt, mit dem es Programmen ermöglicht wurde, neue Speichersegmente in ihren Adressraum zu laden, in denen sich auch ausführbarer Code befinden konnte. Dies ist eine wichtige Voraussetzung für *Plugins*, die nachladbare (neue) Funktionalität enthalten. Im Gegensatz zum Nachladen neuer Funktionalität diskutiert [Fabry 1976] die Änderung von geladenen Modulen. Im Prinzip lässt sich die Laufzeitmodifikation von Software bis zur ENIAC zurückverfolgen, die selbstmodifizierenden Code mit Hilfe von *Overlays* unterstützte. Erst Ende der 80er-Jahre wurde die Änderung der Struktur von Software und der Konfiguration von Modulen von einer breiteren Forschungsgemeinde untersucht. Ausgehend von dem Paradigma des „*Programming-in-the-Large*“ [DeRemer und Kron 1976], waren die formale Beschreibung von evolutionären Änderungen von Software und der dynamische Rekonfigurationsansatz von Kramer und Magee [Kramer und Magee 1990] Ausgangspunkt weiterer Arbeiten. Von besonderer Bedeutung in der Arbeit von Kramer und Magee war die erstmalige Trennung von Modulimplementierung und Konfigurationsbelangen. Fast gleichzeitig wurde mit Argus [Bloom und Day 1993] ein System mit Unterstützung dynamischer Rekonfiguration entwickelt, bei dem die Konsistenzerhaltung durch forcierte Terminierung von Teilsystemen mit verbundener Wiederherstellung gesicherten Zustands realisiert wurde. Eine ausführliche Übersicht über die Arbeiten auf dem Gebiet der dynamischen Rekonfiguration erfolgt im Abschnitt 8.2.

Selbst heute lassen sich wenige Beispiele für Adaption in existierenden Produkten finden. Prominenteste Vertreter siedeln sich im Bereich der Multimedia-Anwendungen an. Apples Quicktime, Real Networks RealPlayer oder Microsofts Mediaplayer unterstützen seit kurzem die dynamische Anpassung von Datenraten bei der Übertragung über das Internet. Während die gewünschte Datenrate in vorherigen Versionen statisch über entsprechende Menüs konfiguriert werden konnte, unterstützen neuere Versionen die automatisierte Anpassung während der Laufzeit. Die Multimediadaten liegen in alternativen Versionen auf dem Server vor. Mit Hilfe des RTP (Realtime Transport Protocol) Control Protocol [Schulzrinne u. a. 2003] berichten Client-Rechner über die Qualität der ausgelieferten Daten. An Hand dieser Daten kann der Server über die auszuliefernde Version der Daten entscheiden.

Schon seit den 80iger Jahren unterstützt das TCP/IP-Protokoll die Adaption protokollspezifischer Parameter und ist damit ein oft genanntes Beispiel wissenschaftlicher Veröffentlichungen zum Thema adaptiver Software. Das TCP/IP-Protokoll stellt die Kontrollfenstergröße (*control window size*) sowie die Retransmissionsraten (*retransmission rate*) in Abhängigkeit der Eigenschaften des zu Grunde liegenden Netzwerkes ein [RFC793 1981]. Bei diesem Beispiel wird die Anpassung an die sich ändernde Zuverlässigkeit der Kommunikationsverbindung realisiert. Die Anpassung findet dynamisch also während der Laufzeit statt.

Im Bereich komplexer Unternehmenssoftware wird die Entwicklung adaptiver Anwendungen seit der Initiierung der *Autonomic-Computing*-Initiative im Jahre 2001 durch IBM vor allem auch von den großen IT-Firmen HP (*Adaptive Enterprise*), Microsoft (*Dynamic Systems*) und IBM (*Autonomic Computing*) untersucht. Einige der in diesem Bereich entstandenen Techniken werden im weiteren Verlauf dieses Abschnitts vorgestellt.

Die Geschichte der Entwicklung von Techniken für adaptive Software lässt sich für das relativ junge Gebiet der Informatik schon recht lange zurückverfolgen. Aber erst die Anforderungen moderner IT-Systeme erfordern die Weiterentwicklung entsprechender Techniken. Vor allem aber die Verfügbarkeit der Komponententechnologie und entsprechender Programmierplattformen eröffnet heute neue Möglichkeiten für die Realisierung von Adaption mit einfacheren Programmiermodellen. An dieser Stelle leistet die vorliegende Arbeit den Beitrag, auch mit der Betrachtung existierender Techniken, neuartige Möglichkeiten für die Entwicklung adaptiver Software bereitzustellen.

Nach dem kurzen geschichtlichen Abriss der Entwicklung adaptiver Software werden im Folgenden wichtige Bestrebungen vorgestellt, die das Forschungsgebiet geprägt haben.

2.2.1 Allgegenwärtige Rechner

Wesentlich bestimmt wurde die Erforschung adaptiver Techniken durch die Vision des allgegenwärtigen Rechners (*Ubiquitous Computing*). Der Begriff des allgegenwärtigen Rechners wurde von Mark Weiser in seinem Artikel aus dem Jahre 1991 [Weiser 1991] vorgestellt, welcher für das 21. Jahrhundert die Allgegenwärtigkeit des Rechners vorhersagt. Er beschreibt, dass der Rechner nicht mehr als Gegenstand an sich im Mittelpunkt steht, sondern vielmehr durch seine Funktionalität geprägt wird - seine Existenz nur durch eine Anwendung gerechtfertigt wird. Der Rechner in seiner heutigen Form verschwindet im Alltag (*„The disappearing computer“*).

Die Umsetzung dieser Vision, die heute von vielen Wissenschaflern aufgegriffen wurde und sich einer breiten Forschungsgemeinschaft erfreut, ist dabei sehr komplex. Im Rahmen des *Ubiquitous Computing* müssen eine Reihe von Fragestellungen gelöst werden. Die Realisierung adaptiver Anwendungen spielt dabei eine zentrale Rolle. Die Einbettung der Rechner in die Umgebung des Menschen erzwingt die Verkleinerung der Rechner und damit einhergehend eine Einschränkung der verfügbaren Ressourcen. Für tragbare Geräte wird eine drahtlose Kommunikation mit der Umgebung notwendig, welche wiederum für unterschiedliche Situationen variierende Eigenschaften aufweisen. Zusätzlich müssen sich diese mobilen Geräte automatisch finden, um bestimmte Aufgaben lösen zu können, wobei sich bei diesem Vorgang die Sicherheitsproblematik in den Vordergrund drängt. Auch spielt die Position der Nutzer oft eine wichtige Rolle. Ein Beispiel hierfür ist die Betrachtung von Urlaubsbildern, bei der ein Rechner mit einem in der Nähe befindlichen Bildschirm zum Anzeigen verwendet werden sollte.

Unter dem Thema „Allgegenwärtiges Rechnen“ sind in den vergangenen Jahren eine Reihe von Forschungsarbeiten entstanden, welche zum Teil auch erste Ansätze für die Entwicklung adaptiver Software liefern. Einige davon werden im Rahmen der verwandten Arbeiten in Kapitel 8 vorgestellt.

Der Begriff „*Pervasive Computing*“ wird oft synonym zu „*Ubiquitous Computing*“ gebraucht. Für die Allgegenwärtigkeit von Rechnern wird deshalb auch oft die Abkürzung „UbiComp“ verwendet.

2.2.2 Kontextsensitive Systeme

Kontextsensitive Systeme (*Context-aware computing*) wurden erstmals von [Schilit und Theimer 1994] im Jahre 1994 als „Anpassung an den Ort, in der Nähe befindliche Personen und Dinge sowie Änderungen dieser Objekte über die Zeit“ [Schilit und Theimer 1994] vorgestellt. Als erste Forschungsinitiative zur Kontextsensitivität gelten allerdings die *Olivetti Active Badges* [Want u. a. 1992] aus dem Jahre 1992. Für die Entwicklung adaptiver Software hat dieser Forschungsbereich wichtige Grundlagen für die Beschreibung, die Klassifizierung und Verarbeitung von Umgebungsinformationen geliefert. Der Begriff *„context-aware“* wird von [Abowd u. a. 1999] definiert durch: *“A system is context-aware if it uses context to provide relevant information and/or services to the user, where relevancy depends on the user's task.“*
Ein Kontext umfasst Informationen, welche die Situation eines Nutzers bzw. eines Objekts näher charakterisieren und für eine Anwendung von Bedeutung sind. Kontextinformationen lassen sich in unterschiedliche Kategorien einteilen. In der Literatur existieren verschiedene Kategorisierungsansätze [Abowd u. a. 1999]. Diese definieren Kontext als die sich ständig ändernde Ausführungsumgebung und enthalten dabei folgende Eigenschaften:

- *Computing environment:* verfügbare Rechner, verfügbare Geräte für Ein-/Ausgabe eines Nutzers, Netzwerkkapazität, Konnektivität oder Kosten
- *User environment:* Aufenthaltsort des Nutzers, in der Nähe befindliche Personen, die soziale Situation, Identität des Nutzers
- *Physical environment:* Lichtbedingungen, Umgebungslautstärke

Häufig wird die Zeit als zusätzliche Kontextkategorie eingeführt - vor allem um Kontexthistorien betrachten zu können. Zusätzlich wird der soziale Kontext (des Nutzers) oft weiter untergliedert. So können Informationen über die Stimmung eines Nutzers oder seine aktuelle Tätigkeit Inhalt des Kontexts sein. Der Aufenthaltsort eines Nutzers wird im untergeordneten Forschungsgebiet der ortsspezifischen Anwendungen (*location-based applications*) untersucht. Typischerweise können nicht alle Kontextinformationen direkt gemessen werden. Es werden primäre Kontexttypen wie z.B. Aufenthaltsort, Identität oder Zeit unterschieden von sekundären Kontexttypen, die direkt aus den primären abgeleitet werden können. So kann z.B. aus der Nutzeridentität die E-Mail-Adresse abgeleitet werden [Abowd u. a. 1999]. Kontextinformationen können, nach diesem Verfahren hierarchisch geordnet einer Anwendung bereitgestellt werden. Ausgehend von den Sensorrohdaten können komplexere Informationen daraus gewonnen werden. Hierfür können Kontextinformationsserver oder Verzeichnisdienste verwendet werden.
Im Rahmen der kontextsensitiven Systeme können verschiedene Anwendungsfälle für die Nutzung von Kontextinformationen gefunden werden. [Chen und Kotz 2000] kategorisieren kontextsensitive Anwendungen in aktive und passive Anwendungen. Aktive Anwendungen passen sich automatisch dem gemessenen Kontext an, während passive die Information darstellen oder für die spätere Nutzung speichern. [Abowd u. a. 1999] unterscheiden drei Kategorien von Anwendungsfähigkeiten bei der Verarbeitung von Kontextinformationen:

- Darstellung von Kontextinformation für den Nutzer

- Automatische Ausführung von Diensten
- Gewinnung von Informationen für spätere Abfragen

Generell kann unterschieden werden zwischen Kontextinformationen die anwendungsspezifisch deren Funktionalität verändern, und denen, die orthogonal zur eigentlichen Anwendungsfunktionalität genutzt werden, um adaptives Verhalten zu ermöglichen. In dieser Arbeit werden Techniken zur aktiven Anpassung von Anwendungen an Kontextinformationen untersucht, wobei hauptsächlich Eigenschaften der unterliegenden Infrastruktur in die Betrachtungen einbezogen werden. Forschungsergebnisse der kontextsensitiven Systeme können für die Bestimmung von Umgebungseigenschaften verwendet werden.

2.2.3 Autonomic Computing

Der Begriff des *Autonomic Computing* wurde maßgeblich durch IBM geprägt [Kephart und Chess 2003]. Die grundlegende Idee wurde im Jahr 2001 von Paul Horn an der Harvard University vorgestellt. Der Begriff *autonomic* ist an das autonome Nervensystem des Menschen angelehnt, das sich selbstständig um die grundlegenden lebenserhaltenden Körperfunktionen kümmert. *Autonomic Computing* beschäftigt sich im Wesentlichen mit der automatisierten Verwaltung von komplexen verteilten Softwaresystemen.
Um dieses Ziel umzusetzen, definiert das *Autonomic Computing* „Selbst-Eigenschaften". Zentraler Bestandteil ist dabei die Selbst-Verwaltung (*Self-Management*), welche weitere Eigenschaften, Selbst-Konfiguration (*Self-Configuration*), Selbst-Heilung (*Self-Healing*), Selbst-Schutz (*Self-Protection*) sowie Selbst-Organisation (*Self-Organisation*) umfasst. Die Selbst-Verwaltung befasst sich mit Anpassungs- und Wartungsarbeiten an Software, wenn sich deren Anforderungen, Arbeitsbelastung oder externe Eigenschaften verändern.
Selbst-Konfiguration beschäftigt sich mit Techniken zur Installation, Konfiguration und Integration großer komplexer Softwaresysteme. Autonome Systeme konfigurieren sich selbst in Übereinstimmung mit abstrakten Regeln, die z.B durch Geschäftsziele festgelegt werden. Bei der Integration neuer Komponenten werden diese entsprechend dem laufenden Gesamtsystem angepasst.
Selbst-Organisation untersucht Techniken zur Optimierung von Parametern, um eine optimale Leistungsfähigkeit zu erreichen. Dabei sollen die Systeme lernend entsprechende Einstellungen automatisiert vornehmen.
Selbst-Heilung beschäftigt sich mit der Lokalisierung und automatisierten Behebung von Fehlern bzw. Ausfällen in komplexen Softwaresystemen.
Selbst-Schutz beschreibt eine Eigenschaft komplexer Software, sich selbstständig gegen bösartige Attacken oder kaskadierende Fehler zu wehren. Dabei können z.B. Rückschlüsse aus vergangenen Ereignissen gezogen werden.
In [Kephart und Chess 2003] werden zahlreiche Herausforderungen für die Umsetzung der Vision des *Autonomic Computing* genannt, um die beschriebenen Eigenschaften umsetzen zu können. IBM hat mit dem *Autonomic Computing Tool-Kit* [Jacob u. a. 2004] eine Sammlung von Bibliotheken und Werkzeugen für die Analyse von *Log-/Trace*-Dateien (Untersuchung von Fehlerursachen), Ereignis- und Ressourcenmodellen bereitgestellt, die erste Teilproblematiken des *Autonomic Computing* lösen.

Im Rahmen der *Autonomic-Computing*-Initiative wurden von einigen großen IT-Unternehmen entsprechende Strategien vorgestellt, deren Inhalt im Folgenden kurz beschrieben werden soll. IBM bietet zusätzlich zu entsprechenden Softwarelösungen auch Hardwareunterstützung für die Umsetzung des *Autonomic Computing* an. Die aktuellen Systeme der *z-* und *p-Series* unterstützen die Partitionierung von Hardwareressourcen [Jann u. a. 2003]. Betriebssysteminstanzen werden in logischen Partitionen (*LPAR*) ausgeführt, denen Speicher-, CPU-, Netzwerk- und Festplattenressourcen zugeordnet werden können. Das Betriebssystem AIX unterstützt die Änderung von zugeteilten Ressourcen während der Laufzeit. Einer logischen Partition kann damit zusätzlicher Speicher und auch zusätzliche CPU-Leistung zugeordnet werden, wenn die in dieser Partition ausgeführte Software zusätzliche Ressourcen benötigt. Die Partitionierung lässt sich sehr feingranular verwalten, denn eine CPU kann beliebig zwischen mehreren logischen Partitionen aufgeteilt werden. Unter dem Motto des "*On Demand Business*" stellt IBM bereits einige Lösungen bereit, die es Firmen ermöglicht, auf wechselnde Geschäftsanforderungen zu reagieren.

SAP Adaptive Computing

SAP vermarktet unter dem Thema „*Adaptive Computing*" [KG 2007] die dynamische Ressourcenzuordnung für Unternehmenssoftware. In diesem Geschäftsbereich kommt es zu zyklisch auftretenden Fluktuationen bei der Nutzungsintensität der Software, die zum einen auf den Tageszeitunterschied zwischen Europa und Amerika zurückzuführen ist, zum anderen aber auch auf die Nutzung der Software an bestimmten Tagen des Monats z.B. Monats- bzw. Jahresabschluss. Um Kosten für die zu betreibende Hardware zu sparen, deren Dimension typischerweise für die Maximallast ausgelegt ist, werden Rechenzentren eingerichtet, die dynamisch zuschaltbare Ressourcen anbieten. Werden zusätzliche Ressourcen für die Ausführung der Unternehmenssoftware benötigt, können Administratoren neue zusätzliche Ressourcen aus dem Rechenzentrum hinzuschalten. Technologisch basiert das Hinzufügen neuer Rechner bei SAP auf einer zentralen Datenbank. Ein verteiltes Dateisystem ermöglicht das Einbinden (*Mounten*) und Starten entsprechender Systemkonfigurationen auf neuen Rechnern. Durch die Nutzung der zentralen Datenbanken wird die Problematik des Zustandstransfers gelöst, indem dynamisch erzeugte Systeme auf diese Daten zugreifen können. Die Migration laufender Systeme wird durch das Herunterfahren des alten und Neustart des Systems auf einem anderen Rechner implementiert [Gebhart 2006].

2.2.4 Organic Computing

Organic Computing [Schmeck 2004, 2005] ist eine deutsche Initiative zur Erforschung selbstorganisierender Systeme mit Anlehnung an Funktionen und Mechanismen aus der Biologie. Dabei werden Emergenzeffekte studiert, die durch die Programmierung von einfachen Protokollen unerwartetes bzw. auch gewünschtes Verhalten im Großen erzeugen. Ein Beispiel für Emergenzeffekte sind Ameisen, die im Vergleich zu ihrer eigenen Dimension sehr große Gebilde schaffen können, obwohl die einzelnen Individuen sehr einfachen Verhaltensmustern folgen. Im Rahmen des *Organic Computing* sollen Techniken entwickelt werden, um die Emergenz komplexer Softwaresysteme beherrschen zu können, die auch die Adaption einzelner Elemente im Rahmen der Selbstorganisation umfasst.

Nach der Vorstellung wichtiger Projekte im Bereich der adaptiven Software, sollen im Folgenden wichtige Grundlagen für die Ausführung komponentenbasierter verteilter Anwendungen beschrieben werden.

2.3 Komponentenplattformen für adaptive Anwendungen

Die Komponentenorientierung bildet eine wesentliche Grundlage dieser Arbeit. Die Komponententechnologie wurde dabei von drei Kräften geprägt. Darunter sind die *Common Object Request Broker Architecture (CORBA)*-Standards der Object Management Group (OMG), Microsofts *Component Object Model* (COM) und .NET sowie Suns Java-basierte Standards. Während sich schon 1992 mit Microsofts Visual Basic ein erster Markt für Komponenten (Visual Basic Extensions (VBX)) eröffnete, startete OMGs CORBA 2.0 1995 im Unternehmensbereich [Szyperski 1999, S. XXII]. Kurz darauf folgte Microsofts *Distributed* COM (DCOM) als Antwort auf CORBA sowie Java mit den JavaBeans. Ende 1999 folgte das heute weit verbreitete serverseitige Komponentenmodell der *Java Enterprise Beans*. Anfang 2002 wurde dann Microsofts .NET Plattform als Alternative zu Java veröffentlicht. Für ausführliche Informationen zu den vorgestellten Plattformen sei der Leser auf [Szyperski 1999] verwiesen.

2.3.1 OMG CORBA und CCM

Zunächst stand die OMG vor dem Problem der verteilten objektbasierten Kommunikation in heterogenen Systemen. Als Ergebnis wurde die CORBA entwickelt. Schnittstellen verteilter Objekte wurden mit der *Interface Definition Language* (IDL) beschrieben, was die Unabhängigkeit von Programmiersprachen, Hardware-Architekturen und Betriebssystemen garantierte. Um die Interoperabilität zwischen *Object Request Brokern* (ORB) verschiedener Hersteller zu ermöglichen, wurde mit CORBA 2.0 das *internet inter-ORB protocol (IIOP)* standardisiert. Für die dynamische Bindung zwischen Objekten wurde das *dynamic invocation interface (DII)* eingeführt. Mit dem *CORBA Component Model (CCM)* wurde dann später ein auf CORBA basierendes Komponentenmodell eingeführt. CCM definiert ein Binärformat für CCM-Komponenten, welche zusammen mit Metadaten in Form einer Deployment-Konfiguration (*deployment configuration*) in der sogenannten *CCM assembly* gespeichert sind. CCM-Komponenten sind charakterisiert durch Ports, die Verbindungspunkte für synchrone und ereignisbasierte Kommunikation beschreiben, und durch Attribute (Parameter), welche die Konfiguration des Verhaltens ermöglichen sowie die unterstützten Schnittstellen (*home interfaces*).

2.3.2 Sun Java und Enterprise JavaBeans

Java wurde als neue objektorientierte Programmiersprache Mitte der 90er-Jahre eingeführt. Vor allem die sandboxartige Ausführung von Java-Anwendungen in Form von *Applets*, die als ein erstes Komponentenmodell in Java angesehen werden [Szyperski 1999], ermöglichte die sichere Ausführung fremder Anwendungen in Web-Browsern. Die Plattformunabhängigkeit von Java durch die Verwendung einer virtuellen Maschine, die eine Zwischenrepräsentation - den Bytecode - ausführt, war ein Grund für den Siegeszug von Java. Der Einsatz von *just-in-time (jit) compilern* erlaubte die effiziente Ausführung von Java-Bytecode. Keine der von Java verwendeten Techniken war zum Zeitpunkt der Veröffentlichung wirklich neu, aber die Kombination einer

sicheren Sprache (das Fehlen von Zeigern, kontrollierte Ausführung, Verifikation), einer virtuellen Maschine und die Objektorientierung ermöglichten den Erfolg [Szyperski 1999, S. 261 f.]. Die Komponententechnologie hat Java durch die Einführung separater Schnittstellen (*interfaces*) auf Sprachebene geprägt, da mit diesem Konzept Schnittstellen von Softwarekomponenten beschrieben werden können. Mit den Schnittstellen ist eine wichtige Voraussetzung geschaffen, alternative Varianten einer Komponente zu erstellen, welche die gleiche Schnittstelle mit anderen nicht-funktionalen Eigenschaften implementieren.

Mit der Einführung der *JavaBeans*, welche die verbindungsorientierte Programmierung und Komposition als Komponentenmodell unterstützen, konnten Java-Komponenten mit Hilfe des JAR-Formats verpackt werden. JAR-Dateien enthalten die notwendigen Java-Klassen, Ressourcen und auch Hilfsdokumente für das Deployment und die Erzeugung von Objekten aus einer Komponente. Eine weitere Errungenschaft der *JavaBeans* ist die Unterstützung von Reflexion in Java. Dies erlaubt die Untersuchung von Strukturdetails (*introspection*), aber auch die Manipulation von Attributen sowie die dynamische Ausführung von Methoden während der Laufzeit. Reflexion hat sich als wichtige Basistechnologie für die Erstellung von Werkzeugen im Rahmen dieser Arbeit herauskristallisiert.

Mit den *Enterprise Java Beans (EJB)* wurde ein Container-basiertes, serverseitiges Komponentenmodell vorgestellt. Eine wichtige Neuerung der EBJs waren sogenannte Deployment-Deskriptoren, die beschreiben, wie Softwarekomponenten installiert und ausgeführt werden sollen. Ein Deployment-Deskriptor kann unter anderem Details über die Beschaffenheit des umgebenden Containers oder Zugriffsoptionen auf eine Datenbank festlegen.

2.3.3 Microsoft DCOM und .NET

Microsofts Weg zur Komponentenorientierung führte von den nicht-objektorientierten *Visual Basic for Applications* (VBA) über *Object Linking and Embedding* (OLE), *Component Object Model* (COM), *Distributed* COM (DCOM) bis hin zur .NET Plattform. Die ersten Schritte sollen dabei nicht weiter beschrieben werden, sie können wiederum in [Szyperski 1999, S. 329 f.] nachgelesen werden. Das *Component Object Model* (COM) ist ein Standard, der die Abbildung verschiedener Programmiersprachen auf ein Binärformat beschreibt. Als wesentliches Konzept wurden hierfür Schnittstellen eingeführt, deren Format von den virtuellen Methodentabellen (wie z.B. in C++ vorhanden) abgeleitet wurden. Mit DCOM wurde eine Kommunikations-Middleware auf der Basis von COM eingeführt, welche die Kommunikation von COM-Komponenten über Rechnergrenzen hinweg ermöglichte. Auch die Verarbeitung von Metadaten wurde mit den *type libraries* unterstützt.

Mit der .NET Plattform wurden erweiterbare Metadaten eingeführt. Mit Hilfe von Annotationen, die in der CLI als Attribute bezeichnet werden, können Strukturelemente (u.a. Typen, Felder, Methoden) näher beschrieben werden. Annotationen, die inzwischen auch in Java als *Annotations* eingeführt wurden, können von Werkzeugen und Programmen benutzt werden um Informationen über .NET-Komponenten zu verarbeiten. Ähnlich wie Java basiert .NET unter anderem auf einer virtuellen Ausführungsumgebung, der *Common Language Runtime* (CLR), unterstützt umfangreiche Klassenbibliotheken, eine automatischen Speicherverwaltung, Sicherheitsmechanismen (*Code Access Security*), Reflexion, Serialisierung und das dynamische Laden

von Komponenten. Einheiten des Deployments werden in .NET als Assemblies bezeichnet, die neben den Typen auch Metadaten über diese Typen enthalten. Ein wichtiges Konzept, das mit dem .NET Framework eingeführt wurde, ist die Versionierung von Assemblies. Im Manifest einer jeden Assembly sind Versionierungsinformationen gespeichert, welche die Koexistenz und somit die Verarbeitung multipler Versionen einer Assembly auf einem Rechner unterstützen. Das Typsystem, die Laufzeitumgebung CLR und die Sprache C# wurden unter anderem von der ECMA standardisiert ([Ecma International 2006a,b]). Die standardisierte Laufzeitumgebung wird als *Common Language Infrastructure (CLI)* bezeichnet. Neben der kommerziellen Implementierung von Microsoft, der *CLR*, existieren zahlreiche „quellcodeoffene“ Projekte, welche den Standard ECMA-335 implementieren. Unter ihnen ist das Mono-Projekt [Novell 2008] sowie Portable.NET [Bollow 2008]. Während die Microsoft CLR für die Windows-Betriebssysteme und eingeschränkt für Windows CE zur Verfügung steht, ermöglichen Mono und Portable.NET die Ausführung von CLI-Anwendungen auf MacOS X, Linux, FreeBSD, Solaris und anderen Betriebssystemen. Mit der *Shared Source Common Language Infrastructure (SSCLI)* [Microsoft Corporation 2008] steht der Quellcode auch für Microsofts Implementierung in Auszügen zur Verfügung.
Mit .NET Remoting enthält .NET eine Kommunikations-Middleware mit der Unterstützung von Objektfernaufrufen, die inzwischen von der *Windows Communication Foundation* (WCF) abgelöst wurde, welche dabei grundlegende Konzepte von .NET Remoting integriert. Vor allem die mit .NET eingeführten Annotationen, die in Java erst Jahre später Einzug hielten, und die mächtige Verarbeitung von Metadaten in .NET durch das eingeführte Assembly-Format sowie die Unterstützung von Versionierung auf Assembly-Ebene wiesen die CLI für diese Arbeit als ideale Komponentenplattform aus.
Einige der in der Arbeit vorgestellten Techniken wurden auf Basis der .NET Plattform implementiert. Dabei wurde darauf geachtet, bei der Implementierung CLI-konforme Anwendungen zu entwickeln. Im weiteren Verlauf der Arbeit wird deshalb die .NET Plattform allgemein als CLI bezeichnet, sofern die entwickelten Anwendungen nicht explizit nicht CLI-konforme Elemente enthalten, wie dies z.B. bei graphischen Nutzerschnittstellen der Fall ist, die nicht standardisiert wurden.

2.3.4 Komponentenplattformen für mobile Geräte

Auch im Umfeld der mobilen und eingebetteten IT-Systeme hielt die Komponententechnologie Einzug. Mit der *Java 2 Micro Edition (J2ME)* bzw. JavaFX steht für Java eine eingeschränkte Version für mobile Geräte mit limitierten Ressourcen zur Verfügung. J2ME definiert Profile, welche die enthaltene Funktionalität für verschiedene Geräteklassen einschränken. Auch für Microsofts .NET Plattform steht mit dem *.NET Compact Framework* eine Plattform mit reduziertem Funktionsumfang für Geräte mit limitierten Ressourcen zur Verfügung. Für Geräte mit stark eingeschränktem Speicherumfang wurde im Fachgebiet für Betriebssysteme und Middleware am Hasso-Plattner-Institut die Lego.Net-Compiler-Erweiterung geschaffen, welche die Ausführung von .NET-Programmen z.B. auf den Lego Mindstorm Robotern ermöglicht. Die vom Autor dieser Arbeit entwickelte Erweiterung *Real-Time.Net* [von Löwis und Rasche 2006] erlaubt die vorhersagbare Ausführung von CLI-Programmen mit Echtzeitanforderungen in eingebetteten Systemen.
In den wesentlichen Grundzügen stehen die in dieser Arbeit identifizierten Basistechnologien auch in den limitierten Komponentenplattformen für mobile Geräte zur

Verfügung, was bei der anschließenden Vorstellung dargelegt wird. Dies erlaubt den Einsatz der entwickelten Techniken auch in diesem Umfeld.

2.4 Basistechnologien für adaptive Anwendungen

Das Ziel dieser Arbeit war es, eine Ausführungsumgebung sowie Werkzeuge für die Entwicklung, adaptiver Anwendungen auf der Basis moderner Komponentenplattformen zu entwickeln. Dabei wurde eine Reihe von Basistechnologien identifiziert, die allgemein die Ausführung adaptiver Anwendungen ermöglichen.

2.4.1 Komponentenorientierung

Die Komponentenorientierung wurde schon in der Einleitung als Schlüsseltechnologie identifiziert, die neue Möglichkeiten bei der Entwicklung adaptiver Anwendungen bietet. Das Grundkonzept der Komponentenorientierung alternative, austauschbare Bausteine bereitzustellen, ermöglicht die Auswahl von Alternativen für verschiedene Umgebungssituationen einer Anwendung. Ein wichtiges Konzept, das im Rahmen dieser Arbeit verwendet wurde, ist die Versionierung von Softwarekomponenten. Diese erlaubt die Installation alternativer Algorithmen in Komponenten unterschiedlicher Versionen. Zusätzlich bilden Komponenten die Grundlage des Deployments. Dieses Konzept wurde in dieser Arbeit für die entfernte Erzeugung von Komponenten zur Laufzeit verwendet.

2.4.2 Metaprogrammierung und Reflexion

Die Metaprogrammierung bzw. Reflexion (*computational reflection*) [Maes 1987; Smith 1982] beschreibt die Fähigkeit von Programmen, sich selbst zu analysieren oder ihr eigenes Verhalten zu verändern [Sadjadi und McKinley 2003]. Die Metaprogrammierung stellt Schnittstellen auf abstrahierte Objekte einer Implementierung, in der Basisebene, bereit, die durch Metaobjekte innerhalb einer Metaebene verarbeitet und verändert werden können. Die Verbindung zwischen Basis- und Metaebene wird durch ein Metaobjektprotokoll (MOP) realisiert. Daten über Objekte der Basisebene werden als Metadaten bezeichnet.
Bei der Metaprogrammierung wird zwischen der Struktur- und Verhaltensreflexion unterschieden. Während die Strukturreflexion die Analyse und Manipulation von Strukturdetails, also dem internen Aufbau einer Anwendung umfasst, beschäftigt sich die Verhaltensreflexion mit der Manipulation des Verhaltens von Programmen, was z.B. durch das Abfangen von Methodenaufrufen der Basisebene und der Umlenkung in eine Metaebene realisiert werden kann.
Zusätzlich wird zwischen *Introspection* und *Intercession* unterschieden. Ersteres umfasst den reinen Lesezugriff auf Strukturinformationen. *Introspection* wird von der CLI und der Java-Plattform unterstützt und bietet dort unter anderem Lesezugriff sowohl auf Assembly-, Typ-, Vererbungs-, Methoden- und Feldinformationen als auch auf Typ-Annotationen. Im Gegensatz dazu wird unter *Intercession* die Manipulation von Metadaten verstanden, die in Java und der CLI nicht bzw. nur eingeschränkt unterstützt werden. Die Manipulation von Metadaten ist z.B. in Smalltalk [Goldberg und Robson 1983] möglich, wo auch neue Methoden während der Laufzeit einer existierenden Klasse hinzugefügt werden können. In der CLI ist es nur möglich mit Hilfe

der *Reflection.Emit*-Schnittstelle neue Typen während der Laufzeit zu erzeugen und zu benutzen.
Für die Trennung von funktionalen und nichtfunktionalen Belangen spielt die Verwendung von Reflexion eine wichtige Rolle. So kann das funktionale Verhalten einer Basisebene, abgekoppelt von einer Metaebene aus manipuliert werden, um z.B. Anpassungen an wechselnde Umgebungseigenschaften vorzunehmen. In dieser Arbeit wird Reflexion zum einen für die Verarbeitung von Metadaten inklusive nutzerdefinierter Annotationen zur Erstellung von Werkzeugen verwendet. Zum anderen wird Reflexion zur Laufzeit eingesetzt, um von einer orthogonalen Ebene Verbindungspunkte und Parameter auf der funktionalen Anwendungsebene zu konfigurieren. Reflexion wird demnach für die Entwicklung adaptiver Anwendung als Basistechnologie benötigt und wird von den vorgestellten Komponentenplattformen unterstützt. Auch das *.NET Compact Framework* bietet die benötigten *Introspection*-Mechanismen. Nur die Funktionalität der *Reflection.Emit*-Schnittstelle steht hier nicht zur Verfügung. Da die Werkzeuge nicht auf den mobilen Geräten ausgeführt werden müssen, ist diese Einschränkung unerheblich.

2.4.3 Dynamisches Laden von Softwarekomponenten

Um die Adaption von Anwendung während der Laufzeit realisieren zu können, müssen vor allem im Falle unvorhersehbarer Änderungen Möglichkeiten zum Laden neuer Logik existieren. Auf die Komponententechnologie übertragen, wird das dynamische Nachladen von Komponenten benötigt.
Die *Java Standard Edition* stellt das Konzept von *Class-Loadern* bereit. Diese können von Anwendungsentwicklern erstellt werden und zum dynamischen Laden von Klassen während der Laufzeit eingesetzt werden. Mit den *Class-Loadern* können u.a. Java-Klassen von entfernten Rechnern nachgeladen und dann Exemplare von den dynamisch geladenen Klassen erzeugt werden. Jede Klasse wird während der Laufzeit eindeutig durch ihren Namen und den *Class-Loader* identifiziert. Dies ermöglicht im Prinzip die Koexistenz einer Klasse in mehreren Versionen, indem für jede Version ein separates *Class-Loader*-Exemplar verwendet wird.
Die CLI bietet die Möglichkeit, Assemblies zur Laufzeit zu laden und Exemplare der enthaltenen Typen zu erzeugen. Die CLI-Klassenbibliothek bietet verschiedene Methoden zum Laden von Assemblies an. Die Klasse *System.Reflection.Assembly* stellt diese Funktionen als statische Methoden bereit. Die verschiedenen Methoden unterscheiden sich beim Auflösen referenzierter Assemblies und bei der Zuordnung zu einem Lade-Kontext. Geladene Assemblies werden durch ihren Lade-Kontext eindeutig identifiziert, weshalb bei der Sicherstellung referenzieller Integrität bei der dynamischen Aktualisierung von Komponenten der Lade-Kontext betrachtet werden muss. Prinzipiell wird zwischen dem System-Kontext für Assemblies, die durch statische Referenzen bei der initialen Programmausführung geladen wurden, und mehreren Lade-Kontexten für dynamisch geladene Assemblies unterschieden.
Beim dynamischen Laden von Assemblies müssen auch abhängige Assemblies geladen werden. Hierfür müssen an Hand der Metadaten im Assembly-Manifest mit Hilfe der Assembly-Namen die entsprechenden Dateien der abhängigen Assemblies gefunden werden. Dieser Vorgang wird auch als Auflösen (*resolution*) bezeichnet. Die CLR hat dafür einen eingebauten Mechanismus, der durch die *Fusion-Engine* implementiert wird. Dieser kann abhängige Assemblies im lokalen Dateisystem oder auf Netzwerkressourcen finden. Ein in dieser Arbeit verwendeter Mechanismus ist die Möglichkeit,

Assemblies in der Form binärer Felder (*byte[]*) direkt aus dem Hauptspeicher zu laden. Dabei wurde beachtet, dass die Auflösung von Assemblies eigenhändig implementiert werden muss. Hervorzuheben ist, dass im Fall des entfernten Ladens von Komponenten ein sehr leistungsfähiger Mechanismus entwickelt werden konnte, indem Abhängigkeiten von Assemblies systematisch analysiert, an den entfernten Rechner übertragen wurden und dort direkt aus dem Hauptspeicher geladen werden können. Zusätzlich lassen sich durch die Implementierung eigener Auflösungsalgorithmen effektive Cache-Verfahren für adaptive Anwendungen entwickeln. Das *.NET Compact Framework* unterstützt das dynamische Laden von Komponenten direkt aus dem Dateisystem. Das direkte Laden aus dem Hauptspeicher oder von Netzressourcen wird aber nicht unterstützt.

2.4.4 Aspektorientierte Programmierung

Die aspektorientierte Programmierung [Kiczales u. a. 1997] beschreibt Techniken, um nicht-funktionale Aspekte, die über die Anwendungslogik verteilt sind (*cross-cutting concerns*), separiert zu implementieren. Ein Aspektweber ist in der Lage, die getrennte Implementierung zu einer ausführbaren Anwendung zu vereinen, indem er Codefragmente (*Advices*) in die Anwendungslogik integriert. Ein *Pointcut* definiert aus einer Menge möglicher Verwebungspunkte die Integrationspunkte. *Advices* bestimmen, ob Aspektcode vor, nach oder anstatt eines Methodenaufrufs ausgeführt wird. Über *Introductions* ist es möglich, die Implementierung neuer Schnittstellen in die Anwendungslogik zu integrieren.

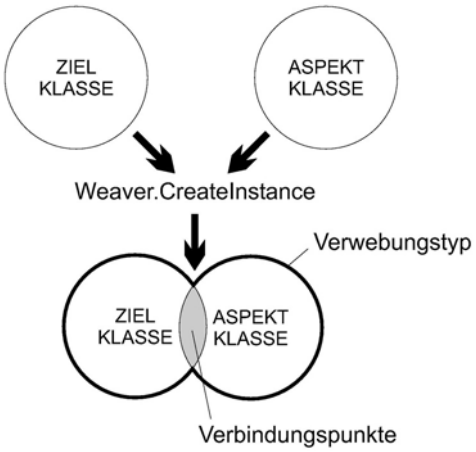


Abbildung 2.1: Aspektorientierte Programmierung [Schult und Polze 2002]

Je nach Verwebungszeitpunkt im Softwarelebenszyklus können Aspektweber in unterschiedliche Kategorien eingeteilt werden. Zunächst werden statische und dynamische Aspektweber unterschieden. Statische Aspektweber operieren vor der Laufzeit der Anwendung, dynamische zur Anwendungslaufzeit. Die dynamischen Aspektweber können weiterhin in Ladezeit- und Laufzeitaspektweber klassifiziert werden. Aspektweber, die auf Basis des Quellcodes arbeiten, werden als Quellcodeaspektweber bezeichnet, während Aspektweber, die auf der Basis von Bytecode- bzw. IL-Code-Instruktionen arbeiten, werden als Bytecode-Aspektweber bezeichnet.
Der erste verfügbare Aspektweber war *AspectJ* [Kiczales u. a. 2001], welcher am *Xerox Parc* entwickelt wurde. *AspectJ* erweitert die Programmiersprache Java um Konzep-

te der aspektorientierten Programmierung. *AspectJ* definiert eine eigene Sprache, die die Implementierung von Aspektcode und die Konfiguration entsprechender Verwebungsinformationen ermöglicht. Im Rahmen des Loom.Net Projekts [The Loom.NET Project 2005] wurden für die CLI von Wolfgang Schult, einem Kollegen des Autors, ein statischer und ein dynamischer Aspektweber entwickelt, die im Folgenden kurz vorgestellt werden sollen, da sie im Rahmen der vorliegenden Arbeit eingesetzt wurden.

Loom.Net - Ein Aspektweber für die CLI

Loom.Net ist ein statischer Bytecodeweber, der als Eingabedaten IL-Code-Instruktionen verarbeitet und damit direkt auf den Assemblies der CLI operiert. Er wird im weiteren Text als statischer Aspektweber bezeichnet. Er arbeitet auf einer *Template*-basierten Aspektbeschreibung, die durch Textersetzung mit der Anwendungslogik verknüpft wird. Dabei wird für die zu verwebenden Zielklassen eine Ableitung generiert, die zusätzliche Logik enthält. Die abgeleitete Klasse wird durch Nutzung von Reflektionsinformationen der funktionalen Klasse und der Aspektbeschreibung als C#-Quelle erzeugt. Durch die Unterstützung der Assembly-übergreifenden Vererbung von Klassen in der CLI ist es der erzeugten Klasse möglich, von der funktionalen Klasse zu erben, obwohl sie in einer separaten Assembly gespeichert ist.

Rapier-Loom.Net

Der ebenfalls von Wolfgang Schult entwickelte Aspektweber Rapier-Loom.Net [Schult und Polze 2002] ist ein dynamischer Aspektweber. Rapier-Loom.Net generiert bei der Objekterzeugung mit Hilfe der *Reflection.Emit*-Schnittstelle dynamische Proxy-Klassen, welche den Aspektcode in die Funktionalität der Klassen integrieren. Dieser Proxy-Code wird in Form einer Ableitungskette an die funktionale Klasse gebunden. Für jeden Aspekt wird eine neue Klasse erzeugt die, von der zuvor erzeugten Klasse abgeleitet ist. Aspekte sind Objekte, die von der Klasse *Aspect* abgeleitet sind. Im Aspektcode kann das verwobene Zielobjekt mit Hilfe einer Kontextvariablen (*Context.Instance*) manipuliert werden. Diese stellt den Zugriff auf Methodenamen, das Zielobjekt sowie Methoden zum Weiterleiten eines Methodenaufrufs an das Zielobjekt im Falle einer *Instead*-Verwebung bereit. Eine Referenz auf das jeweilige Aspektobjekt ist im Attribut *_aspect_* in jeder Ableitungsstufe im verwobenen Zielobjekt gespeichert. Die Konfiguration von Aspekten im funktionalen Code kann zum einen durch die Verwendung der Rapier-Loom.Net Fabrikmethode *Weaver.CreateInstance*, der als Parameter der Typ der Zielklasse sowie die Referenzen auf die Aspektobjekte übergeben werden, realisiert werden. Zum anderen steht eine deklarative Konfigurationsmöglichkeit mit Hilfe von Metadaten-Annotationen zur Verfügung.

```
[Trace]
class Calculator{
  public int Add(int[] numbers);
}
```

Abbildung 2.2: Deklarative Konfiguration von Aspekten

Abbildung 2.2 zeigt die deklarative Aspektkonfiguration an einem Beispiel. Durch die *Trace*-Annotation wird angezeigt, dass alle Methoden der *Calculator*-Klasse mit der

Funktionalität des *Trace*-Aspekts verwoben werden sollen. Durch die Objekterzeugung über die Fabrikmethode *Weaver.CreateInstance* wird die *Trace*-Logik in das erzeugte Objekt integriert.
Im Folgenden werden abschließend wichtige Konzepte der Laufzeitumgebung von Komponentenplattformen vorgestellt. An dieser Stelle soll noch kurz auf die Verwendung des Begriffs **Attribut** eingegangen werden, der oft mehrdeutig verwendet wird. Im Rahmen dieser Arbeit werden die Datenelemente eines Objekts als Attribute bezeichnet. Die Attribute der CLI, die Annotationen von Metadaten realisieren, werden als CLI-Attribute bezeichnet. Auch im *CORBA Component Model* werden Parameter von Komponenten als Attribute bezeichnet. Im Rahmen dieser Arbeit wird hierfür der Begriff Parameter verwendet.

2.5 Virtuelle Laufzeitumgebungen

Die CLI und auch die Java-Plattform umfassen eine typsichere objektorientierte Laufzeitumgebung. Diese unterstützt die grundlegenden Konzepte der Objektorientierung: Kapselung, Vererbung und Polymorphie. Klassen definieren das Verhalten von Objekten, die Dinge der wirklichen Welt repräsentieren. Objekte werden durch die Exemplarbildung von Klassen oder Objektprototypen erzeugt, die in Softwarekomponenten gespeichert sind. Der Zustand eines Objekts wird durch eine eindeutige Identität sowie die Werte aller Attribute eindeutig bestimmt. Die Attribute können letztendlich skalare Daten oder Referenzen auf andere Objekte enthalten. Daten werden in der CLI durch ein Typsystem (*Common Type System*) beschrieben. Typen sind Prädikate, die Verhalten und Repräsentation der CLI-Entitäten bestimmen. In der CLI werden Wert- und Referenztypen unterschieden, wobei erstere als Strukturen (*struct*) und letztere als Klassen (*class*) bzw. Schnittstellen (*interface*) deklariert werden. Referenzen sind in der CLI typisiert, das heißt sie können immer nur auf kompatible Typen verweisen - nicht typisierte Zeiger (z.B.: *void** in C++) werden nicht unterstützt, wodurch die Stabilität der Anwendungen erhöht wird. Werttypen sind zum einen elementare eingebaute Typen (*string*, *int*, *double* usw.), zum anderen können sie nutzerdefiniert sein. Es existiert ein Mechanismus in der CLI, Werttypen in Referenztypen zu überführen. Dieser funktioniert, indem von der Laufzeitumgebung eine Objekthülle erzeugt wird, welche die Referenzsemantik auf die Werte realisieren. Dieses Verfahren wird als *Boxing* bezeichnet. Im Sinne einer abstrakten Beschreibung der Laufzeitumgebung sollen im Rahmen dieser Arbeit alle Laufzeitentitäten der CLI als Objekte betrachtet werden. Alle Exemplare von Werttypen werden als Objekte mit Attributen verstanden. Besitzt ein Objekt einen Werttyp als Attribut, wird dieses Attribut als Referenz auf ein weiteres Objekt mit den Werttypattributen als Attribute betrachtet. Mit dieser Festlegung können die Laufzeitumgebung der Java-Plattform und der CLI als äquivalent betrachtet werden.
Das Konzept des *Boxing* soll noch einmal kurz an Hand eines kleinen Beispiels demonstriert werden. Der Beispielcode (dargestellt in Abbildung 2.3) enthält die Definition des komplexen Werttyps *Point* sowie den Referenztype *Location*, der unter anderem ein Attribut vom Typ *Point* besitzt. Abbildung 2.4 zeigt auf der linken Seite das normale Objektlayout für ein Exemplar des Typs *Location*. Der Werttyp *Point* ist direkt im Speicherlayout des Typs *Location* enthalten. Betrachtet man nun das Attribut *point* als Referenz auf das Exemplar von *Point* als Objekt, wird in *Location* aus dem Werttyp eine Referenz auf eine Objekt-*Box* namens *PointBox*. Dies ist auf der rechten Seite von

```
struct Point                          class Location
{                                     {
  int x;                                  Point point;
  int y;                                  string name;
  int z;                                  int ID;
}                                     }
```

Abbildung 2.3: *Boxing*: Beispielcode

Abbildung 2.4 dargestellt.

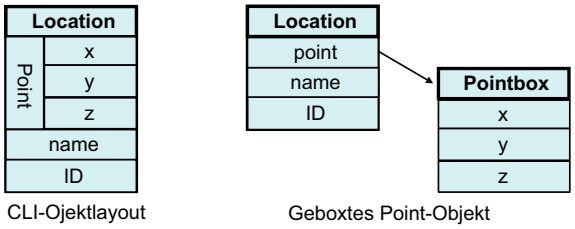


Abbildung 2.4: Objektlayout und Boxing

Letztendlich wird in der CLI zwischen statischen Methoden bzw. Attributen, die allen Exemplaren eines Typs gemein sind und Exemplarmethoden bzw. Exemplarattributen unterschieden, die einzigartig pro Objekt bzw. Wert existieren.

2.6 Zusammenfassung

Im Rahmen dieses Kapitels wurde zunächst die grundlegende Terminologie der Arbeit eingeführt. Dem gefolgt wurden die historische Entwicklung adaptiver Software und grundlegende Projekte in diesem Themenbereich vorgestellt. Anschließend wurden mit der Beschreibung von wichtigen Technologien und Vertretern von Komponentenplattformen, der Metaprogrammierung und der aspektorientierten Programmierung wichtige Grundlagen für den in der Arbeit diskutierten Ansatz für die Entwicklung adaptiver Anwendungen vorgestellt. Die Diskussion der Konzepte der objektorientierten Laufzeitumgebungen der Java- bzw. .NET-Plattform bildeten den Abschluss dieses Kapitels.

3 Dynamische Rekonfiguration

Im Rahmen dieses Kapitels soll die Realisierung adaptiven Verhaltens durch die dynamische Rekonfiguration verteilter komponentenbasierter Anwendungen aufgezeigt werden. Moderne Komponentenplattformen wie die .NET und die Java-Plattform enthalten eine objektorientierte, virtuelle Laufzeitumgebung, die für die Ausführung der Anwendungen verwendet wird. Grundlegende Konzepte dieser Laufzeitumgebungen sollen zunächst in diesem Kapitel modelliert werden.
Ausgehend von dem Modell der grundlegenden Konzepte, wird in diesem Kapitel anschließend ein Modell verteilter komponentenbasierter Anwendungen vorgestellt. Um die durch die Komponenten geschaffene Abgrenzung einzelner Anwendungsbausteine während des Deployment-Zeitraums auf die Laufzeit zu übertragen, wird das Konzept der Kapseln eingeführt. Kapseln umfassen eine Menge von Objekten und grenzen Anwendungsteile zur Laufzeit ab, die über wohldefinierte Schnittstellen kommunizieren. Kapseln ermöglichen die dynamische Rekonfiguration, ohne die gesamte Anwendung unterbrechen zu müssen. Um die Verletzung der Konsistenz der Anwendungsdaten während einer dynamischen Rekonfiguration zu vermeiden, müssen Anwendungs- und Rekonfigurationsaktivitäten synchronisiert werden. Auf der Basis des beschriebenen Modells werden die vom Autor entwickelten Synchronisationsalgorithmen vorgestellt und die Evaluierung durch den Einsatz formaler Methoden dargelegt.
Im zweiten Teil dieses Kapitels wird die Implementierung der vorgestellten Algorithmen im Rahmen des Adapt.Net-Projekts beschrieben. Zunächst wird hierfür die XML-basierte Spezifikation von Anwendungskonfigurationen erörtert. Der Beschreibung folgt die Vorstellung wichtiger Infrastrukturkomponenten für die Realisierung adaptiver Anwendungen. Einen Kern der Arbeit bildet der Einsatz der aspektorientierten Programmierung für die wieder verwendbare Generierung rekonfigurationsspezifischer Belange, die im Rahmen eines neuen Programmiermodells für adaptive Anwendungen vorgestellt wird. Abschließend wird die Integration heterogener Komponenten in die Anwendungskonfigurationen dargestellt.
Teile dieses Kapitels wurden in [Rasche 2006; Rasche und Polze 2002, 2003, 2008; Rasche u. a. 2005a; Rasche und Schult 2007; Rasche u. a. 2005c] veröffentlicht. Teilaspekte beim Einsatz der aspektorientierten Programmierung sowie der Verwendung von *ReaderWriterLocks* sind in Zusammenarbeit mit Wolfgang Schult entstanden.

3.1 Ein Modell verteilter, komponentenbasierter Anwendungen

Zunächst sollen einige grundlegende Konzepte der Laufzeitumgebungen moderner Komponentenplattformen beschrieben werden. Die Beschreibung erfolgt an dieser Stelle nicht vollständig, sondern ist auf das Notwendige konzentriert.
Ausgehend von den betrachteten Laufzeitumgebungen bestehen Anwendungen während der Laufzeit aus einer Menge von Objekten, deren Verhalten und Struktur der Daten durch eine Menge von Typen definiert wird. Ein Typ definiert unter anderem eine Menge von Methoden und Attributen. Methoden beinhalten das Verhalten, während

Attribute die Daten darstellen. Diese sollen selbst nicht formalisiert, aber verwendet werden. Eine Methode ließe sich prinzipiell als Tripel von Namen und Rückgabetyp sowie einer Menge von Parametertypen und -namen beschreiben. Ein Attribut ist durch eine Zeichenkette Name sowie eine Typinformation (z.B. in Form einer Zeichenkette) beschrieben.

Definition 3.1.1 (Typ) *Ein Typ T wird definiert durch das Tripel T(Name, Methoden, Attribute) mit:*

- *Einer Zeichenkette **Name** der Form [Namensraum.]*Name: Der Name des Typs.*
- *Einer Menge von **Methoden**: Definiert mögliche Operationen auf den Daten.*
- *Einer Menge von **Attributen**: Beschreibt die Struktur der Daten.*

Für einen Typ sind die folgenden Projektionen definiert:

- *tname: $T \rightarrow$ Zeichenkette - der Name des Typs*
- *attributes: $T \rightarrow$ Attribute - die Menge der Attribute eines Typs*
- *methods: $T \rightarrow$ Methoden - die Menge der Methoden eines Typs*

Es können eine Reihe unterschiedlicher Ausprägungen von Typen unterschieden werden. Referenztypen können weiter in **Klassen**, **Schnittstellen** und **Felder** (*arrays*) klassifiziert werden. Typen, die fest in die virtuelle Laufzeitumgebung integriert sind, werden als eingebaute Typen bezeichnet. Skalare Werttypen werden auch als primitive Typen bezeichnet. Für Schnittstellentypen T_I gilt: $attributes(T_I) = \emptyset$. Attribute können entweder Referenzen auf andere Objekte oder primitive Daten enthalten.

Zwischen Typen können Abhängigkeiten bestehen, die sich unter anderem durch komplexe Typen ergeben, die aus anderen Typen zusammengesetzt sein können. Mögliche Abhängigkeiten zwischen Typen sollen im Folgenden formalisiert werden und die Abhängigkeit zwischen Typen durch die „Direkte Abhängigkeits-Relation“: $\triangleright$ beschrieben werden.

Definition 3.1.2 (Direkte Typabhängigkeit) *Ein Typ t hängt direkt von einem zweiten Typ s ab ($t \triangleright s$) gdw:*

- *t und s sind Klassen oder Schnittstellen und t erbt von s oder*
- *t ist eine Klasse und implementiert die Schnittstelle s oder*
- *t hat ein Attribut mit dem Typ s oder*
- *t ist ein Feldtyp und besitzt den Basistyp s oder*
- *t enthält eine Methode mit einem (Rückgabe-)Parameter vom Typ s oder*
- *eine Methode von t erzeugt ein Exemplar von s (new-Operator) oder*
- *eine Methode von t enthält eine statische Typumwandlung auf s (cast).*

3.1.1 Die Einheiten des Deployment - Assemblies

Nach [Szyperski 1999] sind Softwarekomponenten die Einheiten des Deployment. Da der Begriff Komponente oft mehrdeutig verwendet wird, soll an dieser Stelle explizit zwischen Komponenten zur Deployment- und Laufzeit durch die Verwendung verschiedener Begriffe unterschieden werden. Komponenten während der Deploymentzeit werden als **Assemblies**, Komponenten während der Laufzeit als Kapseln bezeichnet. Assemblies enthalten Typen, die ihre Funktionalität über wohldefinierte Schnittstellen zur Verfügung stellen. Bei der Erstellung einer Anwendung kann ein Entwickler Assemblies auswählen und durch die Implementierung einer geringen Menge an Verknüpfungslogik eine Anwendung erzeugen. Der Begriff der Assembly wurde der CLI entnommen.
Eine Assembly besteht mindestens aus einer Datei, kann aber auch über mehrere Dateien verteilt sein. Jede Assembly enthält ein Manifest, eine Tabelle mit Informationen über ihren Inhalt. Im Manifest wird ein eindeutiger Assembly-Name festgelegt, der sich unter anderem aus einer Zeichenkette und einer Versionsnummer zusammensetzt[1]. In der Java-Plattform sind JAR-Dateien die Einheiten des Deployment, die auch ein Manifest mit zusätzlichen Meta-Daten enthalten können. Die Versionierung wird in der Java-Plattform nicht direkt unterstützt, kann aber mit Hilfe verschiedener Techniken ergänzt werden. Eine Variante wurde in einer vom Autor co-betreuten Masterarbeit [Leidner 2006] realisiert. Als abstraktes Konzept soll zunächst der Assembly-Name eingeführt werden:

Definition 3.1.3 (Assembly-Name) *Ein Assembly-Name ist ein Paar AN(name, version) mit:*

- *einer beliebigen Zeichenkette* ***name*** *und*
- *einer Zeichenkette* ***version****.*

In jedem Manifest ist eine Tabelle der in der Assembly gespeicherten Typen enthalten. Im Manifest werden zusätzlich Abhängigkeiten zu anderen Assemblies definiert, welche auf Grund von Abhängigkeiten mit Typen aus diesen Assemblies impliziert werden. Eine Assembly A_1 hängt von einer zweiten Assembly A_2 ab genau dann wenn gilt: $\exists s \in types(A_1), t \in types(A_2)$ mit $s \triangleright t$. Die Abhängigkeit zwischen Assemblies muss explizit bei der Erstellung (Linken) einer Assembly durch den Entwickler angegeben werden. Abhängigkeiten zwischen Assemblies werden im Manifest durch Referenzen auf einen Assembly-Namen gespeichert. Nun sollen Assemblies selbst definiert werden:

Definition 3.1.4 (Assembly) *Sei T eine Menge von Typen und AN eine Menge von Assembly-Namen. Eine Assembly A ist definiert durch die Menge $A(name, \mathcal{P}(T), \mathcal{P}(AN))$ mit:*

- *Einem Assembly-Namen $name \in AN$*
- *Einer Menge von definierten Typen $\mathcal{P}(T)$*
- *Einer Menge Assembly-Namen referenzierter Assemblies $\mathcal{P}(AN)$*

Zusätzlich sind folgende Projektionen auf der Menge A definiert:

[1] In der CLI enthält der Name einer Assembly zusätzlich einen *Culture*-Wert, welcher den Sprachraum bestimmt, sowie einen Signatur-Hash, welche im Modell nicht betrachtet werden, weil sie für die dynamische Rekonfiguration nicht relevant sind.

- *aname:* $A \rightarrow AN$ *- der Name der Assembly*
- *types:* $A \rightarrow \mathcal{P}(T)$ *- die Menge aller Typen einer Assembly*
- *refs:* $A \rightarrow \mathcal{P}(AN)$ *- die Menge der Namen aller referenzierten Assemblies*

Ein Typ ist nicht immer eindeutig durch seinen Namen identifiziert, da er in Assemblies mit unterschiedlicher Versionsnummer mit gleichem Namen vorhanden sein kann. Deshalb werden Typen immer durch die Kombination ihres Namens und einem zugehörigen Assembly-Namen eindeutig identifiziert.
Bei der Erzeugung eines Exemplars eines Typs müssen die definierende Assembly sowie alle abhängigen Assemblies aufgelöst und geladen werden. In diesem Zusammenhang soll die Relation **Abhängigkeitshülle (HUL)** einer Assembly definiert werden, welche die Menge aller abhängigen Assemblies inklusive derer Abhängigkeiten zusammenfasst. Diese wird zur entfernten Exemplarerzeugung von Kapseln benötigt, da potentiell alle Assemblies und deren Abhängigkeiten auf einen entfernten Rechner übertragen werden müssen.

Definition 3.1.5 (Abhängigkeitshülle) *Sei A die Menge aller Assemblies dann ist HUL definiert durch* $HUL : A \rightarrow \mathcal{P}(A)$ *mit:*

- $A_2 \in HUL(A_1)$ *gdw.* $A_1 = A_2$
- $A_2 \in HUL(A_1)$ *gdw.* $\exists A_3 \in HUL(A_1)$ *und* $aname(A_2) \in refs(A_3)$

für alle $A_1, A_2, A_3 \in A$.

Nachdem nun grundlegende Konzepte moderner Komponentenplattformen beschrieben wurden, sollen im Folgenden die zentralen Konzepte der Komponente zur Laufzeit sowie der Anwendungskonfiguration vorgestellt werden.

3.1.2 Die Komponenten zur Laufzeit - Kapseln

Eine Komponente ist eine Abstraktionseinheit, die einen Teil einer Anwendung logisch zusammenfasst. Wie schon zuvor beschrieben ist eine Komponente in der Java- und .NET-Plattform zunächst eine Einheit des Deployment, welche eine Abstraktionseinheit für Software vor ihrer Ausführung darstellt. In dieser Arbeit soll diese Abstraktion auch während der Laufzeit betrachtet werden und wird in diesem Rahmen als **Kapsel** bezeichnet. Das Konzept der Kapsel wird eingeführt, um die Austauschbarkeit von Anwendungsbausteinen während der Laufzeit zu ermöglichen, wie dies während des Deployment-Zeitraums auf Basis der Assemblies gegeben ist. Eine Kapsel trennt dabei einen Teil einer Anwendung logisch ab und dient bei der dynamischen Rekonfiguration als Grundeinheit des Austauschs.
Urspünglich wurde der Begriff der Kapsel erstmals mit dem *Open Distributed Processing* (ODP) Referenzmodell eingeführt und beschreibt dort einen Verwaltungscontainer für konfigurierbare Objekte [Raymond 1995]. Eine zu dieser Arbeit ähnliche Bedeutung besitzt der Kapselbegriff in der Echzeiterweiterung von UML: *Real-time UML* [Selic 1998], in der eine Kapsel eine Menge von Objekten mit Hilfe von Port-Objekten abgrenzt. Im Rahmen dieser Arbeit umfasst eine Kapsel eine Menge von Objekten, die zu ihrer Umwelt durch abgeschlossene Referenzierung abgegrenzt ist. Diese Art der Beschreibung soll im Folgenden formal definiert werden. Hierzu wird zunächst die Erreichbarkeitsrelation eingeführt.

Definition 3.1.6 (Die Erreichbarkeitsrelation) *Ein Objekt b ist von einem Objekt a erreichbar $a \rightsquigarrow b$ falls:*

- *a referenziert b (direkte Referenz)*
- $\exists c : a \rightsquigarrow c, c \rightsquigarrow b$ *(Transitivität).*

Ein Objekt a referenziert ein Objekt b genau dann, wenn ein Attribut, welches Referenztyp ist, als Ziel Objekt b enthält.

Ausgehend von der Erreichbarkeitsrelation kann der Begriff der Kapsel definiert werden, indem über die Erreichbarkeit eine Zusammengehörigkeit von Objekten gegeben ist. Zusätzlich umfasst die Kapseldefinition eine Abgrenzung zwischen verschiedenen Kapseln, die den unabhängigen Austausch von Kapseln während der Laufzeit ermöglicht.

Definition 3.1.7 (Komponente zur Laufzeit - Kapsel) *Eine Kapsel K ist eine Menge von Objekten. Sei $K_{root} \subseteq K$ die Menge der Wurzelobjekte von K, welche durch die Projektion roots(K) abgebildet wird. Für jeden Zeitpunkt t an dem kein Methodenaufruf an einem Objekt der Menge K aktiv ist, gilt:*

- *Alle Wurzelobjekte gehören zur Kapsel K: $K_{root} \subseteq K$ (**Initialmenge**)*
- *Objekte die von einem Wurzelobjekt erreicht werden können, gehören zur Kapsel K: $K = \{x \| r \rightsquigarrow x, r \in roots(K)\}$ gdw. zusätzlich die Bedingung der Exklusivität und Abgrenzung erfüllt ist. (**Erreichbarkeit**)*
- *Ein Objekt gehört immer zu genau einer Kapsel. Für zwei Kapseln K,L gilt: $\forall x \in K : x \notin L$ (**Exklusivitität**)*
- *Falls $x \rightsquigarrow y$, $x \notin K$, $y \in K$: $\exists c \in K_{root}$ mit $x \rightsquigarrow c$ und $c \rightsquigarrow y$. (**Abgrenzung**)*

Abbildung 3.1 zeigt das Modell einer Kapsel. Die Kapselgrenzen werden durch die Wurzelobjekte bestimmt, über die Referenzen aus anderen Kapseln in die Kapsel verweisen. Instruktionen von Methoden der Objekte einer Kapsel werden durch Threads abgearbeitet. Threads können dabei direkt Methoden eines Wurzelobjekts aufrufen oder von einer anderen Kapsel durch einen Objektfernaufruf in eine Kapsel eintreten. Da die Kapseldefinition zunächst als statische Beschreibungen von Objektrelationen formuliert wurde, wird zusätzlich gefordert, dass ein Thread immer über ein Wurzelobjekt in eine Kapsel eintritt. Dies ermöglicht bei einer dynamischen Rekonfiguration die Verfolgung von Thread-Aktivitäten innerhalb einer Kapsel bzw. das Blockieren von Threads vor einer Aktualisierung.
Eine Kapsel kann nach der Definition als gerichteter Graph verstanden werden, der aus einer Menge ausgewiesener Wurzelobjekte entsteht. Die vorgestellte Definition ist nicht konstruktiv, das heißt es existieren oft mehrere Möglichkeiten eine Anwendung in Kapseln zu zerlegen. Unter anderem kann ein Graph einer komplexen Kapsel selbst leicht wieder in mehrere Teilkapseln zerlegt werden. Die Definition von Kapseln erfolgt über die Identifizierung von Wurzelobjekten und ist anwendungsspezifisch. Es hat sich aber in verschiedenen Anwendungsszenarien gezeigt, dass die Identifikation von Wurzelobjekten einfachen Regeln folgt und zusätzlich auch leicht in existierenden Anwendungen durchgeführt werden kann (siehe Kapitel 7).

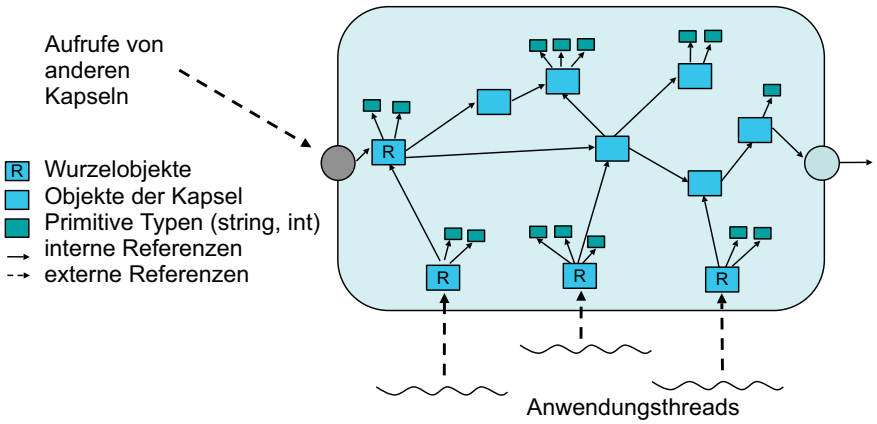


Abbildung 3.1: Modell einer Kapsel

Wurzelobjekte vereinigen die wohldefinierten Schnittstellen, die anderen Kapseln zur Verfügung gestellt werden, sowie definierte Eintrittspunkte für *Threads*. Die Aufteilung einer Anwendung und die Definition der Schnittstellen der einzelnen Kapseln ist ein wichtiger Schritt bei der Entwicklung adaptiver Anwendungen, da auf der Granularitätsebene der Kapseln Alternativverhalten integriert werden kann. Die Abgrenzung der Kapseln ist die Grundlage für die Möglichkeit des unabhängigen Austauschs von Anwendungsteilen, ohne dass die gesamte Anwendung von einer dynamischen Rekonfiguration betroffen ist.

Die Zusammensetzung einer Kapsel kann sich während der Laufzeit verändern. Das bedeutet, dass ihr Objektgraph durch die Abarbeitung von Methoden wachsen oder schrumpfen kann. Während der Abarbeitung einer Methode können sich zusätzlich Referenzen auf Objekte einer Kapsel auf dem Stack eines *Threads* befinden.

Eine Kapsel wird durch die Erzeugung eines Exemplars eines ausgewiesenen Typs gebildet. Der Konstruktor dieses Typs oder auch die Abarbeitung anderer Methoden kann weitere Objekte und auch weitere Wurzelobjekte erzeugen. Das initiale Objekt einer Kapsel wird als **Hauptobjekt** bezeichnet. Da ein Hauptobjekt immer einen Referenztyp besitzt, wird der ausgewiesene Typ als **Hauptklasse** bezeichnet. Die Hauptklasse ist in der **Haupt-Assembly** gespeichert. Das Hauptobjekt ist immer ein Wurzelobjekt.

Jede Kapsel stellt ihre Funktionalität über eine Schnittstelle zur Verfügung. Für die Implementierung von Schnittstellen wird das Schnittstellenkonzept (*interface*) verwendet, das in der CLI und auch in der Java-Plattform existiert. Eine Kapsel kann mehr als nur eine Schnittstelle unterstützen. Eine Kapsel kann aber auch keine Schnittstelle besitzen. Ohne Einschränkung der Allgemeinheit werden die Schnittstellen von genau einem Objekt einer Kapsel - dem Hauptobjekt - exportiert.

An dieser Stelle soll kurz der **logische Zusammenhang zwischen Kapseln und Assemblies** erläutert werden. Die Objekte einer Kapsel werden durch die Exemplarerzeugung der Typen einer Assembly gebildet. Jedes Objekt besitzt einen Typ und kann somit einer Assembly zugeordnet werden. Die Kapseln fassen nun während der Laufzeit Objekte mit Typen aus einer Assembly logisch zusammen. Dabei kann eine Kapsel auch Objekte mit Typen aus einer Menge von mehreren Assemblies enthalten, die dann gemeinsam austauschbare Einheiten bilden.

Es ist weiterhin möglich, dass zwei separate Kapseln Objekte aus ein und derselben

Assembly enthalten: Entweder, weil beide Objekte aus der gleichen Assembly erzeugt wurden, also Exemplare der selben Softwarekomponente sind, oder beide Kapseln gleiche abhängige Assemblies besitzen. Kapseln ermöglichen bei einem Austausch einer Assembly, die Änderungen auf die Laufzeitentitäten zu übertragen. Je nachdem welche Assemblies geändert wurden, wird die Menge der involvierten Kapseln bestimmt. Mit der Einführung des Kapselkonzepts müssen für die Aktivierung der Änderungen nur Kapseln betrachtet werden, die Objekte mit Typen aus geänderten Assemblies enthalten und nicht sämtliche Objekte einer Anwendung.

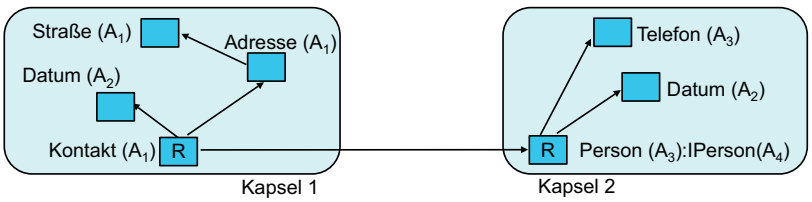


Abbildung 3.2: Beispiel möglicher Kapselgrenzen

Abbildung 3.2 zeigt ein Beispiel für mögliche Kapselgrenzen. In der Abbildung sind 2 Kapseln mit ihren Objekten dargestellt. Jedem Objekt ist ein Typ und in Klammern der Name der definierenden Assembly zugeordnet. Während Kapsel 1 Objekte aus den Assemblies A_1 und A_2 zusammenfasst, enthält Kapsel 2 Objekte der Assemblies A_2 und A_3. Der Typ *Person* implementiert die Schnittstelle *IPerson*. Dies bedeutet für die Abhängigkeiten der Assemblies, dass A_1 und A_3 jeweils A_4 referenzieren, da die Typen *Kontakt* und *Person* von der Schnittstelle *IPerson* abhängen. Die Abbildung gibt ein Beispiel dafür, dass die Implementierung von *IPerson* in Form von *Person* durch den Austausch der Assembly A_3 realisiert werden kann. Dabei wäre nur Kapsel 2 von der Rekonfiguration betroffen. Genauso könnte die Assembly A_1 ausgetauscht werden, ohne dass Kapsel 2 betroffen wäre. Für den Austausch von A_2 müssten die Kapselgrenzen alle gezeigten Objekte umfassen. In diesem Fall lägen dann auch A_1 und A_3 in einer neuen Version vor, da sie von A_2 abhängen und neu übersetzt werden müssten.

Durch die Definition der Schnittstellen in separaten Assemblies ist es möglich, die Implementierung einer Schnittstelle durch den Austausch der entsprechenden Assemblies und einer Übertragung der Änderungen auf die Laufzeitentitäten an wechselnde Umgebungseigenschaften anzupassen, z.B. indem alternative Algorithmen verwendet werden.

Kapseln kommunizieren durch den Aufruf von Methoden ihrer Objekte. Die Kommunikation zweier Kapseln erfolgt immer über einen **Konnektor**. Ohne Einschränkung der Allgemeinheit wird der Startpunkt eines Konnektors durch ein Attribut des Hauptobjekts einer Kapsel repräsentiert. Eine Kapsel kann mehrere Konnektoren besitzen, die durch je ein Attribut repräsentiert werden. Jeder Konnektor ist einer Schnittstelle auf der Senkenseite zugewiesen, welche von einem Objekt der Senkenkapsel unterstützt wird. Ein Konnektor wird also durch eine Referenz zwischen zwei Kapseln repräsentiert. Die Abarbeitung einer Methode (durch einen Thread) kann den Aufruf weiterer (**verschachtelter**) Methoden zur Folge haben. Wird eine Methode einer Kapsel aus dem Kontext einer Methode derselben Kapsel aufgerufen, wird dies als **re-entranter Aufruf** bezeichnet.

3.1.3 Die Anwendungskonfiguration

Nachdem der Begriff der Kapsel definiert ist, soll der Begriff der Anwendungskonfiguration eingeführt werden. Eine verteilte Anwendung kann als gerichteter Graph betrachtet werden, bei dem die Knoten **Kapseln** ($\mathcal{K}$) der Anwendung und die Kanten **Konnektoren** ($\mathcal{C}$) zwischen den Kapseln repräsentieren. Ein Konnektor wird auf der Quellseite durch ein *Quellattribut* $q \in Q_{K_1}$ einer Quellkapsel $K_1 \in \mathcal{K}$ identifiziert und mit einer *Schnittstelle* $s \in S_{K_2}$ einer Zielkapsel $K_2 \in \mathcal{K}$ verbunden. Damit definieren sich die Konnektoren durch die Menge $\mathcal{C} \in \mathcal{K} \times Q_{K_1} \times \mathcal{K} \times S_{K_2}$ mit $K_1, K_2 \in \mathcal{K}$. Die Verbindungspunkte einer Kapsel werden allgemein als Ports bezeichnet.
Die Abbildung $Z{:}K \rightarrow \{Rechner\}$ beschreibt die **Zuordnung** bzw. Platzierung (*placement*) von Kapseln auf die verfügbaren Rechner in einem verteilten System. Jede Kapsel wird genau einem Rechner zugeordnet. Die Menge der verfügbaren Rechner wird zunächst beim Start einer Anwendung als bekannt vorausgesetzt.
Das Verhalten und auch die nichtfunktionalen Eigenschaften einer Kapsel können durch veränderliche **Parameter** beeinflusst werden. Eine Kapsel kann keine oder mehrere veränderliche Parameter besitzen. Die Menge der Parameter einer Kapsel $K_1 \in \mathcal{K}$ ist durch die Menge P_{K_1} definiert. Ohne Einschränkung der Allgemeinheit sind alle Parameter einer Kapsel durch ein Attribut des Hauptobjekts der Kapsel repräsentiert. Der Typ eines Parameters wird durch die Typdeklaration eines Attributes der Hauptklasse bestimmt. Erlaubt sind dabei alle primitiven Typen der Laufzeitumgebung.

Definition 3.1.8 (Konfiguration) *Sei $\mathcal{K}$ eine Menge von Kapseln, $\mathcal{C}$ eine Menge von Konnektoren, Z eine Rechnerzuordnung und $\mathcal{P} = \bigcup_{K_i \in K} P_{K_i}$ die Menge aller Parameter einer Anwendung. Dann ist die **Konfiguration** einer Anwendung eindeutig durch das 4-Tupel $Konf(\mathcal{K}, \mathcal{C}, \mathcal{P}, Z)$ definiert. Für eine Konfiguration sind die Projektionen:*

- *comps(Konf) - die Menge der Kapseln einer Konfiguration,*
- *conns(Konf) - die Menge der Konnektoren einer Konfiguration und*
- *placement(Konf) - die Rechnerzuordnung einer Konfiguration definiert.*

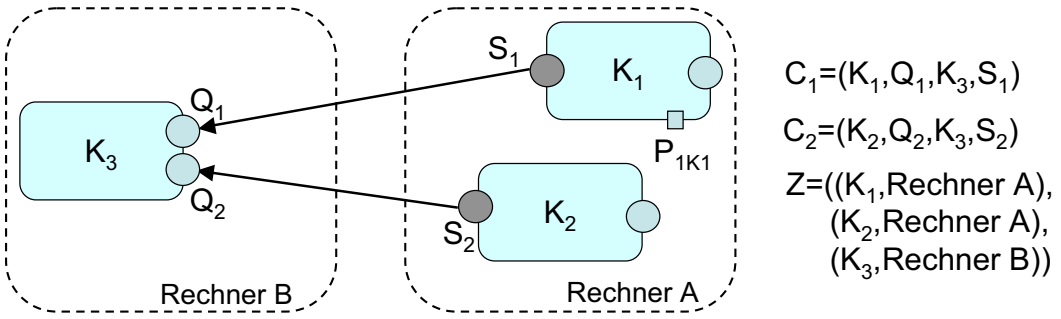


Abbildung 3.3: Beispiel einer Anwendungskonfiguration

Abbildung 3.3 zeigt ein Beispiel einer Anwendungskonfiguration. Die Konfiguration enthält drei Kapseln und ist auf zwei Rechnern verteilt. Die drei Kapseln K_1,K_2,K_3 sind durch die Konnektoren C_1 und C_2 miteinander verbunden.

Im Rahmen der Definition der Anwendungskonfiguration wurden Kapseln um einige Merkmale erweitert (Quellattribute, Schnittstellen, Parameter). Bis zu diesem Zeitpunkt wurde die Charakteristik einer Kapsel im Sinne der statischen Laufzeitstruktur beschrieben. Für die Konfigurierbarkeit müssen weitere Merkmale von Kapseln formal festgehalten werden.

Definition 3.1.9 (Erweiterte Kapseldefinition) *Eine Kapsel K besitzt zusätzlich zu den schon bekannten Eigenschaften:*

- *Eine Menge von Parametern P_K. Die Projektion params(K) entspricht P_K.*
- *Eine Menge von Quellattributen Q_K. Die Projektion sources(K) entspricht Q_K.*
- *Eine Menge von Schnittstellen S_K. Die Projektion interfaces(K) entspricht S_K.*
- *Eine Zeichenkette mit dem eindeutigen Bezeichner Name. Die Projektion name(K) entspricht dem Wert des Bezeichners Name.*

Die Kaspeldefinition wurde um einen Namen erweitert, um mögliche Parameteränderungen in der statischen Konfigurationsbeschreibung als Konfigurationsänderungen erkennen zu können. Der reine Umgang mit der Identität von Mengen ist nicht ausreichend, da bei geänderten Parametern ein neues Element in der Menge der Kapseln entsteht.
Quellattribute können wiederum als Tripel mit dem Namen des Attributs und den Typ- und Assembly-Namen des Schnittstellentyps defininiert werden. Für jeden Konnektor einer Konfiguration muss unter der Voraussetzung, dass T_Q der Typ seines Quellattributs und T_S der Typ einer Schnittstelle ist, gelten: $T_Q = T_S$ oder T_S ist von T_Q abgeleitet. Die Typen des Quellattributs und der Zielschnittstelle müssen also kompatibel sein.

3.1.4 Ausführungsmodell

Wie zuvor formal definiert, wird eine Anwendung als gerichteter Graph mit den Kapseln als Knoten und Konnektoren als Kanten betrachtet. Kapseln kommunizieren durch den Aufruf von Methoden an anderen Kapseln, die Teil deren Schnittstelle sind. Diese Aufrufe werden im Falle verteilter Kapseln als Objektfernaufrufe realisiert.
Die Instruktionen der Methoden der Objekte einer Kapsel können durch einen oder mehrere Kontrollflüsse (Threads) abgearbeitet werden. Der Kontrollfluss kann dabei entweder von einer Kapsel in eine andere durch einen Methodenaufruf übergehen oder ein kapseleigener Thread sein, der direkt auf den Methoden der Objekte der Kapsel arbeitet. Kapseleigene Threads können durch die Ausführung von Instruktionen der Objektmethoden einer Kapsel erzeugt werden. Die Aufrufe von kapseleigenen Threads werden als eingehende Konnektoren betrachtet. Das Ziel eines solchen Konnektors ist nicht auf Schnittstellen beschränkt, sondern kann direkt Methoden an einem Wurzelobjekt aufrufen. Wichtig festzuhalten ist an dieser Stelle, dass ein Thread immer nur Methoden eines Wurzelobjekts aufruft. Deren Aufruf kann dann den Aufruf beliebiger Methoden entsprechend der Kapseldefinition zur Folge haben.
Kontrollflüsse können sich über Rechnergrenzen hinweg fortpflanzen. Beim Überschreiten von Rechnergrenzen wechselt die Identität des Threads, da Betriebssystem-Threads nur lokal existieren. Der rufende Thread wird bis zur Rückkehr des Methodenaufrufs

angehalten, während auf der Seite der gerufenen Kapsel ein anderer Thread die Abarbeitung der Methode übernimmt. Kontrollflüsse können aber z.B. anhand von logischen Thread-IDs, wie sie in der CORBA-Spezifikation [Object Management Group 2004] bzw. in DCOM als *causality-ids* zur Implementierung von *single threaded apartments* [Thai 1999] eingeführt wurden, über Rechnergrenzen hinweg verfolgt werden. Später (Abschnitt 3.6.5) wird gezeigt, dass logische Threads auch in der verwendeten .NET Remoting Middleware implementiert werden können. Logische Thread-IDs werden für die Implementierung der vorgestellten Rekonfigurationsalgorithmen benötigt.

In der Welt der dynamischen Rekonfiguration wurde von Kramer und Magee das **Konzept der Kommunikationstransaktion** eingeführt [Kramer und Magee 1990], welches virtuell eine Menge von Interaktionen zwischen zwei Entitäten bündelt. Dieses Konzept soll auch für Methodenaufrufe zwischen zwei Kapseln im Rahmen dieser Arbeit verwendet werden. Im Original wird das Konzept der *transactions* eingeführt. Die deutsche Übersetzung als Kommunikationstransaktion ist dem Buch „Verteilte Systeme und Rechnernetze“ [Kramer und Sloman 1988] entnommen.

Eine **Kommunikationstransaktion** fasst eine oder mehrere Methodenaufrufe über einen Konnektor zusammen. Eine Kommunikationstransaktion ist immer gerichtet, wobei sie in der initiierenden Kapsel beginnt. Kommunikationstransaktionen werden in endlicher Zeit beendet und der Initiator einer Kommunikationstransaktion wird über deren Ende informiert. Diese Forderung gilt im Rahmen dieser Arbeit auch für einzelne Methodenaufrufe zwischen den Kapseln. Eine Kommunikationstransaktion besteht immer zwischen zwei Kapseln. Sind mehrere Kapseln an einer Kommunikationstransaktion beteiligt, werden Abhängigkeiten zwischen Kommunikationstransaktionen eingeführt. Eine Kommunikationstransaktion t_1 hängt von einer nachfolgenden Kommunikationstransaktion t_2 ab (t_1/t_2), wenn ihre erfolgreiche Ausführung von der Beendigung von Kommunikationstransaktion t_2 abhängt. Eine Kommunikationstransaktion die gestartet, aber noch nicht beendet ist, wird als **offen** bezeichnet.

Bei der Ausführung von Anwendungen soll zunächst davon ausgegangen werden, dass diese fehlerfrei erfolgt. Das Auftreten von Fehlern und ein entsprechendes Fehlermodell soll in einem späteren Kapitel (6) im Rahmen eines Ausblicks diskutiert werden.

3.2 Anforderungen an die dynamische Rekonfiguration

Die **dynamische Rekonfiguration** bezeichnet die Änderung der Konfiguration einer Anwendung während ihrer Laufzeit. Mit den vorgestellten Konzepten der Kapsel und der Anwendungskonfiguration wurden die Voraussetzungen geschaffen, das Verhalten einer Anwendung in separaten Teilen während der Laufzeit auszutauschen und damit die Eigenschaften der Anwendung zu justieren. Wie der Austausch auf Basis von Komponenten während der Laufzeit umgesetzt werden kann, wird in diesem Abschnitt beschrieben. Zunächst soll ein Korrektheitskriterium für den Vorgang der dynamischen Rekonfiguration formuliert werden.

3.2.1 Erhaltung von Konsistenz

Die dynamische Rekonfiguration einer Anwendung darf nicht die Konsistenz ihrer Daten verletzen. Kramer und Magee beschreiben den Begriff der Konsistenz zunächst informal als Zustand, in dem die Verarbeitung normal weitergeführt werden kann

und nicht in einem Fehlerzustand endet [Kramer und Magee 1990]. Nach [Moazami-Goudarzi 1999] wird die Konsistenz einer Anwendung durch deren Korrektheit definiert. Ein verteiltes System wird als korrekt bezeichnet, wenn die folgenden drei Bedingungen erfüllt sind [Moazami-Goudarzi 1999]:

- **Das System ist strukturell integer.** Diese Bedingung erfasst die Korrektheit einer Anwendungskonfiguration. Im Wesentlichen müssen hier für alle Kapseln Bedingungen bezüglich ihrer Kommunikationspartner erfüllt sein. Die Schnittstellen müssen kompatibel verbunden, Parameter bestimmt und alle benötigten Schnittstellen konnektiert sein.

- **Die Kapseln des Systems befinden sich in einem gegenseitig gültigen Zustand.** Kapseln verändern ihren internen Zustand durch Interaktion mit anderen Kapseln. Nach dem erfolgreichen Abschluss einer Interaktion befinden sich alle beteiligten Kapseln wieder in einem gültigen Zustand. Wird z.B. während der zeilenweisen Übertragung eines Bildes das verwendete Datenformat geändert, ist die Bedingung des gegenseitig gültigen Zustands verletzt, da in der Zielkapsel das Bild nicht mehr dargestellt werden kann.

- **Die anwendungsspezifischen Invarianten des Systems sind gültig.** Anwendungsspezifische Invarianten sind Prädikate, die Aussagen über den Zustand einer Teilmenge der Bestandteile des Systems treffen. Ein Beispiel hierfür ist das Vorhandensein eines einzigen Zugriffstokens innerhalb eines Synchronisationsverfahrens (vgl. Tokenringprotokoll).

Eine dynamische Rekonfiguration darf diese Konsistenzbedingungen nicht verletzen. Im Folgenden soll zunächst diskutiert werden, was die Erhaltung der vorgestellten Konsistenzbedingungen für die eingesetzten Komponentenplattformen bedeutet. Anschließend sollen Möglichkeiten für die Erhaltung der Anwendungskonsistenz erörtert werden. Während existierende Ansätze der dynamischen Rekonfiguration in einem separaten Abschnitt beschrieben werden (Abschnitt 8.2), werden im Rahmen dieses Kapitels die vom Autor dieser Arbeit entwickelten Algorithmen vorgestellt.

Strukturelle Konsistenz

Die Sicherstellung der strukturellen Konsistenz einer Anwendung muss zunächst für jede Konfiguration erfolgen. Dies kann z.B. durch ein Regelwerk bei der Konfigurationserstellung erfolgen. Die Typkompatibilität von Konnektoren, aber auch die Erfüllung sämtlicher benötigter Schnittstellen an den einzelnen Quellattributen der Kapseln muss gewährleistet sein. Die von einer Kapsel benötigten Schnittstellen müssen kompatibel sein mit der von der Zielkapsel implementierten Schnittstelle. Benötigt die Quellkapsel die Funktionalität einer Schnittstelle I_1, muss die Zielkapsel diese auch implementieren - direkt oder in Form einer abgeleiteten Schnittstelle. Für die Einhaltung dieses Konsistenzkriteriums muss während einer dynamischen Rekonfiguration sichergestellt werden, dass keine Interaktionen zwischen den Kapseln verschiedener Konfigurationen aktiv sind bzw. ein Aufruf an einer Referenz durchgeführt werden, die gerade noch kein gültiges Ziel hat. Ausgehend von der Erfüllung struktureller Konsistenz einzelner Konfigurationen muss die dynamische Rekonfiguration ein atomares Umschalten zwischen gültigen Konfigurationen realisieren. Die Umschaltung zwischen Konfigurationen kann nur als Folge von Einzelschritten (Erzeugen einer Kapsel, Erstellen einer Verbindung

usw.) realisiert werden. Während der Abarbeitung dieser Einzelschritte muss sichergestellt sein, dass keine Anwendungsaktivitäten einen temporär inkonsistenten Zustand erfahren. Die Einzelschritte einer Konfiguration dürfen also nicht mit Kapselinteraktionen interferieren. Die Rekonfigurationsschritte werden in einem separaten Thread ausgeführt, der potentiell konkurrierend zu Anwendungs-Threads ausgeführt wird.

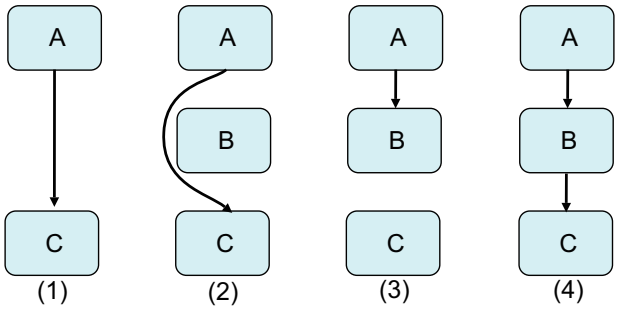


Abbildung 3.4: Rekonfigurationsschritte

Abbildung 3.4 zeigt die Umschaltung zwischen zwei Anwendungskonfigurationen. Zwischen Kapsel A und C wird eine neue Kapsel B hinzugefügt. Die nötigen Einzelschritte sind in der Abbildung dargestellt. Würde während der Umschaltung von der Ausgangskonfiguration zur neuen Konfiguration zum Zeitpunkt 2 oder 3 eine Interaktion stattfinden, kann die Anwendung in einen ungültigen Zustand gelangen. Wenn z.B. zum Zeitpunkt 3 Kapsel A eine Methode an B aufruft, was einen Aufruf an C zur Folge hätte, könnte es zu einer Ausnahme (*Exception*) in B kommen, da die Verbindung zwischen B und C noch nicht etabliert wurde.

Gegenseitig gültiger Kapselzustand

Übertragen auf die eingesetzten Komponentenplattformen bedeutet die Erhaltung eines gegenseitig gültigen Zustands für Kapseln, dass die Abarbeitung von Methoden der Kapseln nicht während einer zusammenhängenden Interaktion (einer Kommunikationstransaktion) unterbrochen werden darf. Während einer Kommunikationstransaktion kann sich ein Teil des Systemzustandes auf dem Stack eines Threads oder in den Registern der CPU befinden bzw. gerade über eine Netzwerkverbindung übertragen werden. Da bei einer Aktualisierung einer Kapsel auf den gesamten Kapselzustand zugegriffen werden muss und der Zugriff auf den Stack, die Register und das Netzwerk in den betrachteten Laufzeitumgebungen nicht (effizient) möglich ist, muss dafür Sorge getragen werden, dass keine offenen (gestartete, aber noch nicht beendete) Kommunikationstransaktionen an beteiligten Kapseln existieren oder dass beim Verlust eines solchen Teilzustandes ein gültiger Zustand wiederhergestellt wird.

Erhalt von Invarianten im Kontext des Anwendungsmodells

Die Erhaltung von Invarianten ist in einigen Fällen anwendungsspezifisch. Beim Entfernen einer Kapsel kann Zustand verloren gehen, über den Invarianten formuliert wurden. Ein Beispiel ist das Token, welches sich zum Zeitpunkt des Entfernens innerhalb der Kapsel befindet. Problematisch kann auch das Hinzufügen einer Kapsel sein, die ihren Initialzustand in Abhängigkeit des Zustands anderer Kapseln bestimmen muss,

um geforderte Invarianten nicht zu verletzen. Der Erhalt von anwendungsspezifischen Invarianten erfordert also in einigen Fällen die Ausführung von anwendungsspezifischer Logik, die in Form von Initialisierungs- und Finalisierungsroutinen integriert werden können. Es muss beachtet werden, dass die Initialisierungs- und Finalisierungsroutinen potentiell Zugriff auf den Zustand aller Kapseln einer Anwendung besitzen müssen (Ein Beispiel hierfür ist die Wiederherstellung des verlorenen Tokens). Die Ausführung von Initialisierungs- und Finalisierungsroutinen darf dann nicht mit Anwendungsaktivitäten interferieren.

Nach Moazami-Goudarzi können zwei prinzipielle Ansätze der Konsistenzerhaltung unterschieden werden. Bei Algorithmen mit dem Schema **„Konsistenz durch Erholung“** (*consistency through recovery*) werden Rekonfigurationsaktionen unmittelbar ausgeführt. Unterbrochene Kommunikationstransaktionen müssen mit Hilfe entsprechender Mechanismen zurückgenommen (*roll back*) und wiederholt werden, um zu einem gültigen Zustand der Vergangenheit zu gelangen. Voraussetzung für diese Kategorie von Algorithmen sind entsprechende Mechanismen zur Wiederherstellung (*Recovery*), wie sie z.B. in Datenbankmanagementsystemen bzw. Transaktionsmonitoren zu finden sind. Arbeiten mit Konsistenzerhaltung durch Erholung finden sich in Argus [Bloom und Day 1993]. Auch Palma et. al. [Palma u. a. June 2001] bilden Anwendungsinteraktionen direkt auf Transaktionen ab und synchronisieren diese mit Rekonfigurationstransaktionen mit starker Isolation mit Hilfe eines erweiterten Transaktionsmonitors. Auf der anderen Seite umfasst das Schema **„Konsistenz durch Vermeidung“** (*consistency through avoidance*) Algorithmen, welche die Ausführung von Rekonfigurationsaktionen zu einem Zeitpunkt vermeiden, an dem sie zu Inkonsistenzen führen könnten.

In den Laufzeitsystemen moderner Komponentenplattformen ist kein inhärenter Mechanismus für die Zurücknahme von Interaktionen integriert. Es existieren zwar Produkte (z.B. *Microsoft Transaction Server* [Limprecht 1997]) für die Implementierung transaktionaler Systeme, diese werden häufig nur für komplexe Unternehmenssoftware verwendet, weil sie unter anderem die Programmierung komplexer gestalten. Insbesondere müssten Zurücknahmeaktionen anwendungsspezifisch implementiert werden. Im Rahmen dieser Arbeit sollen Algorithmen nach dem Schema „Konsistenz durch Vermeidung“ betrachtet werden. Um adaptives Verhalten mit Hilfe dynamischer Rekonfiguration zu realisieren, ist es effizienter, auf einen sicheren Zustand in der Zukunft zu warten als teure Zurücknahmeaktionen in Kauf zu nehmen. In einem späteren Kapitel wird die dynamische Rekonfiguration von eingebetteten Echtzeitsystemen betrachtet, bei denen das Konzept der „Konsistenz durch Erholung“ noch einmal betrachtet wird.

Eine Rekonfiguration soll genau dann durchgeführt werden, wenn sich kein Kapselzustand auf dem Stack, in den Registern oder auf dem Netzwerk befindet, also keine offenen Kommunikationstransaktionen existieren. Dieser Zustand einer Anwendung wird in der Literatur auch als *frozen state* oder *quiescence* bezeichnet [Kramer und Magee 1990]. Im Rahmen dieser Arbeit wird dieser Zustand im Deutschen als **rekonfigurierbar** bzw. auch inaktiv bezeichnet. Die Herstellung eines solchen Zustands der Inaktivität ist ein zentrales Problem bei der dynamischen Rekonfiguration. Insbesondere sollen Einschränkungen der gesamten Anwendung möglichst minimiert werden und nur von der dynamischen Rekonfiguration betroffene Teile inaktiviert werden. In der Literatur existieren einige Lösungen für das Problem der Inaktivierung; für die vorgestellte Laufzeitumgebung sind diese aber nicht direkt anwendbar. Existierende Lösungen für dieses Problem werden im separaten Abschnitt 8.2 erläutert.

3.3 Dynamische Rekonfiguration in einer Komponentenplattform

Im diesem Abschnitt sollen die im Adapt.Net-Projekt verwendeten Algorithmen zur dynamischen Rekonfiguration vorgestellt werden. Je nach den durch eine Anwendungskonfiguration gegebenen Umständen können unterschiedlich optimierte Algorithmen verwendet werden, die für eine gegebene Situation die beste Leistung bieten. Leistungsmaße umfassen zusätzliche Instruktionen (Zusatzkosten), die den normalen Methodenaufrufen hinzugefügt werden müssen, um Rekonfigurations- und Anwendungsaktivitäten zu synchronisieren. Weitere wichtige Leistungsmaße sind die Dauer der Rekonfigurations- und Unterbrechungsphase.
Die Anforderungen an einen Rekonfigurationsalgorithmus für die definierte Laufzeitumgebung sollen noch einmal kurz zusammengefasst werden. Diese sind:

- Unterstützung multipler Threads
- Geringe Zusatzkosten für funktionale Methodenaufrufe
- Nutzung von Basistechnologien (keine Manipulation der Laufzeitumgebung)
- Erhalt der Anwendungskonsistenz

Ein wesentliches Unterscheidungskriterium bei der dynamischen Rekonfiguration ist zunächst die Existenz von Zyklen im Verbindungsgraphen einer Anwendung. Der Ruhezustand in Konfigurationen mit azyklischen Verbindungsgraphen kann effektiver durch die Ordnung von Blockierungsoperationen, wie von [Bidan u. a. 1998] theoretisch vorgeschlagen, erreicht werden, während bei der Existenz von Zyklen Methodenaufrufe bzw. Kommunikationstransaktionen selektiv in jeder beteiligten Kapsel blockiert werden müssen. Zunächst soll die Problemstellung beim Erreichen des Ruhezustandes formalisiert werden. Hierzu werden an einer dynamischen Rekonfiguration beteiligte Elemente als Menge von Rekonfigurationselementen definiert, welche die in einer Ausgangskonfiguration beteiligten Kapseln und Konnektoren umfassen. Rekonfigurationselemente werden durch elementare Rekonfigurationsoperationen wie z.B. das Hinzufügen einer Kapsel, das Ändern eines Parameters usw. bestimmt. Ausgehend von einer Ausgangskonfiguration (alte Konfiguration) und dem Vergleich mit einer Zielkonfiguration (neue Konfiguration) können die benötigten Rekonfigurationsoperationen für die Aktivierung der Zielkonfiguration berechnet werden.

Definition 3.3.1 (Rekonfigurationselemente) *Sei $Konf_A(K_A, C_A, P_A, M_A)$ eine Ausgangskonfiguration und $Konf_N(K_N, C_N, P_N, M_N)$ die Zielkonfiguration mit den beteiligten Kapseln (K), den Konnektoren (C), den Parametern (P) und der Rechnerzuordnung (M) einer dynamischen Rekonfiguration. Als Rekonfigurationselemente wird die Menge $RE \in K_A \cup C_A$ bezeichnet, bei der für jedes $e \in RE$ gilt: e ist in die Rekonfiguration involviert. Ein Konnektor oder eine Kapsel ist in eine Rekonfiguration involviert, wenn für $e \in K_A \cup C_A$ mindestens eines der folgenden Prädikate wahr ist:*

- $e \in K_A$ *und* $\nexists f \in K_N$ *mit* $name(e) = name(f)$ *(Entfernen einer Kapsel)*
- $p_{ne} \neq p_{nf}$ *mit* $e \in K_A, f \in K_N$ *und* $name(e) = name(f)$ *(Parameteränderung)*
- $\exists e \in C_A$ *mit* $e \notin C_N$ *(Ändern eines Konnektors, Schnittstellenänderung)*
- $\exists e \in K_A$ *mit* $(e, R_1) \in M_A$ *und* $(e, R_2) \in M_N$ *und* $R_1 \neq R_2$ *(Migration)*

Die Änderung einer Assembly einer Kapsel wird als dynamische Aktualisierung bezeichnet. Durch eine dynamische Aktualisierung kann z.B. eine alternative Implementierung für eine gegebene Schnittstelle aktiviert werden. Dynamische Aktualisierungen von Kapseln seien zunächst als Änderung eines Parameters behandelt, wobei die Versionsnummer der zugrundeliegenden Assembly als zusätzlicher Parameter betrachtet wird. Neue Kapseln können über veränderte Konnektoren erkannt werden, müssen allerdings bei der Herstellung eines rekonfigurierbaren Zustands nicht betrachtet werden, da sie in der alten Konfiguration nicht vorhanden sind und somit keine Inkonsistenzen erfahren können. Die Anzahl der Parameter einer Kapsel ist zunächst als konstant angenommen, da eine Parameteränderung im Modell an der Nummer des Parameters identifiziert wird. Bei der Implementierung werden Parameteränderungen über Parameternamen verarbeitet.
Eine dynamische Rekonfiguration setzt sich aus einer Reihe elementarer Rekonfigurationsoperationen zusammen. Jede komplexe Konfigurationsänderung kann durch die Ausführung der Basisoperationen durchgeführt werden. Die Basisoperationen sind:

- das Erzeugen einer neuen Kapsel,
- das Entladen einer Kapsel,
- das Ändern eines Parameters,
- das Entfernen eines Konnektors und
- das Erstellen eines Konnektors.

RO wird als die Menge von Rekonfigurationsoperationen für eine Konfigurationsänderung verwendet. Um die Konsistenz einer Anwendung durch eine dynamische Rekonfiguration nicht zu verletzen, müssen, wegen des schon angesprochenen Kriteriums der strukturellen Konsistenz, zusammengesetzte Rekonfigurationsoperationen atomar ausgeführt werden. Um diese Bedingung formal festhalten zu können, wird die Menge der Transaktionselemente definiert.
Die Menge der Transaktionselemente (TE) definiert die an einer Kommunikationstransaktion beteiligten Kapseln und Konnektoren. Die Menge der Anwendungstransaktionselemente (AT) wird definiert als die Menge aller an irgendeiner Kommunikationstransaktion beteiligten Elemente. Dies sind typischerweise alle Kapseln und Konnektoren einer Anwendung.

Definition 3.3.2 (Transaktionselemente) *Sei K die Menge der Kapseln einer Konfiguration und C die Menge der Konnektoren. Die Transaktionselemente einer Kommunikationstransaktion t sind definiert durch die Menge $TE_t \in K \cup C$, mit:*

- *$e \in TE_t$ und $e \in K$ initiiert die Kommunikationstransaktion t, oder*
- *$e \in TE_t$ und $e \in C$ t findet über den Konnektor e statt, oder*
- *$e \in TE_t$ und $e \in K$ ist Zielkapsel der Kommunikationstransaktion t.*

Sei AT die Menge aller Kommunikationstransaktionen einer Anwendung. Dann ist die Menge der Anwendungstransaktionselemente $TEs \in K \cup C$ definiert durch:

- $TEs = \bigcup_{e \in TE_t \wedge t \in AT} e$.

Ausgehend von der Definition der Transaktions- und Rekonfigurationselemente kann nun mit Hilfe einer logischen Zeitrelation das zentrale Entwurfsziel eines Rekonfigurationsalgorithmus formuliert werden: Kommunikationstransaktionen und Rekonfigurationsoperationen dürfen, bei gleichen beteiligten Elementen, zeitlich nicht überlappend ausgeführt werden. Diese Bedingung wird als Konsistenzkriterium der Rekonfiguration bezeichnet.

Definition 3.3.3 (Konsistenzkriterium der Rekonfiguration) *Sei AT die Menge aller Kommunikationstransaktionen einer Anwendung, RE die Menge der Rekonfigurationselemente der betrachteten Konfigurationsänderungen. Sei weiterhin $T_{Start}(a)$ der Initiierungszeitpunkt einer Aktion a und $T_{Ende}(a)$ der entsprechende Endzeitpunkt der Aktion und r die Rekonfigurationsaktion selbst, dann muss ein Rekonfigurationsalgorithmus sicherstellen, dass gilt:*

- $\forall t \in AT : TE(t) \cap RE \neq \emptyset \rightarrow T_{Start}(r) > T_{Ende}(t)$ *oder* $T_{Ende}(r) < T_{Start}(t)$.

Mit anderen Worten ausgedrückt, darf während einer Rekonfiguration keine Kommunikationstransaktion unter Beteiligung der an der Rekonfiguration beteiligten Elemente offen sein.

3.3.1 Rekonfiguration mit azyklischen Verbindungsgraphen

Ausgehend von Wermelingers Ansatz [Wermelinger 1997], der auf dem Blockieren beteiligter Konnektoren basiert, kann bei Anwendungen mit azyklischen Verbindungsgraphen ein Rekonfigurationsalgorithmus verwendet werden, der auf der Ordnung von Blockierungsoperationen basiert. Ein Konnektor gilt als **blockiert**, wenn alle offenen Kommunikationstransaktionen über den Konnektor beendet sind und keine neuen Kommunikationstransaktionen über den Konnektor initialisiert werden können. Im Prinzip kann auf das Blockieren eines Konnektors passiv gewartet werden, bis alle Kommunikationstransaktionen beendet sind. Da es allerdings vorkommen kann, dass sich Kommunikationstransaktionen immer wieder überlappen, das heißt eine neue Kommunikationstransaktion initiiert wird, bevor eine laufende Kommunikationstransaktion beendet ist, wurde im Rahmen dieser Arbeit das Blockieren aktiv realisiert. Beim Blockieren wird sofort das Initiieren neuer Kommunikationstransaktionen verhindert und auf das Ende offener Kommunikationstransaktionen gewartet. Die vorausgesetzte zeitliche Endlichkeit der Kommunikationstransaktionen garantiert, dass ein Konnektor in endlicher Zeit blockiert werden kann.
Die Blockierungsoperationen müssen geordnet werden, um mögliche Verklemmungen (*deadlocks*) zu vermeiden. Ein kurzes Beispiel soll einen Beleg für mögliche Verklemmungen geben.

Abbildung 3.5: Blockierung abhängiger Kommunikationstransaktionen

Abbildung 3.5 zeigt eine Anwendungskonfiguration mit den drei Kapseln *Client*, *Proxy* und *Server*. Die Kapsel *Client* initiiert über einen Konnektor C_1 eine Kommunikationstransaktion T_1, welche von einer Kommunikationstransaktion T_2 über einen Konnektor

C_2 zwischen *Proxy* und *Server* abhängt. Würde nun Konnektor C_2 als erstes blockiert, könnte eine daraufhin gestartete Kommunikationstransaktion über C_1 nie beendet werden, da die abhängige Kommunikationstransaktion T_2 nicht initiiert werden könnte. Konnektor C_1 muss also vor Konnektor C_2 blockiert werden.
Um Verklemmungen zu verhindern, muss die Reihenfolge des Blockierens an Hand des Verbindungsgraphen ermittelt werden. Die Richtung der Kanten im Verbindungsgraphen beschreibt eindeutig die Richtung, in der Kommunikationstransaktionen initiiert werden können, nämlich vom Quellknoten zum Zielknoten. Mit Hilfe dieser Informationen kann die Blockierungsreihenfolge (*BlockOrder*) berechnet werden:

Definition 3.3.4 (Blockierungsreihenfolge) *Sei RE die Menge der Rekonfigurationselemente und $RE \cap C$ die Menge der von der Rekonfiguration betroffenen Konnektoren. Für jeden Konnektor $C_1(K_{qC_1}, Q_{C_1}, K_{zC_1}, S_{C_1}), C_2(K_{qC_2}, Q_{C_2}, K_{zC_2}, S_{C_2}) \in RE \cap C$ ist die Abbildung $BlockOrder : C \rightarrow \mathbb{N}$ definiert durch:*

- $BlockOrder(C_1) < BlockOrder(C_2)$ *gdw.* $K_{zC_1} = K_{qC_2}$

Für die Blockierungsreihenfolge gilt für $BlockOrder(C_1) < BlockOrder(C_2)$, dass C_1 vor C_2 blockiert wird.

Ein Konnektor C_1 mit einer Zielkapsel K_1 besitzt also eine niedrigere *BlockOrder* als ein Konnektor C_2 mit K_1 als Quellkapsel. Die Abbildung *BlockOrder* ist nicht injektiv. Es kann mehreren Konnektoren die gleiche Ordnungszahl zugeordnet werden. Die Blockierung dieser Konnektoren kann parallel ausgeführt werden. Im Falle der Abwesenheit von Zyklen im Verbindungsgraphen kann immer eine Blockierungsreihenfolge gefunden werden. Ein Algorithmus für die Berechnung der *BlockOrder* ist im folgenden Pseudocode (Algorithmus 1) dargestellt.

Algorithmus 1 Berechnung der Blockierungsordnung: n $BlockOrder(C_x)$

```
1: VorgängerKonnektoren(C_x)=∅
2: for all C_y ∈ C do
3:    if C_y.Senkenkomponente = C_x.Quellkomponente then
4:       C_y → VorgängerKonnektoren(C_x)
5:    end if
6: end for
7: if VorgängerKonnektoren(C_x) = ∅ then
8:    return 0;
9: else
10:   return max(BlockOrder(VorgängerKonnektor(C_x)))+1;
11: end if
```

Der Algorithmus terminiert in endlicher Zeit und besitzt eine quadratische Komplexität $O(n^2)$ bzgl. der Untersuchungen für jeden Konnektor eines Verbindungsgraphen. Der beschriebene Algorithmus ist bzgl. der erreichbaren Parallelität nicht optimal, da angenommen wird, dass zwischen allen Konnektoren Abhängigkeiten existieren - für je zwei Kanten im Verbindungsgraphen mit einem gemeinsamen Knoten abhängige Kommunikationstransaktionen existieren. Um den Algorithmus optimal implementieren zu können, müsste der Anwendungsentwickler für jedes Konnektorpaar Abhängigkeitbeziehungen definieren. Der gewonnene Laufzeitvorteil bei der Blockierung steht zum Aufwand bei der Entwicklung in keinem vorteilhaften Verhältnis.

Wesentlichen Einfluss auf die Leistungsfähigkeit eines Rekonfigurationsmechanismus haben die verwendeten Betriebssystemmechanismen für das Blockieren von Kommunikationstransaktionen. Mögliche Implementierungsstrategien für das Blockieren von Verbindungen wurden im Rahmen dieser Arbeit ausführlich untersucht und werden im Rahmen der Implementierung in einem folgenden Abschnitt vorgestellt. Letztendlich handelt es sich beim Blockieren von Verbindungen um eine komplexe Synchronisationsoperation. Die dynamische Rekonfiguration selbst wird in einem separaten Thread ausgeführt, welcher mit den Anwendungs-Threads synchronisiert werden muss.

3.3.2 Rekonfiguration mit zyklischen Verbindungsgraphen - ReDAC

Im Falle von zyklischen Verbindungsgraphen kann eine Menge von Konnektoren, über die abhängige Kommunikationstransaktionen existieren, nicht in jedem Fall ohne Verklemmungen blockiert werden. Ein Beispiel hierfür ist in Abbildung 3.6 dargestellt.

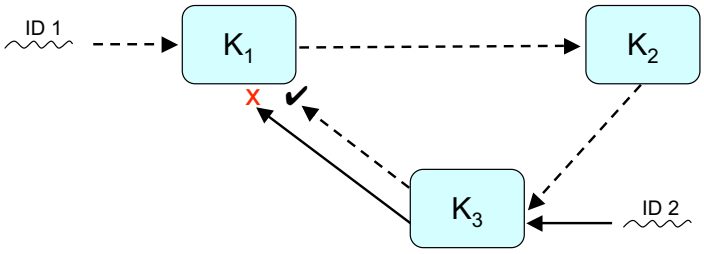


Abbildung 3.6: Rekonfiguration mit zyklischer Verbindungsstruktur

Im Beispiel sind drei Kapseln zyklisch miteinander verbunden. Es existieren zwei Kapseln K_1 und K_3 mit eigenen Threads. Methodenaufrufe einer Kommunikationstransaktionen können re-entrant sein, das heißt ein Thread kann in eine Kapsel mehr als einmal eintreten. Angenommen die Kapseln K_1 und K_2 sollen dynamisch aktualisiert werden, also durch eine neue Version ersetzt werden, dann bedeutet dies für den Rekonfigurationsalgorithmus, dass die komplette Verarbeitung der Kapseln K_1 und K_2 in einen Ruhezustand versetzt werden muss, das heißt keine offene Kommunikationstransaktion mehr innerhalb dieser Kapseln existieren darf. Würde nun der Konnektor zwischen K_3 und K_1 blockiert, erkennt man, dass hier eine Verklemmung entsteht. Die Verarbeitung des Threads mit der ID 1, dessen Kontrollfluss durch die gestrichelte Linie symbolisiert wird, würde blockiert. Dieser hat aber offene Kommunikationstransaktionen in K_1 und K_2. Auch der Thread mit der ID 2 initiiert eine Kommunikationstransaktion über den Konnektor zwischen K_3 und K_1. Die Blockierung dieses Konnektors ist kein Problem, da keine offenen Kommunikationstransaktionen in K_1 und K_2 bestehen. Das heißt, ein entsprechender Rekonfigurationsalgorithmus muss selektiv die Initiierung neuer Kommunikationstransaktionen für bestimmte Threads erlauben, andere Threads, die keine offenen Kommunikationstransaktionen auf Rekonfigurationselementen besitzen, aber direkt blockieren.
Für einen entsprechenden Rekonfigurationsalgorithmus muss also ein Verfahren entwickelt werden, bei dem für jeden Thread die Menge der offenen Kommunikationstransaktionen und der darin involvierten Elemente bekannt ist. An dieser Stelle müssen die grundlegenden Zielsetzungen der Arbeit beachtet werden. Die entsprechenden Algorithmen sollen durch die Nutzung von Basistechnologien auf virtuellen Ausführungsumgebungen implementierbar sein. Eine bekannte Technologie zur Problemlösung ist

die Untersuchung des Ausführungs-Stacks (*stack walk*) der Threads. Das heißt, die Evaluierung offener Kommunikationstransaktionen kann durch eine Analyse der von einem Thread gerufenen Methoden erfolgen. Für jeden Methodenaufruf wird ein Speicherbereich (*stack frame*) alloziert, an Hand dessen sich die offenen Kommunikationstransaktionen ermitteln ließen. Die Verwendung dieses Verfahrens ist insofern problematisch, als in den Laufzeitumgebungen der Java- und CLI-Plattformen kein direkter (effizienter[2]) Zugriff auf den Stack möglich ist. Zusätzlich ist die Untersuchung eines Stacks im verteilten Fall nicht effizient implementierbar. Im Rahmen der Arbeit wird der Ansatz verfolgt, mit Hilfe der aspektorientierten Programmierung Logik in die Kapseln zu integrieren, welche die benötigten Informationen für den Rekonfigurationsalgorithmus verwalten.
Ein neuer Rekonfigurationsalgorithmus **ReDAC** (*dynamic REconfiguration of Distributed component-based Applications with Cyclic dependencies*), der sich auch sehr effizient umsetzen lässt, wurde vom Autor dieser Arbeit entwickelt und soll im Folgenden im Mittelpunkt stehen. Um die offenen Kommunikationstransaktionen für jeden Thread identifizieren zu können, wird in jeder Kapsel beim Eintritt eines Threads ein Thread-spezifischer Zähler inkrementiert und beim Austritt dekrementiert. Beim Blockieren von Kapseln kann durch die Analyse der Zähler ermittelt werden, ob ein Thread noch offene Kommunikationstransaktionen auf einer Kapsel besitzt und damit neue Kommunikationstransaktionen initiieren darf. Für die Beschreibung des Algorithmus muss zunächst die Menge der zu blockierenden Kapseln definiert werden.

Definition 3.3.5 (BlockSet) *Seien K_A und C_A die Kapseln und Konnektoren einer Ausgangskonfiguration und $RE \in K_A \cup C_A$ die Menge der Rekonfigurationselemente für eine Konfigurationsänderung. Dann ist die Menge $BlockSet \subset K_A$ definiert durch:*

$$BlockSet = RE \cap K_A. \tag{3.1}$$

Für den Rekonfigurationsalgorithmus wird ein Mechanismus zur rechnerübergreifenden Identifikation von Threads benötigt, der als logische Thread-IDs bekannt ist. Die Existenz logischer Thread-IDs soll an dieser Stelle vorausgesetzt sein. Die Implementierung eines solchen Mechanismus wird in einem anschließenden Abschnitt (3.6.5) vorgestellt. Die benötigten Zähler lassen sich effizient in Hash-Tabellen (z.B. *CLI Hashtable/Dictionary, Java Hashmap*) verwalten: mit der Verwendung einer logischen Thread-ID als Schlüssel und dem Wert des Zählers als Eintrag in der Tabelle.

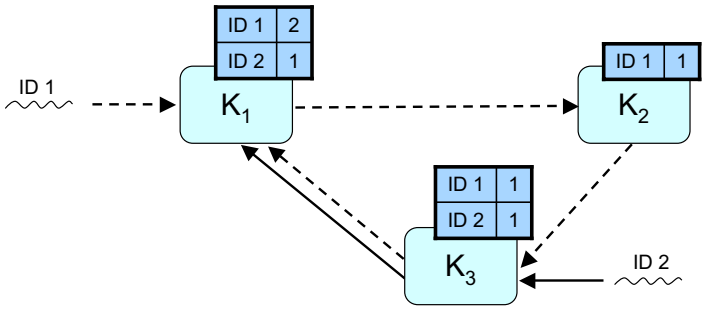


Abbildung 3.7: Rekonfigurationsalgorithmus: Offene Kommunikationstransaktionen

[2]Im Prinzip kann über die *exception frames* ein Stack-*Trace* ermittelt werden. Die notwendige Reifikation (*reification*) ist dabei aber sehr laufzeitintensiv.

Abbildung 3.7 zeigt ein Beispiel für die vom Rekonfigurationsalgorithmus erzeugten Hash-Tabellen. Für jeden Thread wird die aktuelle Aufruftiefe und damit die Anzahl der offenen Kommunikationstransaktionen pro Thread in jeder Kapsel gespeichert. Beim Blockieren der Threads kann nun ausgewertet werden, ob ein Thread auf einer Kapsel des *BlockSet* aktiv ist, und dann entsprechend blockiert werden. Ein großer Vorteil dieses Verfahrens ist, dass die Blockierung von Konnektoren parallel ausgeführt werden kann, das heißt bei allen beteiligten Kapseln kann gleichzeitig das Blockieren initiiert werden. Kehrt der letzte synchrone Blockierungsaufruf zurück, befindet sich die Anwendung im Ruhezustand und die Rekonfigurationsoperationen können ausgeführt werden. Eine Reihe von Details soll bei der genaueren Vorstellung des Algorithmus erörtert werden.
Zunächst wird eine Hilfsfunktion (Algorithmus 2) benötigt, welche ermittelt, ob noch ein Thread offene Kommunikationstransaktionen auf einer Kapsel des *BlockSet* besitzt. In diesem Zustand ist ein Endzustand des Algorithmus erreicht. Die Hilfsfunktion überprüft für jede Kapsel des *BlockSet*, ob in der Hash-Tabelle ein Eintrag für einen Thread existiert, der größer als 0 ist, also ein Thread eine offene Kommunikationstransaktion auf der jeweiligen Kapsel besitzt. Die Menge aller Threads einer Kapsel lässt sich einfach durch die Untersuchung aller Einträge der Hash-Tabellen bestimmen.

Algorithmus 2 Hilfsfunktion NoActiveThreadOnBlockSet

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

Für die Umsetzung des Rekonfigurationsalgorithmus sind zwei Mechanismen in die Kapseln (bzw. in alle Wurzelobjekte) zu integrieren. Zum einen müssen die beschriebenen Datenstrukturen aktualisiert und zum anderen muss während der Blockierungsphase ein entsprechender Synchronisationsmechanismus für die Anwendungs-Threads installiert werden. Der nachfolgende Teil des Algorithmus (Algorithmus 3) wird vor jeder Initiierung neuer Kommunikationstransaktion ausgeführt. Mit Hilfe der Variable *blockMode* kann die Blockierungsphase eingeleitet werden. Hat die Variable *blockMode* den Wert *false* ist die Blockphase nicht aktiv. Der Wert *true* bedeutet eine aktive Blockierungsphase. Der Wert der Variablen muss zur Aktivierung der Blockierung in allen Kapseln simultan geschrieben werden. Verfahren hierfür werden im anschließenden Implementierungskapitel vorgestellt. Im Fall der inaktiven Blockierungsphase wird einfach der Wert für den aktuellen Thread in der Hash-Tabelle (*ThreadList*) inkrementiert.
Bei aktiver Blockierungsphase wird zunächst überprüft, ob der Ruhezustand schon erreicht ist, also keine offenen Kommunikationstransaktionen auf Kapseln des *BlockSet* existieren. Ist dies der Fall, wird das Ereignisobjekt *quiescenceReached* signalisiert (Zeile 2,3), um den wartenden Rekonfigurations-Thread zu aktivieren. In jedem Fall wird der aktuelle Thread, falls dieser keine offenen Kommunikationstransaktionen auf den Kapseln des *BlockSet* besitzt - der Hash-Tabellenwert für diesen Thread in jeder Kapsel des *BlockSet* also 0 ist - blockiert (Zeile 6). An dieser Stelle findet sich der zentrale Wirkmechnismus des Algorithmus, da selektiv für jeden Thread entschieden

Algorithmus 3 Initiierung einer neuen Kommunikationstransaktion

```
1: if blockMode then
2:   if NoActiveThreadOnBlockSet() then
3:     SIGNAL(quiescenceReached)
4:   end if
5:   if ∄c ∈ BlockSet : c.ThreadList[id] > 0 then
6:     WAIT(blockEnd)
7:   end if
8: end if
9: ThreadList[id] ← ThreadList[id] + 1
```

wird, ob dieser blockiert werden kann oder die Verarbeitung fortgesetzt werden muss, um alle offenen Kommunikationstransaktion beenden zu können. Ist die Rekonfiguration abgeschlossen, wird die Verarbeitung des blockierten Threads durch Signalisierung des Ereignisobjekts *blockEnd* fortgesetzt.

Algorithmus 4 Abschluss einer Kommunikationstransaktion

```
1: ThreadList[id] ← ThreadList[id] − 1
2: if blockMode then
3:   if NoActiveThreadOnBlockSet() then
4:     SIGNAL(quiescenceReached)
5:   end if
6: end if
```

Die in Algorithmus 4 dargestellte Logik wird bei der Beendigung einer Kommunikationstransaktion ausgeführt. Ist die Blockierungsphase inaktiv, wird nur der Wert in der Hash-Tabelle für den aktuellen Thread dekrementiert. Andernfalls wird zusätzlich das Ereignisobjekt *quiescenceReached* signalisiert, falls der Ruhezustand genau durch dieses Dekrementieren des Zählers erreicht wurde. Dies geschieht wiederum, um den wartenden Rekonfigurations-Thread über das Erreichen des Ruhezustands zu informieren. Die Benachrichtigung des Rekonfigurations-Thread wird immer genau beim Verlassen der letzten offenen Kommunikationstransaktion durch den „letzten“ Thread veranlasst.

Schlussendlich zeigt der abschließende Teil von ReDAC (Algorithmus 5) das Auslösen der Blockierungsphase, welche im Kontext eines Konfigurationsmanagers parallel zur Anwendung ausgeführt wird. Zunächst werden die beiden verwendeten Ereignisobjekte (*blockEnd* und *quiescenceReached*) reinitialisiert (Zeile 1,2). Gefolgt davon wird die Variable *blockMode* in den Kapseln auf den Wert *true* gesetzt, was wie schon beschrieben atomar erfolgen muss. Anschließend wird überprüft, ob der Ruhezustand schon erreicht ist. Ist dies nicht der Fall, wird auf die Benachrichtigung durch den „letzten“ Thread gewartet (Zeile 6-8). Ist der Ruhezustand erreicht, können die Rekonfigurationsoperationen ausgeführt werden. Zum Abschluss wird die Blockierungsphase beendet und die Ausführung aller blockierten Threads, durch die Signalisierung des Ereignisobjekts *blockEnd*, fortgesetzt.

Algorithmus 5 Rekonfigurationsanforderung

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

3.3.3 Eigenschaften von ReDAC

Um einige Eigenschaften wie die Verklemmungsfreiheit (*deadlock*), den Fortschritt und am wichtigsten den wechselseitigen Ausschluss von Anwendungs- und Rekonfigurationsaktivitäten nachzuweisen, wurde der beschriebene Algorithmus als Petrinetz modelliert. Dies ermöglicht mit Hilfe modellbasierter Überprüfung, also einer vollständigen Fallunterscheidung möglicher Kontrollflüsse, den Nachweis dieser Eigenschaften.
Für die Modellierung als Petrinetz musste eine spezielle Konfiguration gewählt werden. Abbildung 3.8 zeigt die Modellierung von ReDAC an einer Anwendung mit einer Kapsel. Die einzige Aktivität der Anwendung führt wiederholt rekursive Aufrufe durch, wobei der Platz *InMethod* des Petrinetzes die Abarbeitung der Anwendungsfunktionalität symbolisiert. Die Rekursionstiefe wird durch die Anzahl der Marken in der Stelle *RecursionPool* bestimmt. Für die Modellierung wurde das Werkzeug TimeNET [German u. a. 1995] eingesetzt, welches an der Technischen Universität Berlin entwickelt wurde. TimeNET unterstützt die quantitative Analyse stochastischer Petrinetze, eignet sich aber auch für die Evaluation von Invarianten. Im oberen Teil von Abbildung 3.8 ist die anwendungsspezifische Logik dargestellt. Der einzige Anwendungs-Thread traversiert über die Plätze *Thread1Start*, *CallMethod*, *InMethod*, *RecursionPool* und *AftAdvice*. Der restliche Teil der Abbildung realisiert den Rekonfigurationsalgorithmus, welcher durch die Anwendung einfacher Regeln aus den vorangegangenen Beschreibungen des Algorithmus abgeleitet wurde. Am unteren Bildrand ist der initiale Kontrollfluss des Rekonfigurations-Thread mit einer Marke im Platz *ReconfThreadStart* abgebildet. Bei der Ausführung des Petrinetzes konkurrieren Anwendungs- und Rekonfigurations-Thread. Für den wechselseitigen Ausschluss muss die folgende Invariante gelten:

$$P\{\#InMethod + \#Reconfiguration > 1\} = 0 \tag{3.2}$$

In der TimeNET-Syntax bedeutet dies, dass die Wahrscheinlichkeit von mehr als einer Marke in den Plätzen *InMethod* und *Reconfiguration* gleich 0 ist. Es darf also niemals gleichzeitig eine Marke in diesen beiden Plätzen vorhanden sein, was dem wechselseitigen Ausschluss von Rekonfigurations- und Anwendungslogik entspricht. Mit Hilfe der Validierungsfunktion „Stationäre Analyse“ wurde diese Eigenschaft mit TimeNET nachgewiesen.

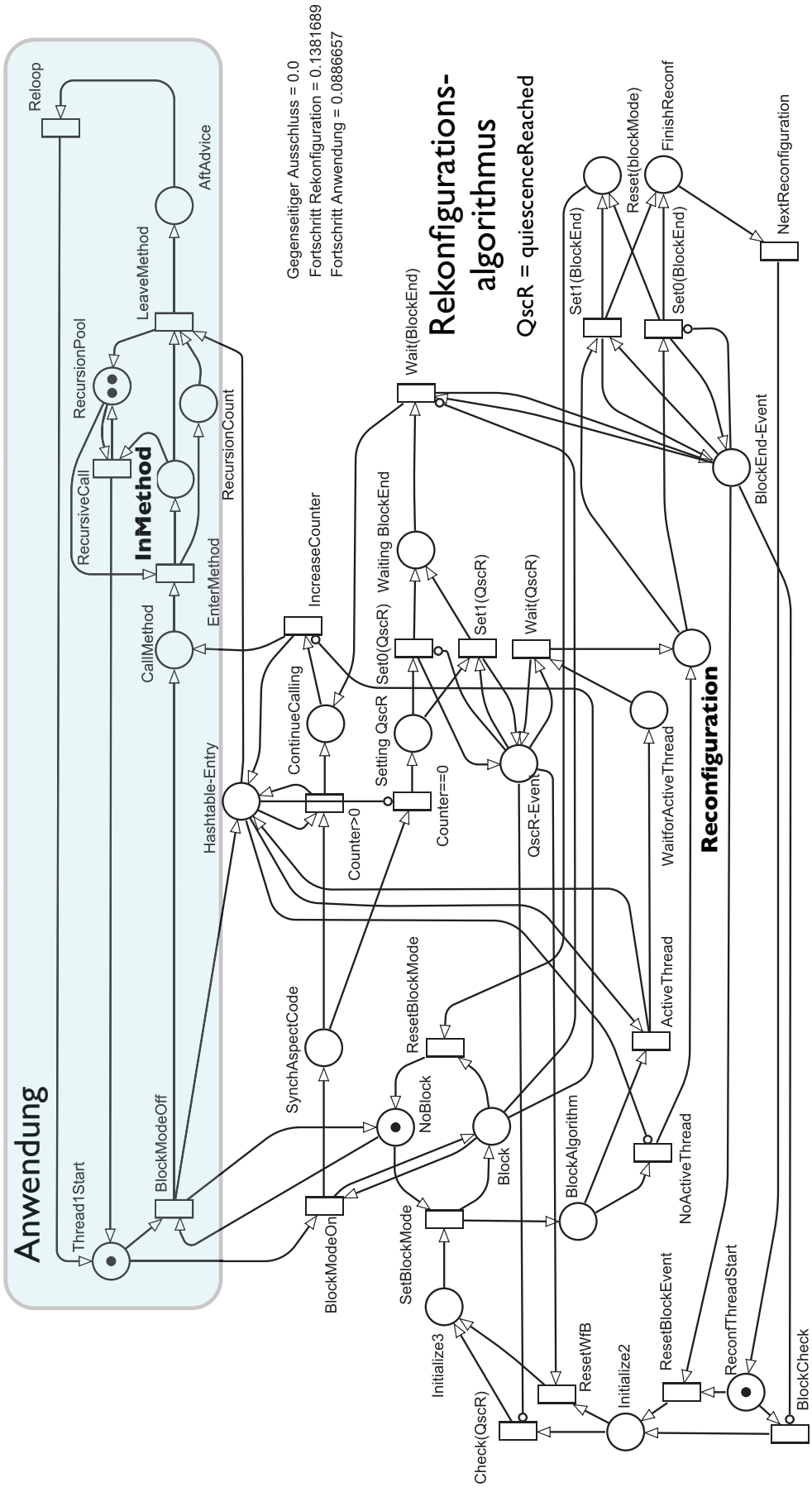


Abbildung 3.8: Modell Rekonfigurationsalgorithmus: 1 Kapsel, 1 Thread, Rekursion

Die Eigenschaften „Fortschritt“ und „Verklemmungsfreiheit“ wurden über die folgenden Invarianten nachgewiesen:

$$E\{\#InMethod\} > 0 \text{ bzw.} \tag{3.3}$$
$$E\{\#Reconfiguraiton\} > 0 \tag{3.4}$$

Mit dem Nachweis eines von 0 verschiedenen Erwartungswertes ist der Fortschritt von Anwendungs- und Rekonfigurations-Thread und damit die Verklemmungsfreiheit gezeigt, da dies bedeutet, dass immer wieder eine Marke auf den entsprechenden Plätzen existiert.
Um die Funktionsweise des Algorithmus auch für komplexere Anwendungskonfigurationen zu demonstrieren, wurde die Modellierung auf eine Konfiguration mit 2 Kapseln mit jeweils einem Anwendungs-Thread erweitert. Auch für dieses Modell wurden die zuvor besprochenen Eigenschaften nachgewiesen. Zur Berechnung der Invarianten musste auf Grund der Komplexität des Modells das Lösungsverfahren auf *iterativ* umgestellt werden.
Die Modellierung der Konfiguration mit 2 Kapseln hat gezeigt, dass durch Anwendung einfacher Regeln noch komplexere Konfigurationen modelliert werden können. Da die umgesetzten Modelle typische Einsatzfälle wie den Einsatz multipler Threads und mehrerer Kapseln abdecken, wurden keine weiteren Konfigurationen untersucht.

3.3.4 Optimierung bei bekannten BlockSets

Die Komplexität des vorgestellten Rekonfigurationsalgorithmus beim Blockieren der Konnektoren hängt linear von der Anzahl der Kapseln des *BlockSets* und der Anzahl aktiver Threads ab (siehe Hilfsfunktion Algorithms 2). Sind alle möglichen Konfigurationsübergänge bekannt und damit alle *BlockSets* für eine Anwendung gegeben, kann der vorgestellte Algorithmus bzgl. der Blockierungsdauer verbessert werden. Für jedes mögliche *BlockSet* könnte ein Zähler die Aktivität in den entsprechenden Kapseln speichern. Beim Blockieren von Konnektoren müsste, anstatt die Zähler aller Kapseln für jeden Thread zu überprüfen, nur ein Zähler betrachtet werden. Bei der Leistungsbewertung der Algorithmen (siehe Kapitel 7) hat sich aber gezeigt, dass die Blockierungsdauer für den vorgestellten Algorithmus akzeptabel ist.

3.3.5 Erweiterung bei Synchronisation zwischen Anwendungs-Threads

Existierende Rekonfigurationsansätze betrachten nicht die mögliche Synchronisation zwischen Anwendungs-Threads. In den meisten Fällen ist dies darauf zurückzuführen, dass keine *Multithread*-fähigen Systeme betrachtet werden. [Almeida u. a. 2001] fordern, dass keine Abhängigkeiten zwischen verschiedenen Threads innerhalb einer Anwendung existieren. In diesem Fall können die verwendeten Algorithmen und auch der im Rahmen dieser Arbeit vorgestellte Algorithmus in eine Verklemmung geraten. Ein Beispiel stellt Abbildung 3.9 dar.
In der Abbildung sind 3 Kapseln dargestellt. Die Kapseln K_2 und K_3 sollen aktualisiert werden. Im Beispiel ruft ein erster Thread (ID 1) eine Methode an K_1 auf und sperrt die Ressource R und wird im weiteren Verlauf eine Methode an K_2 aufrufen, und zwar mit der weiterhin gesperrten Ressource R. Ein Kontextwechsel aktiviert einen zweiten Thread (ID 2), der eine Methode an K_1 aus einer Methode von K_3 aufruft, innerhalb derer versucht wird R zu sperren. Thread ID 2 wird an dieser Stelle blockiert, da R

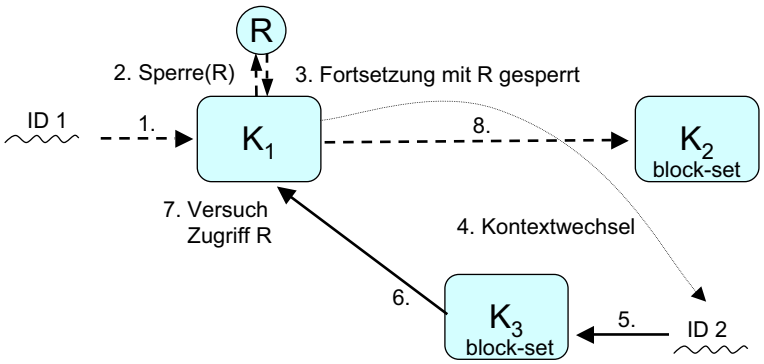


Abbildung 3.9: Anwendungsspezifische Synchronisation

noch von Thread ID 1 gesperrt ist. Eine in diesem Moment ausgelöste Rekonfigurationsanforderung verursacht eine Verklemmung. Der von Thread ID 1 initiierte Aufruf an K_2 würde blockiert, da Thread ID 1 keine offenen Kommunikationstransaktionen auf dem BlockSet besitzt. Die Ressource R würde nie freigegeben, wodurch Thread ID 2 die offene Kommunikationstransaktion in K_3 nie beenden könnte.
Mit ReDAC kann diese Problematik sehr leicht gelöst werden. Um die Synchronisation zwischen Anwendungs-Threads zu unterstützen, werden zusätzlich vor den Rekonfigurationsanforderungen alle Kapseln in das BlockSet aufgenommen, die dort nicht enthalten sind und im Verbindungsgraphen auf dem Pfad zwischen zwei Kapsel des BlockSets liegen.

3.3.6 Erweiterungen

Die Abgrenzungsforderung bei der Kapseldefinition bringt zunächst kleine Einschränkungen mit sich. Werden interne Objektreferenzen über Methodenaufrufe zu einer zweiten Kapsel übertragen, ist verboten, dass diese Referenz nach der Beendigung des Aufrufs im Objektgraphen der zweiten Kapsel weiter existiert. Für eine statische Überprüfbarkeit wäre die ausschließliche Verwendung von Werttypen als Schnittstellenparameter die Konsequenz. Prinzipiell lässt sich diese Einschränkung aber leicht aufheben. Werden Objektreferenzen bei Methodenaufrufen als Argumente übertragen, können übergebene Objektreferenzen transparent zu Wurzelobjekten werden und damit die Kapselgrenzen erhalten bleiben. Mit Hilfe der aspektorientierten Programmierung ist dies ohne zusätzlichen Aufwand für den Anwendungsentwickler möglich. Diese Erweiterung soll im Laufe der Arbeit allerdings nicht weiter diskutiert werden.
Im folgenden Abschnitt wird die Implementierung der bislang eingeführten Mechanismen für die Realisierung adaptiver Anwendungen im Rahmen des Adapt.Net-Projekts vorgestellt. Im Folgenden wird zunächst die Konfigurationsspezifikation in Form von XML-basierten Dokumenten untersucht, gefolgt von einer Beschreibung des zentralen Akteurs der Ausführungsplattform, dem Konfigurationsmanager, der die vorgestellten Rekonfigurationsverfahren realisiert. Bei der Implementierung der Rekonfigurationslogik wurden verschiedene Synchronisationsverfahren evaluiert, die im Anschluss vorgestellt werden. Die Generierung konfigurationsspezifischer Logik mit Hilfe der aspektorientierten Programmierung ist neben der Realisierung heterogener Anwendungen ein wichtiger Bestandteil dieses Abschnitts. Ein neues Programmiermodell für die Entwicklung adaptiver Anwendungen wird durch die Beschreibung der Ausführungsplattform eingeführt.

3.4 Konfiguration und ihre Beschreibung

Für die Implementierung der eingeführten Konzepte wurde zunächst eine persistente Repräsentation von Anwendungskonfigurationen umgesetzt. Eine speicherbare Darstellung einer Konfiguration wird auch als Konfigurationsbeschreibung bzw. **Konfigurationsspezifikation** bezeichnet. Die Spezifikation von Anwendungskonfigurationen wird vor allem mit Hinblick auf die dynamische Rekonfiguration erörtert. Neben der statischen Beschreibung der Anwendungsstruktur wird die Spezifikation von Änderungen während der Anwendungslaufzeit im Mittelpunkt der Untersuchung stehen.

3.4.1 Die Konfigurationsspezifikation

In den letzten Jahren beschäftigt sich vor allem der Forschungszweig der Softwarearchitekturen mit der abstrakten Organisation von Software und deren Beschreibung. Für die Beschreibung wurden die *Architectural Description Languages (ADLs)* entwickelt. Einige Beispiele hierfür sind ACME [Garlan u. a. 1997] oder Rapide [Luckham u. a. 1995]. Als Begründer des Gebiets der Softwarearchitektur gelten Shaw und Garlan, die das Buch „Software Architecture“ [Shaw und Garlan 1996] verfassten. Teilaspekte der Konfigurationsbeschreibung können aus der Softwarearchitektur übernommen werden. In der Softwarearchitektur werden oft verschiedene Sichtweisen auf ein Softwaresystem unterstützt. Unter anderem beschäftigt sich Softwarearchitektur mit der Systemstruktur. Dabei werden Komponenten und Konnektoren zueinander in Beziehung gesetzt und ihre Interaktionen beschrieben. Historisch betrachtet gehen aber die ADLs aus den *structural configuration languages* bzw. *modul interconnection languages* hervor die Anfang der 90er-Jahre entwickelt wurden. Nach einer kurzen Vorstellung existierender Beschreibungsansätze für Softwarestrukturen soll die im Rahmen des Adapt.Net-Projekts entwickelte Konfigurationsbeschreibungssprache vorgestellt werden.

3.4.2 Beschreibung von Konfigurationsänderungen

In der Literatur existieren zwei wesentliche Varianten, Änderungen an einer Anwendungskonfiguration zu beschreiben. Die am weitesten verbreitete ist die programmatische Beschreibung der Konfigurationsänderungen. Ein typischer Vertreter dieser Variante ist DACIA [Litiu und Prakash 2000], ein Framework, mit dem Funktionen zum Erzeugen, Entladen, Verbinden und Entkoppeln von Komponenten zur Verfügung gestellt werden. Änderungen einer Anwendungskonfiguration werden durch ein Rekonfigurationsskript implementiert, welche auf einem niedrigen Abstraktionsniveau die einzelnen Rekonfigurationsschritte repräsentieren. Der Entwickler der Rekonfigurationsskripte ist selbst für den Erhalt der Anwendungskonsistenz verantwortlich und muss die Reihenfolge der Rekonfigurationsschritte mit seinen erforderlichen Kenntnissen über die Anwendungsstruktur ermitteln. Der Erhalt der strukturellen Konsistenz kann hier nicht statisch vor der Laufzeit überprüft werden.
Abbildung 3.10 zeigt ein Beispiel eines Rekonfigurationsskripts in DACIA. Im Beispiel ist das Einfügen eines neuen Paares von Komponenten, welches die De- bzw. Komprimierung von strombasierten Daten realisiert, dargestellt. Die Reihenfolge für die Verbindung der einzelnen Komponenten muss vom Entwickler bestimmt werden. Die programmatische Beschreibung spezifiziert die Dynamik während einer dynamischen Rekonfiguration.

```
Engine.disconnectProcs(Source, 0)
Engine.connectProcs(Sender, 0, Compress, 0)
Engine.connectProcs(Decompress, 1, Receiver, 0)
Engine.connectProcs(Compress, 1, Decompress, 0)
```

Abbildung 3.10: DACIA Rekonfigurationsskript [Litiu und Prakash 2000]

Die Anfang der 90er-Jahre aufgekommenen Konfigurationssprachen *structural configuration languages* bzw. *modul interconnection languages* beschäftigten sich detailliert mit der deklarativen Beschreibung von Softwarestrukturen. Sie waren die ersten Ansätze, die Architektur von Software losgelöst von deren funktionaler Semantik zu beschreiben. Wichtige Vertreter für diese Beschreibungsvariante sind unter anderem Darwin [Magee u. a. 1995] und Conic [Magee u. a. 1989]. Die Verwendung der Konzepte Komponente (hier Kapsel) und Konnektor stammen aus dem Umfeld der Forschung auf diesem Gebiet. Für Darwin wurde zusätzlich auch eine graphische Notation für die Beschreibung von Softwarestrukturen entwickelt. Ein Beispiel hierfür ist in Abbildung 3.11 dargestellt.

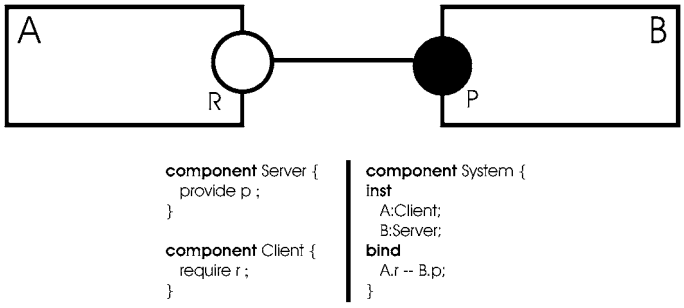


Abbildung 3.11: Konfigurationsspezifikation in Darwin (graphisch,textuell) [Magee u. a. 1995]

In der Abbildung ist eine Konfigurationsspezifikation in der textuellen und graphischen Darwin-Syntax dargestellt. Die Konfiguration enthält zwei Komponenten und einen Konnektor. Grundelemente der graphischen Syntax von Darwin wurden für die Repräsentation von Konfigurationen im Adapt.Net Konfigurationseditor übernommen. Die deklarative Konfigurationsbeschreibung spezifiziert nur die statische Struktur einer Konfiguration, während die Rekonfigurationsoperationen für die Überführung einer Konfiguration zu einer anderen nicht direkt spezifiziert werden.
In der vorliegenden Arbeit wurde eine neue deklarative Spezifikationssprache für die Speicherung von Konfigurationen entworfen, welche die durch den Einsatz moderner Komponentenplattformen gegebenen Anforderungen unterstützt. Änderungen werden durch die Angabe zweier Konfigurationen ausgedrückt. Die nötigen Rekonfigurationsoperationen werden von den Adapt.Net Laufzeitkomponenten (Konfigurationsmanager) berechnet und ausgeführt. Dies ermöglicht dem Entwickler adaptiver Anwendungen auf einer abstrakten Modellierungsebene zu arbeiten und erlaubt es, Konfigurationen für bestimmte Umgebungssituationen zu entwickeln, ohne über die Implementierung der Änderungen selbst nachdenken zu müssen. Die entworfene deklarative Beschreibung von Konfigurationen wird im folgenden Text kurz vorgestellt. Detaillierte Informationen finden sich in [Rasche und Polze 2002].

Die Konfigurationsspezifikationssprache in Adapt.Net basiert auf der *eXtensible Markup Language (XML)*, die sich durch eine einfache Verarbeitung und weitreichende Standardisierungen auszeichnet. Für die Konfigurationsspezifikationsdokumente wurde eine *Document Type Description (DTD)* entworfen, die den Inhalt gültiger Konfigurationsspezifikationsdokumente definiert.
Eine Konfigurationsbeschreibung enthält zunächst XML-Elemente mit dem Namen *configuration*, denen das XML-Attribut *configurationsname* zugeordnet ist. Kindelemente einer *configuration* können die XML-Elemente *capsule* und *connector* sein. *capsule*-Elemente können die folgenden XML-Attribute besitzen:

- *name* - Name der Kapsel
- *loadStrategy* - Art der Kapselerzeugung (siehe Kapselkonfigurator)
- *assembly* - URL der Assembly, aus der die Kapsel erzeugt werden soll
- *assemblyVersion* - Version der Assembly in Form eines Assembly-Namens
- *location* - Name des Rechners, auf dem die Kapsel ausgeführt werden soll
- *mainType* -Typname der Hauptklasse der Kapsel

XML-Kindelemente einer Kapsel können die XML-Elemente *port* und *parameter* sein. Ein *Port* repräsentiert einen Verbindungspunkt einer Kapsel. Einem *Port* sind als Attribute ein Name, der Typ der Schnittstelle und die Art des Verbindungspunkts (*OUT* für die Quelle eines Konnektors und *IN* für die Senke) zugeordnet. Einem *Parameter* sind ein Namensattribut, der Wert des Parameters (*value*) sowie sein Typ zugeordnet. Ein *Connector*-XML-Element besitzt im Wesentlichen Attribute zur Definition von Quell- und Zielpunkt eines Konnktors. Diese umfassen die Namen der Quellkapsel, des Quellattributs, der Senkenkapsel sowie der Zielschnittstelle und zusätzlich einen Konnektortyp. Ein Auszug eines kurzen Beispiels für die XML-basierte Konfigurationsbeschreibung ist in Abbildung 3.12 dargestellt. In der Konfiguration befinden sich zwei Kapseln und ein Konnektor. Im Beispiel handelt es sich um eine heterogene Anwendung mit einer CLI- und einer Java-basierenden CORBA-Kapsel. Mehr Details zur Integration heterogener Komponentenplattformen findet sich in Abschnitt 3.5.5.

```
<configuration configurationname="c1">
    <capsule name="Viewer" loadStrategy="OBJECT" assembly="
       MessageView.dll" assemblyVersion="..." access="" mainType="
       ConfiguredObjectProxy.MessageView" location="localhost">
        <port name="m_source" type="OUT" vartype="IMessageSource" />
        <port name="default" type="IN" vartype="System.Object" />
    </capsule>
    <capsule name="Source" loadStrategy="CorbaCapsule" assembly="
       MessageView.dll" assemblyVersion="..." access="" mainType="
       ConfiguredObjectProxy.MessageSource" location="localhost">
        <port name="default" type="IN" vartype="System.Object" />
        <parameter name="encoding" value="UTF8" type="string"/>
    </capsule>
    <connector sourcecapsule="Viewer" sourceattribute="m_source"
       sinkcapsule="Source" sinkinterface="default" type="IIOP" />
</configuration>
```

Abbildung 3.12: XML-basierte Konfigurationsbeschreibung (Auszug)

3.5 Eine Ausführungsplattform für adaptive Anwendungen

Die Adapt.Net-Infrastruktur mit dem Namen CoFRA (*configuration framework*) ist eine Sammlung von Bibliotheken, Laufzeitkomponenten und Werkzeugen für die Entwicklung und Ausführung adaptiver Anwendungen in einem verteilten System.

Die zentrale Komponente der Adapt.Net-Infrastruktur ist der Konfigurationsmanager, der die Verwaltung von Kapseln sowie die Umschaltung zwischen alternativen Anwendungskonfigurationen mit den im vorangegangenen Abschnitt beschriebenen Rekonfigurationsalgorithmen realisiert. Ein Exemplar des Konfigurationsmanagers wird auf jedem Knoten des verteilten Systems ausgeführt und ist dort lokal für die Verwaltung der Kapseln verantwortlich. Ausgehend von der vorgestellten Konfigurationsbeschreibung sind die Konfigurationsmanager in der Lage, Adapt.Net-Anwendungen zu starten und zu verwalten. Hierzu müssen im Falle verteilter Anwendungen Assemblies auf den entfernten Rechnern verteilt und installiert werden. Dieser Vorgang wird als **Deployment** bezeichnet. Die Konfigurationsmanager kommunizieren dabei über eine entsprechende Schnittstelle, die im weiteren Verlauf dieses Kapitels ausführlich vorgestellt wird.

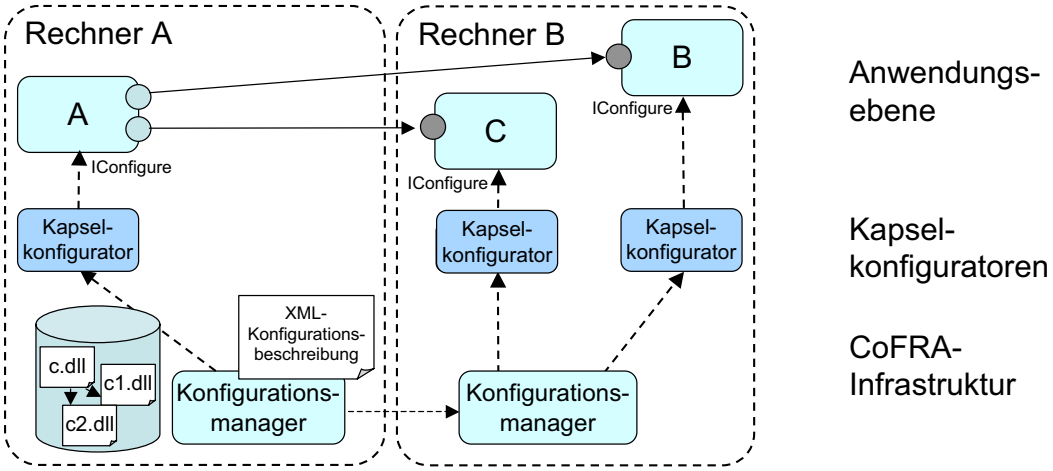


Abbildung 3.13: Die CoFRA-Infrastruktur

Abbildung 3.13 zeigt die Konfiguration einer verteilten Adapt.Net-Anwendung, welche auf zwei Rechnern ausgeführt wird und aus drei Kapseln besteht. Jede Adapt.Net-Anwendung wird auf genau einem Rechner gestartet. Die Assemblies der Kapseln und die Konfigurationsbeschreibung liegen auf dem Rechner bereit, auf dem die Anwendung gestartet wurde. Wenn eine Kapsel auf einem entfernten Rechner erzeugt werden soll, werden die entsprechenden Assemblies übertragen. Die Verwaltung der einzelnen Kapseln erfolgt über Kapselkonfiguratoren, die im weiteren Verlauf dieses Kapitels vorgestellt werden.

Die Verarbeitung der Konfigurationsbeschreibung sowie die Erzeugung und Verwaltung der Kapseln einer Anwendung wird immer von einem Konfigurationsmanager ausgeführt. Dieser Konfigurationsmanager wird als **primärer Konfigurationsmanager** bezeichnet, alle anderen beteiligten Konfigurationsmanager als **sekundär**. Die einzelnen Akteure und Mechanismen der Adapt.Net-Infrastruktur werden im Folgenden vorgestellt.

3.5.1 Der Konfigurationsmanager

Der Konfigurationsmanager verarbeitet die XML-basierenden Konfigurationsbeschreibungen. Diese werden in eine interne Hauptspeicherrepräsentation eingelesen, die während der Rekonfigurationsphase effektiv verarbeitet werden kann. Dieser Schritt ist notwendig, da die Verarbeitung der XML-Dokumente im Vergleich zur Rekonfigurationszeit lange dauert. Zur Verarbeitung der XML-Dokumente wurde ein *Document Object Model (DOM)*-Parser aus der .NET-Klassenbibliothek verwendet.
Kapseln und Konnektoren wurden als separate Klassen implementiert, die als Attribute ihre Eigenschaften enthalten. Ein Konfigurationsobjekt enthält Datenstrukturen, die auf Kapsel- und Konnektorobjekte verweisen. Der Konfigurationsmanager stellt Methoden zum Laden der XML-Dokumente bereit und überführt diese in die interne Darstellung. Abbildung 3.14 zeigt ein UML-Diagramm der internen Repräsentation einer Anwendungskonfiguration.

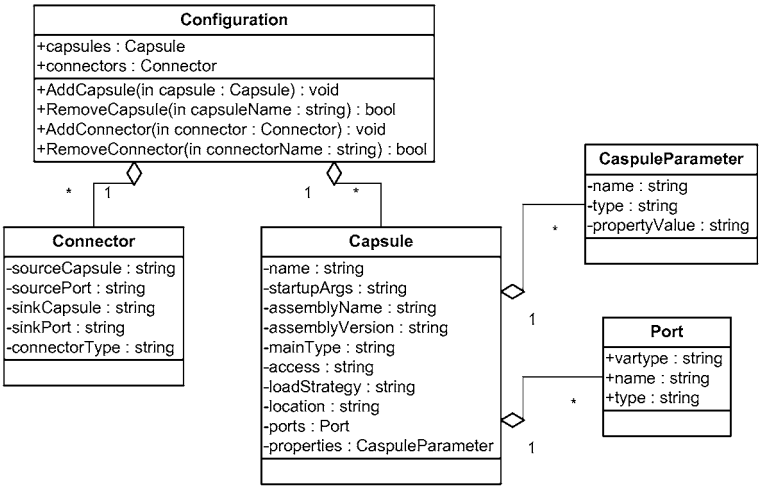


Abbildung 3.14: Interne Verarbeitung von Konfigurationen

Die Methode *ReconfigurefromCurrentConfiguration* enthält die zentrale Funktionalität des Konfigurationsmanagers. Sie implementiert die beschriebenen Rekonfigurationsalgorithmen und ermöglicht damit Änderungen an der Konfiguration der verwalteten Anwendungen. Die Methode kann entweder durch den Anwender selbst, z.B. durch die Benutzung der im Adapt.Net-Projekt entwickelten Konfigurationskonsole, oder automatisiert durch weitere Adapt.Net-Infrastruktur-Komponenten (*AdaptationEngine*) aufgerufen werden. Der Methode muss als Argument ein Exemplar einer *Configuration* übergeben werden.
Die Methode *ReconfigurefromCurrentConfiguration* rekonfiguriert die Anwendung von der aktuellen Konfiguration, die in der internen Variable *currentConfiguration* gespeichert ist, in die übergebene Konfiguration. Nach dem Start des Konfigurationsmanagers ist die leere Konfiguration aktiviert. Sie enthält keine Kapseln und Konnektoren. Zunächst soll die Implementierung dieser Methode in groben Einzelschritten beschrieben werden.

Schritt 1 In der Methode *ReconfigurefromCurrentConfiguration* werden zunächst die alte und neue Konfiguration analysiert und miteinander verglichen. Dabei werden die Mengen der neuen, entfernten, aktualisierten, migrierten Kapseln sowie Kapseln mit

geänderten Parametern ermittelt. Die Vereinigung dieser Mengen wurde als Menge der Rekonfigurationselemente eingeführt. Die ermittelten Kapseln werden in separaten Listen gespeichert. Eine Kapsel kann in mehr als einer Liste enthalten sein, z.B. migriert und aktualisiert. Die Bestimmung neuer und entfernter Kapseln beruht auf dem Namen der Kapseln als eindeutigem Identifikator.

Schritt 2 Im nächsten Schritt folgt ausgehend von den ermittelten Änderungen der Kapseln die Berechnung der zu blockierenden Konnektoren sowie ggf. die Reihenfolge in der das Blockieren auszuführen ist. Details über die Berechnung der Konnektoren und das Blockieren folgen in einem separaten Abschnitt (3.6 Erreichen des rekonfigurierbaren Zustands).

Schritt 3 Jede Kapsel einer Adapt.Net-Anwendung implementiert die Schnittstelle *IConfigure*, welche Methoden für deren Verwaltung enthält. Die Methode *BlockConnection* dieser Schnittstelle realisiert das Blockieren von Konnektoren. Mit dem Aufruf dieser Methode wird die sogenannte Unterbrechungsphase (engl. *blackout phase*) eingeleitet, in der beteiligte Kapseln potentiell keine Methodenaufrufe mehr verarbeiten. Es ist ein wichtiges Optimierungskriterium bei interaktiven und zeitkritischen Anwendungen, diesen Zeitraum so kurz wie möglich zu halten. Das Erzeugen neuer Kapseln ist ein relativ zeitaufwendiger Schritt und wird deshalb vor der Initiierung der Unterbrechungsphase ausgeführt.

Der Lebenszyklus der Kapseln wird in Adapt.Net durch dynamisch nachladbare Kapselkonfiguratoren übernommen (siehe Abschnitt 3.5.5). Für das Erzeugen von Kapseln stellen diese Module eine Fabrikmethode bereit. Für jede neue Kapsel wird die *Create*-Methode des entsprechenden Kapselkonfigurators aufgerufen, der an Hand der Kapselart aus der Konfigurationsbeschreibung ermittelt wurde. Vor dem Erzeugen der Kapseln werden die Assemblies durch den primären Konfigurationsmanager ggf. an einen entfernten Rechner übertragen. Die Strategie (vgl. Entwurfsmuster Strategie) [Gamma u. a. 1995] für das Laden von Kapseln wird durch den Kapselkonfigurator realisiert. Eine verwendete Strategie ist die Erzeugung von Kapseln durch die Verwendung des *new*-Operators der virtuellen Laufzeitumgebung mit einem ausgewiesenen Typ als Argument.

Schritt 4 In diesem Schritt wird die Blockierung von Konnektoren durchgeführt. Je nach verwendetem Algorithmus wird die Blockierung in der zuvor ermittelten Reihenfolge ausgeführt oder parallel die Blockierung aller beteiligten Rekonfigurationselemente initiiert. Am Ende dieses Schrittes befindet sich die Anwendung in einem rekonfigurierbaren Zustand.

Schritt 5 Im rekonfigurierbaren Zustand kann zunächst der Zustandstransfer von migrierten und aktualisierten Kapseln erfolgen. Dieser Schritt wird ausführlich in den Abschnitten Migration (3.7) und Dynamische Aktualisierung (4) vorgestellt. Im Falle migrierter Kapseln wird unter Umständen schon in Schritt 3 ein neues Exemplar der Kapsel erzeugt, um die teure Operation des Assembly-Transfers und das Erzeugen neuer Betriebssystemprozesse (in bestimmten Kapselkonfiguratoren) während der Unterbrechungsphase zu vermeiden.

Schritt 6 In diesem Schritt werden geänderte Parameter der Kapseln über die Konfigurationsschnittstelle *IConfigure* gesetzt. Dieser Schritt muss nach dem Zustandstransfer erfolgen, da Parameter eine Teilmenge des Zustands sind, weil sie durch ausgewiesene Attribute eines Objekts der Kapseln repräsentiert werden.

Schritt 7 Die Konnektoren zwischen den Kapseln werden entsprechend der neuen Konfiguration erstellt. Dies ist der letzte Schritt innerhalb der Unterbrechungsphase. Es existieren eine Reihe unterschiedlicher Konnektortypen, welche die Art der Kommunikation zwischen den Kapseln bestimmt. Je nach Konnektortyp unterscheidet sich die Erstellung von Konnektoren. Details über die implementierten Konnektortypen folgen im Verlauf des Kapitels.

Schritt 8 Die Verarbeitung der von der dynamischen Rekonfiguration betroffenen Kapseln wird durch den Aufruf der *Start*-Methode der Konfigurationsschnittstelle *IConfigure* angestoßen. Die Reihenfolge des Starts von Kapseln spielt keine Rolle, da keine Verklemmungen entstehen können und die Ausführung der *Start*-Methode keine potentiell blockierenden Synchronisationsaufrufe enthält. Außerdem befindet sich die Anwendung zu diesem Zeitpunkt schon wieder in einem strukturell integren Zustand. Nach vollständiger Ausführung dieses Schritts wird die Anwendung in der neuen Konfiguration ausgeführt.

Schritt 9 Schlussendlich werden die aus der alten Konfiguration entfernten Kapseln entladen. Dieser Schritt wird parallel zur schon in der neuen Konfiguration laufenden Anwendung ausgeführt. Nach dem Entfernen der Kapseln kehrt die Methode *ReconfigurefromCurrentConfiguration* zum Rufer zurück. Potentiell kann dieser letzte Schritt mit einer niedrigeren Betriebssystempriorität ausgeführt werden, um die Anwendung nicht zu beeinflussen.
Nachdem nun die prinzipielle Arbeitsweise des Konfigurationsmanagers beschrieben wurde, soll im nächsten Abschnitt das Deployment von Assemblies bei der entfernten Erzeugung von Assemblies erläutert werden.

3.5.2 Deployment

Kapseln werden durch die Erzeugung von Exemplaren ausgewiesener Typen erzeugt, die in Assemblies gespeichert sind. Assemblies enthalten Bytecode und Metadaten der in ihr definierten Typen. Diese können Anhängigkeiten zu anderen Typen besitzen, die in weiteren Assemblies definiert sein können. Neben der Konfigurationsbeschreibung besteht eine Anwendung auch aus den von ihr benötigten Assemblies. Die Konfigurationsbeschreibung und die Position der Assemblies sind zunächst unabhängig voneinander, da in der Konfigurationsbeschreibung beliebige Positionen für Assemblies spezifiziert werden können. Bei der Installation einer Anwendung auf einem Rechner müssen Konfigurationsbeschreibung und Assemblies kompatibel abgelegt werden. Im Folgenden soll exemplarisch das Deployment in der CLI beschrieben werden. Das verwendete Verfahren ist dabei leicht auf weitere Komponentenplattformen übertragbar und unterscheidet sich im Wesentlichen in den verwendeten Ladefunktionen der jeweiligen Klassenbibliothek. Für die Java-Plattform können Details der Realisierung in [Rasche u. a. 2005a] nachgelesen werden.

Assembly-Cache

Das Laden von Assemblies in der CLI wurde in Adapt.Net innerhalb der Klasse *AssemblyCache* implementiert. Diese nutzt die Bibliotheksfunktionen *Assembly.Load** der CLI-Klassenbibliothek. Die Kapselkonfiguratoren benutzen die Funktionalität des *AssemblyCache* zum Laden der erforderlichen Assemblies. Beim Erzeugen einer Kapsel wird zunächst die in der Konfigurationsbeschreibung spezifizierte Assembly geladen. Mögliche Abhängigkeiten werden durch eine Ereignisbehandlungsroutine geladen, die beim Auftreten des *AssemblyResolveEvents* aufgerufen wird. Die CLI wurde entsprechend konfiguriert, dieses Ereignis bei jeder abhängigen Assembly auszulösen. Die Ereignisbehandlungsroutine lädt die abhängigen Assemblies aus dem Verzeichnis der initialen Assembly. Auf Wunsch des Anwendungsentwicklers können einmal geladene Assemblies im Hauptspeicher zwischengespeichert werden (*Cache*) und müssen bei einer erneuten Benutzung nicht mehr von der Festplatte gelesen werden. Dies ist bei adaptiven Anwendungen ein häufig auftretender Fall, da zwischen zwei oder mehr Konfigurationen wiederkehrend umgeschaltet wird, eine Konfiguration also mehr als einmal geladen wird.

Im Gegensatz zur eingebauten Auflösung von referenzierten Assemblies in der CLI, die im Wesentlichen auf der Suche im lokalen Verzeichnis beruht, hat der *AssemblyCache* den Vorteil, dass zum einen die Cache-Funktionalität als Optimierung bei der Aktivierung von Konfigurationen zur Verfügung steht und zum anderen bei der entfernten Erzeugung von Kapseln diese direkt aus dem Hauptspeicher geladen werden können und ein Umweg über die Festplatte nicht notwendig ist.

Verteilte Anwendungen

Im Falle einer entfernten Erzeugung einer Kapsel überträgt der primäre Konfigurationsmanager die benötigten Assemblies an den zuständigen sekundären Konfigurationsmanager. Hierfür berechnet der primäre Konfigurationsmanager die Abhängigkeitshülle (HUL) der Hauptassembly der Kapsel (siehe Abschnitt 3.1.1).

Bei der Berechnung der Abhängigkeitshülle einer Assembly werden zunächst alle Abhängigkeiten der Haupt-Assembly übernommen. Dann werden wiederum die Abhängigkeiten dieser Assemblies untersucht und evtl. übernommen. Dies wird so lange fortgesetzt, bis die ermittelten Abhängigkeiten nur noch Assemblies der schon berechneten Abhängigkeitshülle sind - die Menge also abgeschlossen ist. Die Berechnung der Abhängigkeitshülle wurde in einer rekursiven Methode implementiert. Durch eine entsprechende Zyklenerkennung und die vorausgesetzte Endlichkeit der Menge der Assemblies terminiert der Algorithmus.

Die ermittelten Assemblies werden an den sekundären Konfigurationsmanager übertragen. Dort werden sie zunächst im Assembly-Cache zwischengespeichert. Das Laden der Kapseln wird vom primären Konfigurationsmanager über die Inter-Konfigurationmanager Schnittstelle *IConfigurationManager* ausgelöst. Die Assemblies werden als Binärdaten in den Hauptspeicher geladen und in Form eines Bytefeldes übertragen. Dann wird die Kapsel aus den zuvor übertragenen Assemblies aus dem Speicher erzeugt. Mit Hilfe der Methode *Assembly.Load(byte[],...)* kann die Kapsel direkt aus dem Speicher des Assembly-Caches geladen werden. Abhängigkeiten werden wiederum über die Ereignisbehandlungsroutine des *AssemlbyResolveEvents* aus den übertragenen Assemblies direkt aus dem Speicher aufgelöst, die auch zuvor übertragen wurden. Die

Konfiguration der neu geladenen Kapseln wird direkt über die entsprechenden Kapselkonfiguratoren realisiert.

Sicherheit in Adapt.Net

Für das verteilte Deployment mussten entsprechende Sicherheitsmechanismen integriert werden, damit nicht von Unberechtigten Schadprogramme über die Deployment-Schnittstelle des Konfigurationsmanagers ausgeführt werden können. Hierfür werden die Sicherheitsmechanismen der .NET-Plattform genutzt. Anhand einer *Evidence* (Herkunftsbeweis) kann die Ausführungserlaubnis für jede beliebige Assembly konfiguriert werden. Die *Evidence* einer Assembly kann ihr Name, der Ort im Dateisystem oder ihr *Strongname* sein. Der *Strongname* einer Assembly kann eine digitale Signatur enthalten, die belegt, dass die Assembly von einer autorisierten Person bzw. einem Unternehmen erstellt und mit einem Zertifikat (privater Schlüssel) unterzeichnet wurde. Der Entwickler einer Adapt.Net-Anwendung unterzeichnet alle Assemblies der Anwendung mit seinem Zertifikat. Der Nutzer konfiguriert bei der Installation der Anwendung bzw. des Konfigurationsmanagers die .NET Sicherheitsrichtlinien so, dass Assemblies, die mit dem entsprechenden Zertifikat signiert wurden, als vertrauenswürdig geladen werden.

3.5.3 Konfigurierbare Kapseln

Die Konfiguration einzelner Kapseln erfolgt über die Konfigurationsschnittstelle *IConfigure*, die jede Kapsel einer Adapt.Net-Anwendung implementieren muss. Der Konfigurationsmanager interagiert über diese Schnittstelle mit den einzelnen Kapseln. In diesem Abschnitt sollen zunächst die einzelnen Methoden dieser Konfigurationsschnittstelle vorgestellt werden.
Mit Hilfe der Methoden **GetParameter** und **SetParameter** können Kapselparameter manipuliert werden. Die Werte der Parameter werden in serialisierter Form als Zeichenkette (*string*) übergeben, wie sie im beschreibenden Konfigurationsdokument vorliegen. Die konfigurationsspezifische Logik der Kapseln muss die Werte der Parameter entsprechend der vorliegenden Typen deserialisieren. Die zugrunde liegenden Typen sind in der aktuellen Implementierung immer primitiv, obwohl nichts gegen die Verwendung komplexer Datentypen spricht. Jeder Parameter besitzt einen Namen, der das Attribut der Hauptklasse einer Kapsel bestimmt, welches den Parameter repräsentiert.
Die Methode **Connect** wird für die Erzeugung von Konnektoren zwischen den Kapseln benötigt. Jeder Konnektor einer Kapsel wird einem Attribut ihres Hauptobjekts zugeordnet. Beim Erstellen eines Konnektors wird immer die *Connect*-Methode der Konfigurationsschnittstelle der Quellkapsel aufgerufen, die für den Konnektoraufbau verantwortlich ist. Das erste Argument der Methode *Connect* enthält den Namen des Quellattributs. Ein weiteres Argument beschreibt den Konnektortyp. Für die Erstellung heterogener Anwendungen kann an dieser Stelle die Kommunikations-Middleware selektiert werden. Es existiert eine Reihe von Implementierungen objektbasierter Middleware, die das ORPC-basierte Kommunikationsschema unterstützen. Unter ihnen finden sich CORBA, DCOM, .NET Remoting, Java RMI, SOAP und viele weitere, die jeweils als Konnektortypen in Frage kommen.
Schlussendlich erfordert die gewählte Kommunikations-Middleware verschiedene Argumente für die Verbindungserstellung. Diese werden der *Connect*-Methode als Feld

unbestimmter Daten übergeben (*object[]*). Typ und Semantik der einzelnen Argumente wird durch die Implementierung des Konnektortyps definiert, die zum einen im Konfigurationscode der Kapseln, zum anderen im Konfigurationsmanager verankert ist.

Während der dynamischen Rekonfiguration einer Anwendung wird diese in einen rekonfigurierbaren Zustand überführt. Wie bereits beschrieben, basiert der hier verwendete Ansatz auf dem Blockieren von Konnektoren zwischen den Kapseln. Die Methode **BlockConnection** muss sicherstellen, dass alle Kommunikationstransaktionen über einen gegebenen Konnektor beendet werden und alle in Zukunft initiierten Kommunikationstransaktionen bis zum nächsten Aufruf der *Start*-Methode blockiert werden. Der jeweilige Konnektor wird über ein Argument *BlockConnection*-Methode, welches den Namen des Quellattributs enthält, identifiziert. Die Methode *BlockConnection* arbeitet synchron, das heißt sie kehrt genau nach Erreichen des beschriebenen Zustands zum Rufer, dem Konfigurationsmanager, zurück. Die Rückkehr der Methode signalisiert dem Konfigurationsmanager das erfolgreiche Blockieren eines Konnektors.

Mit Hilfe der Methode **Start** wird die Verarbeitung einer Kapsel aktiviert. Beim initialen Aufruf der *Start*-Methode wird die Verarbeitung der kapseleigenen Threads initiiert. Die Erzeugung der Threads sollte aus Performance-Gründen während der Kapselinitialisierung erfolgen. Die Terminierung der Threads einer Kapsel kann innerhalb der **Finalize**-Methode implementiert werden, die beim Entladen einer Kapsel aufgerufen wird.

Zunächst wurde die fehlerfreie Ausführung von Kapseln angenommen. Einige Fehler lassen sich durch eine Erweiterung während der dynamischen Rekonfiguration behandeln. Der Ausfall von Rechnern, auf denen neue Kapseln erzeugt werden sollen, und Fehler beim Starten von Kapseln können durch ein Rollback von Rekonfigurationsaktionen behandelt werden. Tritt ein Fehler während der Rekonfigurationsphase auf, kann durch den Aufruf der **Rollback**-Methode jeder Kapsel der alten Konfiguration diese wieder aktiviert werden. Jede Kapsel muss deshalb beim Setzen von Parametern und der Erzeugung von Konnektoren die alten Werte der entsprechenden Attribute, die Parameter und Konnektorquellpunkte repräsentieren, speichern und beim Rollback wiederherstellen. Dies kann z.B. mit Hilfe der Reflexionsfunktionen und einer Hash-Tabelle mit dem Attributnamen als Schlüssel und dem alten Wert als Eintrag sehr effizient realisiert werden. Die entsprechenden Strukturen müssen bei jedem ersten Aufruf von *BlockConnection*, *SetParameter* oder *Connect* nach einem Aufruf der *Start*-Methode gelöscht werden. Besondere Bedeutung kommt der Rollback-Funktionalität bei der *Einhaltung von Zeitschranken* zu. Da die Dauer des Blockierens von Konnektoren von der Dauer der Methodenaufrufe und vor allem vom aktuellen Bearbeitungszustand aktiver Kommunikationstransaktionen abhängt, kann die gesamte Unterbrechungsdauer je nach Inititiierungszeitpunkt des Blockierens variieren. Deshalb ist es möglich, eine Rekonfiguration mehrmals zu „versuchen“ und dabei unterschiedliche Unterbrechungsdauern zu erreichen. Nach Überschreitung einer bestimmten Dauer könnte durch Auslösen eines Rollbacks eine alte Konfiguration aktiviert werden und erst bei einem späteren Versuch die Rekonfiguration mit den geforderten Zeitvorgaben zum Erfolg führen. Es wird davon ausgegangen, dass eine zuvor funktionierende Konfiguration fehlerfrei wieder geladen werden kann, ein Rollback also niemals rekursiv ausgeführt werden muss.

Während des rekonfigurierbaren Zustands wird für alle verbleibenden Kapseln die Methode **Initialize** aufgerufen. Für zu entladende Kapseln wird die **Finalize**-Methode

aufgerufen. Innerhalb dieser Methode kann anwendungsspezifisch die Erhaltung von Invarianten realisiert werden. Die zeitaufwendige Initialisierung der Kapsellogik sollte über entsprechende Mechanismen der Laufzeitumgebung realisiert werden, z.B. Konstruktoren, Destruktoren. Diese werden dann bei der Erzeugung der Kapseln außerhalb der Unterbrechungsphase ausgeführt. Eine Problematik bei der Wiederherstellung von Invarianten ist, dass hierfür potentiell auf den Zustand aller Kapseln zugegriffen werden muss. In diesem Fall muss ein Mechanismus gefunden werden, mit dem der Entwickler spezifizieren kann, der Zustand welcher Kapseln benötigt wird. Diese müssen dann bei der dynamischen Rekonfiguration auch in die betroffenen Rekonfigurationselemente aufgenommen werden. An dieser Stelle soll die Annahme getroffen werden, dass die Wiederherstellung der Invarianten immer nur den Zustand der einen Kapsel betrifft. Erweiterungen der Konfigurationsspezifikation sollen im Rahmen dieser Arbeit nicht untersucht werden.

3.5.4 Der Konfigurationsaspekt

Teile der konfigurationsspezifischen Logik, welche durch die Implementierung der Konfigurationsschnittstelle in jede Kapsel integriert werden muss, sind über den funktionalen Teil der Kapsellogik verstreut. Der Entwickler konfigurierbarer Kapseln muss die Logik für die Synchronisation für das Blockieren von Verbindungen, den Zugriff auf Parameter und Verbindungsendpunkte usw. in den funktionalen Code integrieren. Die Konfiguration von Kapseln sollte aber eigentlich nicht bei der Entwicklung ihrer Funktionalität betrachtet werden müssen, da sie ein nichtfunktionaler Bestandteil der Anwendung ist. Die Kernidee der aspektorientierten Programmierung ist es, genau diese nichtfunktionalen Belange losgelöst von der eigentlichen Anwendungslogik zu implementieren.
Im Folgenden soll nun exemplarisch für die CLI diskutiert werden, wie die Konfiguration von Kapseln als separater Aspekt implementiert werden kann. Dabei wird eine generische Implementierung der Konfigurationsschnittstelle *IConfigure* vorgestellt, die mit Hilfe von Werkzeugen der aspektorientierten Programmierung in die Kapsellogik integriert werden kann.
Die Schnittstelle *IConfigure* wird als *Introduction* an die Hauptklasse einer Kapsel verwoben. Als *Introduction* wird das Hinzufügen neuer Schnittstellen an eine Klasse bezeichnet. Der statische Weber Loom.Net erzeugt bei der Verwebung eine Ableitung der Hauptklasse, die zusätzlich die Schnittstelle *IConfigure* implementiert. Die Interaktion zwischen funktionalem Anwendungscode und Konfigurationslogik wurde durch Reflexion realisiert.
Abbildung 3.15 zeigt einen ersten Ausschnitt der Implementierung des Konfigurationsaspekts für den statischen Aspektweber Loom.Net. Im Verlauf dieser Arbeit wurden sämtliche entwickelten Aspekte auch für den dynamischen Aspektweber Rapier-Loom.Net verwirklicht, der Vorteile bei der Programmierung bietet, da Aspekte als Klassen in einer beliebigen .NET-Sprache implementiert werden können. Der statische Weber hatte während der Entstehungsphase dieser Arbeit Laufzeitvorteile, da für den Zugriff auf den Aspektkontext kein zusätzlicher TLS-Zugriff während der Laufzeit notwendig war. In einer aktuellen Version wurde dieser Nachteil durch Wolfgang Schult beseitigt. Im weiteren Verlauf der Arbeit werden einige entwickelte Aspekte auch in der Version für den dynamischen Aspektweber vorgestellt.
Die Templatetoken `/*[CLASSPROTECTION]*/`, `/*[BASENAMESPACE]*/` und `/*[CLASSNAME]*/` werden durch den Aspektweber mit den Entsprechungen der Zielklasse ersetzt.

```
using _/*[CLASSNAME]*/BASE=/*[BASENAMESPACE]*/./*[CLASSNAME]*/;

[Serializable]
/*[CLASSPROTECTION]*/ class /*[CLASSNAME]*/:_/*[CLASSNAME]*/
   BASE, AdaptNet.CoFra.IConfigure
{
    // Implementierung der Schnittstelle IConfigure
}
```

Abbildung 3.15: Implementierung zusätzlicher Schnittstellen in Loom.Net

Im gezeigten Code-Fragment wird eine neue Klasse definiert, die den Namen der Basisklasse trägt, der in einem anderen Namensraum definiert ist. Die neue Klasse erbt von der Zielklasse, die zuvor in der initialen *using*-Direktive mit `/*[CLASSNAME]*/BASE` definiert wurde. Zusätzlich wird die Implementierung der Schnittstelle *AdaptNet.CoFra.IConfigure* deklariert und die erstellte Klasse mit dem CLI-Attribut `[Serializable]` als serialisierbar markiert, was die Persistierung von Objekten dieses Typs ermöglicht.

Der statische Aspektweber erstellt aus dem beschriebenen Aspektdokument und der funktionalen Klasse eine Quellcodedatei, welche die neue Klasse enthält. Zusätzlich muss bei der Generierung die Assembly der funktionalen Klasse sowie deren Name spezifiziert werden. Die entstandene Quellcodedatei muss nun in eine Assembly übersetzt werden und enthält dann zusammen mit der Assembly der funktionalen Klasse eine konfigurierbare Komponente.

Abbildung 3.16 zeigt einen Ausschnitt der Loom.Net Aspektimplementierung. Der Ausschnitt zeigt die Implementierung der *SetParameter*-Methode der hinzugefügten Schnittstelle.

```
public bool SetParameter(string name, string parameterValue)
  {
    Type _theType= typeof(/*[BASENAMESPACE]*/./*[CLASSNAME]*/);
    FieldInfo _field = _theType.GetField(name);
    _field.SetValue(this,Convert.ChangeType(parameterValue,
       _field.FieldType));
    return true;
  }
```

Abbildung 3.16: IConfigure.SetParameter

Der Quellcode zeigt, wie mit Hilfe der Reflexionsschnittstelle der CLI-Bibliothek der Wert eines Elements der funktionalen Klasse manipuliert wird. Dabei wird zunächst ein *FieldInfo*-Objekt, welches anhand des Arguments *name* identifiziert wird, über das Typobjekt der funktionalen Klasse akquiriert. Der nächste Schritt umfasst die Konvertierung des neuen Parameterwertes mit Hilfe der CLI-Konvertierungsfunktion *Convert.ChangeType*. Der Zieltyp kann über das ermittelte *FieldInfo*-Objekt anhand der *FieldType*-Property abgerufen werden. Abschließend wird der neue Wert an das Element der funktionalen Klasse zugewiesen (*SetValue*), welches den Parameter repräsentiert.

Parameter- und Konnektorquellattribute

Im Zusammenhang mit der Entwicklung der Aspekte wurde ein neues Programmiermodell für die Verwendung von Parametern und Konnektorquellen innerhalb der Kapsellogik entworfen. Wie schon erwähnt, werden Parameter und Konnektorquellen durch Attribute einer Klasse repräsentiert und können somit auch innerhalb der Kapsellogik verwendet werden. Im Adapt.Net-Projekt Annotationen definiert, mit denen diese Interaktionspunkte für die Aspektlogik markiert werden können. Die Metadatenannotierungen können später vom Werkzeug für die graphische Konfigurationserstellung verwendet werden.

```
public class Filter : MarshalByRefObject, IPictureSource{
    [AdaptNet.CoFra.Parameter]
    private double compressionRate;
    [AdaptNet.CoFra.Connection]
    private IPictureSource imageSource;
    public byte[] GetPicture(string name){
      // Benutzung einer anderen Kapsel
      Picture pict = imageSource.GetPicture(name);
      // Zugriff auf einen Parameter
      return this.compress(compressionRate,pict).ToByteArray();
    }
}
```

Abbildung 3.17: Programmiermodell für Parameter und Verbindungspunkte

Der Quellcodeausschnitt in Abbildung 3.17 zeigt ein Beispiel für die Deklaration und Annotierung von Parametern und Konnektorquellen sowie deren Verwendung in den Methoden. Das Attribut *compressionRate* repräsentiert einen Parameter der Kapsel *Filter*. Der Zugriff auf den Parameter erfolgt wie ein normaler Attributzugriff. Das Programmiermodell erleichtert die Programmierung verteilter Anwendungen, da die Konfiguration von Kommunikationsverbindungen durch das Framework realisiert wird und die Verbindungen zu anderen Kapseln über ein graphisches Werkzeug realisiert werden kann.

Reflexion vs. direkter Zugriff im Aspektcode

Im vorangegangenen Abschnitt wurde die Interaktion zwischen funktionalem Code und Aspektlogik über Methoden der Reflexionsschnittstelle realisiert. Im Vergleich zum direkten Zugriff auf die Elemente der funktionalen Klasse ist die Verwendung von Reflexion um den Faktor 100 langsamer [Forax u. a. 2005] (siehe auch Abschnitt 7.1 dieser Arbeit). Im Prinzip sind zur Erstellung der Konfigurationslogik alle Details bekannt, um den Elementzugriff direkt zu generieren. Abbildung 3.18 zeigt ein Beispiel für diese Variante.

```
public bool SetCompressionParameter(int parameterValue)
    this.compression = parameterValue;
```

Abbildung 3.18: Direkter Elementzugriff im Aspektcode

Mit dieser Variante kann sehr effektiv ein Kapselparameter manipuliert werden. Es ergibt sich hier nur ein Nachteil: Es kann nicht auf private Attribute der funktionalen Klasse zugegriffen werden, da Loom.Net eine Ableitung der Zielklasse erzeugt und dort den Aspektcode integriert. Die Verwendung von Reflexion erlaubt den Zugriff auf alle Attribute einer Klasse. Da sich die Zeit für das Setzen von Parametern im Bereich von Mikrosekunden bewegt und die Rekonfiguration im Verhältnis zu normalen Methodenaufrufen selten ausgelöst wird, wurde im Adapt.Net-Framework die erste Variante integriert. Neue Versionen von Aspektwebern, die mit der Manipulation der Zwischenrepräsentation (*Intermediate Language, Bytecode*) arbeiten, können die Vorteile beider vorgestellten Varianten vereinigen.

Konnektoren

Wie schon zuvor beschrieben, werden Konnektoren durch ein Attribut des Hauptobjekts einer Kapsel repräsentiert. Die Zuweisung einer Referenz auf eine andere Kapsel wurde äquivalent zu Kapselparametern implementiert. Um die Entwicklung heterogener Anwendungen zu unterstützen, wurden verschiedene Konnektortypen eingeführt. Je nach verwendeter Kommunikations-Middleware wird bei der Zuweisung an das Konnektorquellattribut ein Proxy-Objekt oder aber die direkte Objektreferenz auf ein anderes Hauptobjekt verwendet. In Adapt.Net sind eine Reihe von Konnektortypen integriert worden, die im Folgenden kurz vorgestellt werden sollen.
Befinden sich Quell- und Senkenkapsel eines Konnektors in einer *Application Domain*, kann die Kommunikation sehr laufzeiteffektiv durch direkte Methodenaufrufe (Konnektortyp **Direktaufruf**) am Hauptobjekt der Senkenkapsel vorgenommen werden. *Application Domains* sind leichtgewichtige Prozesse in der CLI, die Speicherschutz gegeneinander realisieren. Es können mehrere *Application Domains* innerhalb eines Betriebssystemprozesses existieren. Bei der Erzeugung eines Konnektors wird der *Connect*-Methode der Konfigurationsschnittstelle eine Referenz auf das Hauptobjekt der Senkenkapsel übergeben, welches mit Hilfe der Reflexionsmechanismen direkt induziert wird. Der große Vorteil der Möglichkeit von Direktaufrufen ist, dass austauschbare Einheiten in Form von Kapseln auch ohne den *Overhead* einer Kommunikations-Middleware innerhalb einer *Application Domain* unterstützt werden können.
Im Falle von verteilten Kapseln kommt eine Kommunikations-Middleware zum Einsatz, welche die Übertragung von Objektfernaufrufen über Rechnergrenzen hinweg realisiert. .NET Remoting ist eine Kommunikations-Middleware für die .NET Plattform, welche die Integration verschiedener Transportkanäle sowie unterschiedlichen *Formattern*, die das Format der übertragenen Daten bestimmen, unterstützt. In Adapt.Net wurden die gebräuchlichen Kombinationen **TCP/IP-Kanal mit binärer Formatierung** und **HTTP-Kanal mit SOAP Formatierung** als eigene Konnektortypen integriert. In .NET Remoting werden Aufrufe auf der Client-Seite an Proxy-Objekten durchgeführt, welche mit Hilfe von Metainformationen erstellt werden. Das Proxy-Objekt übernimmt die Verpackung der Argumente (*Marshalling*) und das Erstellen der Nachricht sowie die Übertragung über die Kommunikationsverbindung. Bei der Erzeugung von Konnektoren zwischen verteilten Kapseln wird zunächst eine .NET Remoting Verbindung zum Hauptobjekt der Senkenkapsel erstellt und eine Referenz auf das erzeugte Proxy-Objekt dem Konnektorquellattribut des Hauptobjekts der Quellkapsel zugewiesen.
Die Integration von CORBA-Objekten wurde über **IIOP-Konnektoren** realisiert. Es existieren verschiedene Frameworks, welche die Interoperation von CORBA-Objekten

und .NET-Remoting-Objekten ermöglichen. Im Rahmen eines „Großen Belegs“ [Puhlmann 2005] wurden von M. Puhlmann mit Borland's Janeva [Borland 2007] und IIOP.NET [IIOP.NET Homepage 2007], zwei Frameworks für die Integration von CORBA-Objekten, evaluiert. Beide Frameworks integrieren eine IIOP-Protokollinfrastruktur sowie einen ORB und einige CORBA-Dienste in die .NET Plattform. Die IIOP-Integration wurde durch die Implementierung eines speziellen *Formatters*, der die Abbildung der Typsysteme übernimmt, realisiert. Der *Connect*-Methode der Konfigurationsschnittstelle einer Kapsel wird in diesem Falle eine CORBA-Referenz (IOR) der Senkenkapsel und ein Typname übergeben, der bei der Typumwandlung der Referenz (*Narrow*) benötigt wird.
Abbildung 3.19 zeigt die generische Implementierung der *Connect*-Methode mit drei unterstützten Konnektortypen. Es wird zunächst über den *connectionType*-Parameter die Implementierung eines Konnektortyps selektiert. Beim Konnektortyp „Direktaufruf“ wird der *options*-Parameter, der die Referenz auf das Hauptobjekt der Senkenkapsel enthält, direkt an des Konnektorquellattribut übergeben. Bei den Konnektortypen `REMOTING` und `IIOP` wird eine Verbindung zum Hauptobjekt der Senkenkapsel erstellt. Eine Referenz auf das entstandene Proxy-Objekt wird dem Konnektorquellattribut zugewiesen.

```
public bool Connect(string portName, string connectionType,
   object options){
  if(connectionType == "DIRECT"){
    FieldInfo conn = this.GetFieldofThis(portName);
    conn.SetValue(this,options);
    return true;
  }
  if(connectionType == "REMOTING"){
    object[] params = (object[]) options;
    FieldInfo conn = this.GetFieldofThis(portname);
    string connStr = "tcp://"+params[1]+"/"+params[0];
    conn.SetValue(this,Activator.GetObject(conn.FieldType,
       connStr));
    return true;
  }
  if(connectionType == "IIOP"){
    FieldInfo conn = this.GetFieldofThis(portname);
    object corbaRef = Narrow(options);
    conn.SetValue(this,corbaRef);
    return true;
  }
}
```

Abbildung 3.19: Auszug der Implementierung von IConfigure.Connect

Die Adapt.Net-Konnektorarchitektur ermöglicht die Erstellung heterogener Anwendungen, die vor allem im Bereich mobiler Systeme benötigt wird, wo eine Vielzahl unterschiedlicher Middleware-Plattformen existiert. Zusätzlich ermöglicht die Konnektorarchitektur die Integration externer Dienste, wie z.B. der verbreiteten *Web-Services*. Dieses Thema wird im Bereich der dynamischen Dienstintegration in Abschnitt 3.8 untersucht.

Start kapseleigener Threads

Die Implementierung der *Start*-Methode muss für die Erzeugung der kapseleigenen Threads sorgen. In Adapt.Net existieren zwei generische Implementierungen für die *Start*-Methode. Zum einen kann die Hauptklasse einer Kapsel die Methode *void Main (string[] args)* implementieren. In diesem Fall wird in der Implementierung der *Start*-Methode der Konfigurationsschnittstelle ein Thread mit dieser *Main*-Methode als Eintrittspunkt erzeugt. Die Thread-Erzeugung, die während der Initialisierung außerhalb der zeitkritischen Blockierungsphase erfolgt, wurde getrennt vom eigentlichen Thread-Start während der Blockierungsphase realisiert. Eine Besonderheit in der .NET Plattform stellen Kapseln für die Nutzerinteraktion in Form von *WinForms* und *UserControls* dar. Diese müssen durch die Übergabe einer Referenz des Hauptobjekts an die Methode *Application.Run()* aktiviert werden. Diese startet eine Schleife, welche die Nachrichten des Windows-Subsystems verarbeitet. Kapseln für die graphische Nutzerinteraktion müssen durch eine separate Implementierung der *Start*-Methode behandelt werden. Im Gegensatz zur ersten Variante wird bei diesem Kapseltyp ein neuer Thread gestartet, der die statische Methode *Application.Run(this)* aufruft und in dieser Methode bis zum Ende der Anwendung verweilt. Das Windows-Subsystem sorgt dafür, dass die entsprechende Ereignisbehandlungsroutine der Kapseln aufgerufen wird.
Die Auswahl der Startart innerhalb der konfigurationsspezifischen Logik muss bei der Erstellung der konfigurierbaren Anwendung vom Entwickler ausgewählt werden bzw. wird vom später beschriebenen graphischen Konfigurationswerkzeug automatisch realisiert.

3.5.5 Der Kapselkonfigurator

Um Integration heterogener Komponentenplattformen zu ermöglichen, wird die Erzeugung von Kapseln von einer Fabrik übernommen. Diese abstrahiert von plattformspezifischen Details und implementiert Methoden zum Erzeugen und Entladen von Kapseln sowie die Abfrage des Zustands ihres Lebenszyklusstatus. Die Verwaltung der Kapselkonfiguratoren im Konfigurationsmanager wurde als Strategie-Entwurfsmuster [Gamma u. a. 1995] implementiert. Zusätzlich nutzt der Konfigurationsmanager die Kapselkonfiguratoren zum Zugriff auf die Konfigurationsschnittstellen der einzelnen Kapseln. Die Kapselkonfiguratoren implementieren die Schnittstelle *IConfigure*. Als Proxy-Objekte leiten sie Aufrufe vom Konfigurationsmanager an die Hauptobjekte der Kapseln weiter. Dabei müssen ggf. Anpassungen an Plattformspezifika der jeweiligen Komponentenplattform vorgenommen werden.

```
interface ICapsuleConfigurator:IConfigure,IStateTransfer,IUpdate
{
    bool Create(string name, string assembly, string mainType,
          string args, string access,string assemblyVersion);
    bool UnLoad();
    event OnStateChange stateChange;
    State GetState;
    object GetObjectProxy;
    int GetAccessPort;
}
```

Abbildung 3.20: Schnittstelle des Kapselkonfigurators

Abbildung 3.20 zeigt einen Ausschnitt der Schnittstelle *ICapsuleConfigurator*, die von jedem Kapselkonfigurator implementiert wird. Zunächst ist in der ersten Zeile zu erkennen, dass jeder Kapselkonfigurator zusätzlich die Schnittstellen *IConfigure*, *IStateTransfer* und *IUpdate* implementieren muss, um den Zugriff auf die verwalteten Kapseln zu ermöglichen. Neben der Konfigurationsschnittstelle können auch die letztgenannten Schnittstellen vom Hauptobjekt einer Kapsel implementiert werden. Diese werden für dynamische Aktualisierungen und Zustandstransfers benötigt.
Eine Kapsel wird durch den Aufruf der *Create*-Methode erzeugt. Dabei werden der Name und die Version der Assembly, der Name der Hauptklasse und evtl. weitere Argumente (je nach Erzeugungsart einer Kapsel z.B. Konstruktorargumente oder Kommandozeilenparameter) übergeben. Die Methode *UnLoad* wird zum Entladen einer Kapsel verwendet.

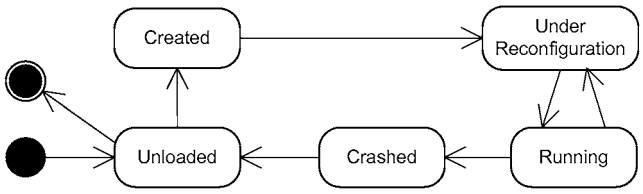


Abbildung 3.21: Lebenszyklus einer Kapsel

Der Lebenszyklus einer Kapsel kann mit Hilfe der Methode *GetState* über den Kapselkonfigurator abgerufen werden. Abbildung 3.21 zeigt die möglichen Zustände einer Kapsel und entsprechende Übergänge. Eine Kapsel befindet sich im Zustand *Crashed* (Ausfall), wenn sie von ihrer spezifizierten Funktionalität abweicht. Ein Ausfall liegt unter anderem vor, wenn der umgebende Betriebssystemprozess terminiert wurde. Wird eine Kapsel durch einen entsprechenden Kapselkonfigurator in einem separaten Betriebssystemprozess gestartet, kann die Möglichkeit der Zustandsüberwachung unterstützt werden. Der Ausfall einzelner Kapseln soll in einem späteren Kapitel betrachtet werden. Zunächst wird die Fehlerfreiheit der Kapseln vorausgesetzt. Das Ereignis *stateChange* kann benutzt werden, um eine Ereignisbehandlungsroutine zu registrieren, die im Falle einer Zustandsänderung aufgerufen wird.
Für die Erzeugung von Konnektoren zwischen den Kapseln wird unter Umständen eine Referenz auf das vom Kapselkonfigurator abstrahierte Hauptobjekt der Kapsel benötigt. Die Methode *GetObjectProxy* ermöglicht den Zugriff auf diese Referenz.
Die Methode *GetAccessPort* wird für die Konfiguration des TCP/IP-Ports einer Kapsel benötigt. Auf einem Rechner können mehrere Exemplare des Konfigurationsmanagers ausgeführt werden, die anhand ihres TCP/IP-Ports unterschieden werden. Zusätzlich können einzelne Kapseln in separaten Betriebssystemprozessen erzeugt werden. Die Verwaltung einer Kapsel erfolgt dann über eine TCP/IP-Verbindung, deren Port wiederum über den Kapselkonfigurator abgefragt werden kann.

Implementierte Kapselkonfiguratoren

Im Adapt.Net-Framework wurden eine Reihe wiederverwendbare Kapselkonfiguratoren implementiert. Diese sollen in diesem Abschnitt vorgestellt werden.
Die Erzeugung von **In-Process**-Kapseln erfolgt mit Hilfe des *new*-Operators der Laufzeitumgebung. Das heißt, das Hauptobjekt wird im Adressraum (bzw. der *Application*

Domain) des Konfigurationsmanagers geladen, der die *Create*-Methode des Kapselkonfigurators aufruft.
Bei Kapselkonfiguratoren des Typs **Remoting** wird zusätzlich zur Erzeugung des Hauptobjekts durch *new*, das Hauptobjekt als .NET-Remoting-Dienst registriert. Dieser wird dann im Kontext des Konfigurationsmanagers ausgeführt. Die benötigten Kommunikationskanäle werden vom Konfigurationsmanager zuvor initialisiert.
Kapselkonfiguratoren des Typs **Separate-Process** starten beim Erzeugen einer Kapsel einen eigenen Betriebssystemprozess, in dessen Kontext die Kapsel ausgeführt wird. Die Eintrittsfunktion des Prozesses initialisiert die .NET-Remoting-Umgebung mit entsprechenden Kommunikationskanälen und startet, falls notwendig, kapseleigene Threads. Um die Kommunikation zwischen dem Konfigurationsmanager und auch zwischen den Kapseln selbst zu ermöglichen, wird das erzeugte Hauptobjekt als .NET-Remoting-Dienst registriert. Der TCP/IP-Port für die .NET-Remoting-Verbindung wird beim Start des Betriebssystemprozesses als Kommandozeilenparameter übergeben. Der Konfigurationsmanager jedes Rechners speichert über ein statisches Attribut die vergebenen TCP/IP-Ports, da auf einem Rechner ein TCP/IP-Port nur einmal verwendet werden darf. Der TCP/IP-Port muss dem Konfigurationsmanager bekannt sein, um eine Verbindung zum Hauptobjekt der Kapsel erstellen zu können. Den verwendeten TCP/IP-Port kann der Konfigurationsmanager über die Methode *GetAccessPort* des Kapselkonfigurators abfragen.
Im Rahmen der CORBA-Integration durch M. Puhlmann wurde der Kapselkonfigurator für **Java-basierende-CORBA**-Kapseln implementiert. Das Konzept der Kapselkonfiguratoren erwies sich dabei als großer Vorteil. Der implementierte Kapselkonfigurator benutzt einen *Class-Loader* zum Laden der Java *.class*-Datei. Mit Hilfe der Reflexionsschnittstelle (*class.newInstance()*) der Java-Laufzeitumgebung kann das Hauptobjekt der Kapsel erzeugt werden. Ein Teil des Kapselkonfigurators wurde in Java implementiert. Anhand einer IDL-Beschreibung, die der Schnittstelle *ICapsuleConfigurator* entspricht, wurde die Java-Seite dieses Kapselkonfigurators implementiert. Mit Hilfe des IIOP.NET Interoperations-Frameworks und unter Nutzung der IDL-Beschreibung wurde die Kommunikation zwischen CLI und Java-Teil realisiert.

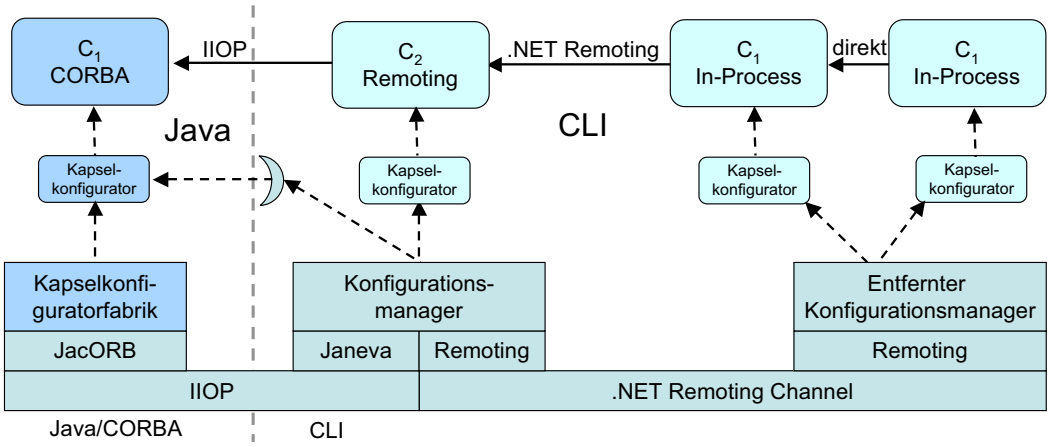


Abbildung 3.22: Integration heterogener Kapseln

Abbildung 3.22 zeigt die Architektur der Integration heterogener Kapseln. Neben der Nutzung eines Interoperationsframeworks wurden zusätzliche Komponenten in Java implementiert. Hierzu gehören eine Transaktionsverwaltung, Monitoring-Komponenten,

ein Teil des Kapselkonfigurators mit der Fabrikfunktionalität sowie konfigurations- und adaptationsspezifische Logik. Details zur Java-Integration finden sich in [Rasche u. a. 2005a].
Nachdem nun die Adapt.Net-Laufzeitumgebung und die Verwaltung von Kapseln beschrieben wurden, soll im folgenden Abschnitt die Implementierung der Rekonfigurationsalgorithmen vorstellt werden. Im Vordergrund steht dabei die Synchronisationslogik für das Blockieren von Konnektoren und damit das Erreichen eines rekonfigurierbaren Zustands.

3.6 Erreichen eines rekonfigurierbaren Zustands

Im vorangegangenen Kapitel wurde die Menge der Rekonfigurationselemente einer dynamischen Rekonfiguration eingeführt. Rekonfigurationselemente können Kapseln und Konnektoren sein. Die vorgestellten Rekonfigurationsalgorithmen basieren auf dem Blockieren von Konnektoren. Ausgehend von der Menge der Rekonfigurationselemente muss zunächst die Menge der zu blockierenden Konnektoren berechnet werden. Beim Blockieren bestehen zwei Entwurfsalternativen für die Integration von Sychronisationslogik, die im Folgenden diskutiert werden sollen.

3.6.1 Synchronisationspunkte und Programmierschnittstellen

Abbildung 3.23 verdeutlicht mögliche Blockierungsstellen einer Adapt.Net-Anwendung. Die dargestellte Konfiguration enthält drei Kapseln und zwei Konnektoren. Wie schon im vorangegangenen Kapitel beschrieben, werden auch direkte Methodenaufrufe von Threads als Konnektoren betrachtet, was in der Abbildung durch den Konnektor C_{t1} verdeutlicht ist.

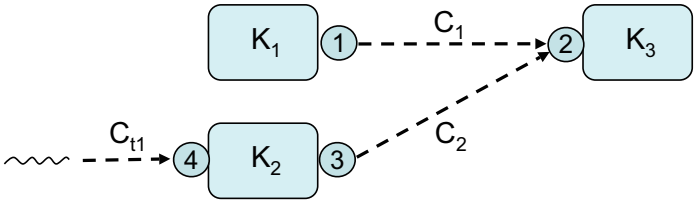


Abbildung 3.23: Synchronisationspunkte beim Blockieren

Konnektoren können auf der Seite des Rufers oder auf der Seite des Gerufenen blockiert werden. Im Rahmen dieser Arbeit wurden für beide Varianten entsprechende Mechanismen entwickelt. Zum einen kann auf der Seite des Rufers (**client-seitig**) durch die Verwendung einer Bibliothek Anfang und Ende von Kommunikationstransaktionen markiert werden. Diese Bibliothek enthält die entsprechende Synchronisationslogik für das Blockieren von Konnektoren. Der Entwickler muss bei der Implementierung der Kapsellogik die Kommunikationstransaktionen markieren (Abbildung 3.24). In Abbildung 3.23 betrifft diese Variante die Punkte 1 und 3.
Abbildung 3.24 zeigt ein Beispiel für die Benutzung der Adapt.Net-Bibliotheksfunktionen zum Markieren von Kommunikationstransaktionen. Im Falle des Blockierens des Konnektors „*image*“ wird innerhalb der statischen Methode *Transaction.Begin* das Warten des Kontrollflusses auf die Beendigung der dynamischen Rekonfiguration realisiert.

```
Transaction.Begin("image");
  for(int i=0;i<512;i++)
    image.SendLine(picture[i]);
Transaction.End("image");
```

Abbildung 3.24: Markieren von Kommunikationstransaktionen (client-seitig)

Zum anderen wird für die Blockierung auf der Seite des Gerufenen (**server-seitig**) ein dynamischer Proxy-Ansatz verwendet. Bei dieser Art der Integration der Blockierungsfunktionalität werden vor bzw. nach jeder Methodenausführung entsprechende Synchronisationsprimitive integriert (Punkte 2 und 4 in Abbildung 3.23). Die Synchronisationslogik für das Server-seitige Blockieren von Konnektoren kann als Aspekt von der funktionalen Anwendungslogik separiert werden. Dieser Ansatz wird als dynamischer Proxy bezeichnet.

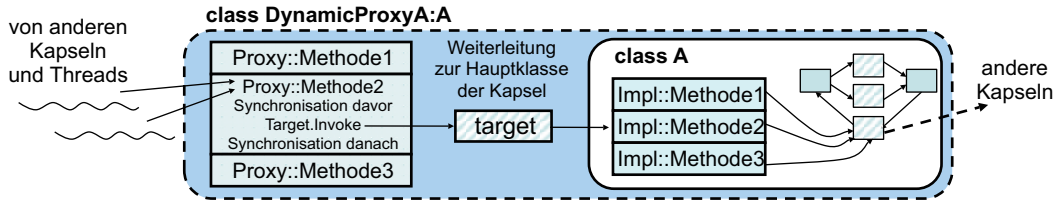


Abbildung 3.25: Dynamischer Proxy-Ansatz

Abbildung 3.25 zeigt den proxybasierten Ansatz für die Integration von Synchronisationslogik auf Seite der gerufenen Kapsel. Als Referenz auf eine Kapsel existieren bei diesem Ansatz nur Referenzen auf einen Proxy. Der Proxy leitet Methodenaufrufe an die originale Kapsel weiter, die durch eine Variable (*target*) im Proxy referenziert ist. Beim Erzeugen einer Kapsel werden vom Konfigurationsmanager nur Referenzen auf Exemplare des Proxys verteilt und keine direkte Referenz auf das Hauptobjekt der Kapsel.
Da alle Client-Kapseln bzw. Threads einer Kapsel nur Referenzen auf den Proxy besitzen, ist es mit diesem Ansatz sehr komfortabel möglich, alle Aufrufe an einer Kapselmethode zu verfolgen bzw. Synchronisationslogik vor und nach der Abarbeitung der eigentlichen Methoden zu integrieren. Bei dieser Variante des Blockierens von Verbindungen werden automatisch sämtliche eingehenden Konnektoren blockiert, da innerhalb des Proxys die Rufer nicht mehr differenziert werden können.
Abbildung 3.26 zeigt einen Auszug des Rekonfigurationsaspekts für den dynamischen Weber Rapier-Loom.Net. Die Methode *InvokeAll* wird nach der Verwebung an Stelle der Zielmethode ausgeführt und leitet die Methodenaufrufe über die *Context*-Methode *Invoke* weiter. Wurde die Kapsel seit ihrer Erzeugung noch nicht rekonfiguriert, werden die Aufrufe direkt an die Originalmethoden des verwobenen Exemplars weitergeleitet (*Context.Invoke*). In diesem Fall existiert nur ein Exemplar des verwobenen Typs, der auch die funktionale Schnittstelle der Kapsel implementiert. Nach der ersten Aktualisierung werden die Methodenaufrufe per *Context.InvokeOn* direkt an die Methoden einer potentiell aktualisierten Version der Kapsel weitergeleitet, die durch die *target*-Variable referenziert wird. Diese Technik wird für die dynamische Aktualisierung von Kapseln verwendet.

```
[Introduces(typeof(IConfigure))]
public class SynchronizationAspect:Aspect{
  private object target;
  [Call(Invoke.Instead)]
  [IncludeAll]
  public object InvokeAll(object[] args){
    try{
      // Synchronisationslogik vor dem Methodenaufruf
      if (target == null)
        return Context.Invoke(args);
            else
              return Context.InvokeOn(target, args);
    }
    finally{
      // Synchronisationslogik nach dem Methodenaufruf
    }
  }
}
```

Abbildung 3.26: Synchronisationsaspekt: Server-seitiges Blockieren

An dieser Stelle soll noch einmal die Überführung in einen rekonfigurierbaren Zustand an Hand der Beispielkonfiguration in Abbildung 3.23 diskutiert werden. Für eine dynamische Aktualisierung von Kapsel K_3 kann entweder in Punkt 2 blockiert werden, also im dynamischen Proxy dieser Kapsel, oder in den Punkten 1 und 3 mit dem Markieren von Kommunikationstransaktionen auf Ruferseite der Konnektoren C_1 und C_2. Soll aber wiederum eine neue Kapsel zwischen K_1 und K_3 eingefügt werden, muss das Blockieren in den Punkten 1 und 3 erfolgen, da sich nach der dynamischen Rekonfiguration die Verbindungsstruktur ändert und sich in Punkt 2 hängende Aufrufe (bei Nutzung des Proxy-Ansatzes) nicht an die geänderte Struktur anpassen und den Ruf direkt in K_3 fortsetzen und nicht in der hinzugefügten Kapsel.
Auch das Blockieren von Threads erfordert eine gesonderte Betrachtung, wenn der Quellcode der Logik der rufenden Kapsel nicht vorliegt. Dies kommt z.B. bei *WinForms*-basierenden Kapseln vor. Hier können Rufe von einem Thread erfolgen, bei dem das Markieren und damit das Einfügen von Synchronisationslogik auf Grund des fehlenden Quellcodes nicht möglich ist. Das Blockieren kann hier nur mit dem dynamischen Proxy-Ansatz erfolgen. Der Konfigurationsmanager analysiert vor dem Ausführen der Rekonfiguration die Konfigurationsänderungen und kann die möglichen Blockierungspunkte ermitteln. Bei der Implementierung der Transaktionsbibliothek wurden dafür entsprechende Mechanismen vorgesehen. Falls Transaktionsmarkierungen in die Thread-Logik eingefügt wurden, wird dies in der Kapsel gespeichert, damit der Konfigurationsmanager die zu blockierenden Rekonfigurationselemente bestimmen kann. Zunächst soll nur das Blockieren von Konnektoren auf der Seite des Rufers für die Vorstellung der implementierten Sychronisationsverfahren verwendet werden.

3.6.2 Synchronisation bei azyklischen Verbindungsgraphen

Aus den threoretischen Betrachtungen folgt die Menge der an der Rekonfiguration beteiligten Elemente. Das Erreichen eines rekonfigurierbaren Zustands wurde bei Konfi-

gurationen mit azyklischen Verbindungsgraphen durch das Blockieren von Konnektoren realisiert, wobei eine bestimmte Reihenfolge bei der Blockierung beachtet wird, um *Verklemmungen* in der Anwendung während der Blockierungsphase zu vermeiden. Aus der Menge der Rekonfigurationselemente kann die Menge der zu blockierenden Konnektoren berechnet werden. Dies sind alle in der Menge der Rekonfigurationselemente enthaltenen Konnektoren sowie alle eingehenden und ausgehenden Konnektoren von Kapseln, die in der Menge der Rekonfigurationselemente enthalten sind. Die Bestimmung der Reihenfolge wird an Hand des im Abschnitt 3.3.1 beschriebenen Algorithmus implementiert. Auf Grund des begrenzten Platzes wird auf die Beschreibung der Implementierung verzichtet.

3.6.3 Effizientes Blockieren von Konnektoren

Ausgehend von der berechneten Reihenfolge, muss beim Blockieren eines Konnektors sichergestellt werden, dass der Aufruf der Blockierungsmethode der Konfigurationsschnittstelle erst zurückkehrt, wenn alle im Moment des Aufrufs aktiven Kommunikationstransaktionen über diesen Konnektor beendet sind und zusätzlich keine neuen Kommunikationstransaktionen über ihn initiiert werden können. Zunächst soll die Lösung dieses Synchronisationsproblems für azyklische Verbindungsstrukturen betrachtet werden. Dabei sollen verschiedene Lösungsstrategien dargestellt und deren Leistungsmerkmale diskutiert werden.

Leistungsmerkmale

Bei der dynamischen Rekonfiguration sind im Zusammenhang mit dem Blockieren von Konnektoren zwei Leistungsmerkmale entscheidend. Zum einen sind die **Zusatzkosten** (*Overhead*) während der Anwendungslaufzeit zu beachten - also die Ressourcen, die für die Ausführung zusätzlicher synchronisationsspezifischer Logik, notwendig sind. Die Zusatzkosten können unter anderem durch vergleichende Messung von Methodenaufrufen mit und ohne Synchronisationslogik bewertet werden.
Zum anderen ist die **Dauer des Blockierens** von Konnektoren ein wesentliches Leistungsmerkmal. Dies ist der Zeitraum von der Initiierung des Blockierens bis zum Erreichen des rekonfigurierbaren Zustands. Dieser Zeitraum ist ein Bestandteil der Unterbrechnungsdauer, während der involvierte Kapseln potentiell keine Methodenaufrufe mehr bearbeiten. Die Minimierung dieses Zeitraums ist wichtig bei der Entwicklung adaptiver Anwendungen, um ihre Nutzbarkeit nicht zu beeinträchtigen.

Synchronisation

Die Sychronisation von Anwendungs- und Rekonfigurationslogik wurde zunächst mit Hilfe von zwei Mutex-Variablen implementiert. Abbildung 3.27 zeigt einen Quellcodeausschnitt der Adapt.Net-Transaktionsbibliothek und die Methoden der Konfigurationsschnittstelle, welche in die Synchronisation involviert sind.
Beim Beginn einer neuen Kommunikationstransaktion wird zunächst geprüft, ob eine Blockierungsanfrage vorliegt und ggf. auf die Signalisierung des *block_mutex* gewartet. Andernfalls wird die Variable *processing* auf den Wert *true* gesetzt und das Mutex (*proc_mutex*) akquiriert. Bei der Beendigung der Kommunikationstransaktion wird das Synchronisationsobjekt *proc_mutex* wieder freigegeben (*Release*).

```
TransactionBegin:
  if(blocked) Wait(block_mutex);
  processing=true;
  Wait(proc_mutex);
TransactionEnd:
  processing=false;
  Release(proc_mutex);
```

```
BlockConnection:
  blocked=true;
  Wait(block_mutex);
  if(processing) Wait(proc_mutex);
  Release(proc_mutex);
Start:
  blocked=false;
  Release(block_mutex);
```

Abbildung 3.27: Sychronisation mit 2 Mutex-Variablen

Beim Aufruf der *BlockConnection*-Methode wird zunächst die Variable *blocked* auf den Wert *false* und das Mutex *block_mutex* durch den Aufruf *Wait* akquiriert. Nun wird auf die Beendigung aller laufenden Kommunikationstransaktionen gewartet, indem im Falle einer existierenden Kommunikationstransaktion (*processing=true*) auf das Mutex *proc_mutex* gewartet wird. Schlussendlich wird beim Aufruf von *Start* die Blockierung aufgehoben.
Der beschriebene Synchronisationsmechanismus kann für Anwendungen mit einem Thread pro Kapsel eingesetzt werden, wobei pro Kapsel und Konnektor je ein Exemplar der beschriebenen Synchronisationsobjekte benötigt wird. Der beschriebene Algorithmus erlaubt keine verschachtelten Kommunikationstransaktionen (*nested transactions*).

Multithreading und Rekursion

In diesem Abschnitt soll beschrieben werden, wie auch die dynamische Rekonfiguration von Kapseln mit multiplen Threads sowie verschachtelten Kommunikationstransaktionen unterstützt werden kann. Zunächst soll aber ein kurzes Beispiel zeigen, warum der beschriebene Algorithmus im Falle verschachtelter Kommunikationstransaktionen fehlschlägt.
Abbildung 3.28 zeigt ein Beispiel verschachtelter Kommunikationstransaktionen, die durch rekursive Methodenaufrufe innerhalb einer Kapsel entstehen können. Tritt eine Rekonfigurationsaufforderung ein, wenn sich der Kontrollfluss gerade beim Aufruf *comp.SomeMethod()* in der Methode *Call2* befindet, wird zunächst die Methode *Call1* aufgerufen. In *Call1* wird eine neue Kommunikationstransaktion über den Konnektor *comp* initiiert. Das Problem ist nun, dass am Ende von *Call1* die Kommunikationstransaktion über *comp* abgeschlossen wird. Bei Verwendung des beschriebenen Synchronisationsverfahrens würde die Rekonfiguration zu diesem Zeitpunkt ausgeführt, obwohl noch ein offener Transaktionskontext innerhalb von *Call2* existiert.
Zur Synchronisation von Anwendungs- und Rekonfigurationslogik wurden deshalb *ReaderWriterLocks* [Ben-Ari 1982, S. 80 ff.] benutzt. *ReaderWriterLocks* werden allgemeinhin verwendet, um konkurrierende Lese- und Schreibzugriffe auf gemeinsame Datenstrukturen zu synchronisieren. Bei einer Anforderung für einen Schreibzugriff wird auf die Beendigung aller bis zu diesem Zeitpunkt gewährten Sperren für Lesezugriffe gewartet und neue Anforderungen für Lesezugriffe werden in einer Warteschlange gespeichert. Je nach Implementierung können nach Beendigung eines Schreibzugriffs auch zunächst weitere Schreibzugriffe vor wartenden Anforderungen für Lesezugriffe gewährt werden. Diese Variante soll auf Grund der Möglichkeit des Verhungerns (*starvation*) von Leseanforderungen bei einer großen Anzahl an Schreibzugriffen nicht betrachtet werden. Anstatt dessen werden Implementierungen verwendet, bei denen

```
public class NestedTransaction
{
    private IOtherComponent comp;
    public void Call1(){
        Transaction.Begin("comp");
        comp.SomeOtherMethod();
        Transaction.End("comp");
    }
    public void Call2(){
        Transaction.Begin("comp");
        comp.SomeMethod();
        Call1();
        comp.SomeMethod();
        Transaction.End("comp");
    }
}
```

Abbildung 3.28: Synchronistion verschachtelter Kommunikationstransaktionen

Anforderungen nach der Reihenfolge der Warteschlange bearbeitet werden. Eine Erweiterung der *ReaderWriterLocks* ist die Unterstützung rekursiver Anforderungen für Lesezugriffe. Hierbei können trotz anstehender Schreibanforderungen für einen Thread neue Lesezugriffe gewährt werden, wenn dieser schon gewährte Lesezugriffe besitzt.
ReaderWriterLock können mit Hilfe von existierenden Synchronisationsmechanismen moderner Betriebssysteme wie Mutex- oder Semaphor-Variablen sowie mit Threadlokalem Speicher (TLS) implementiert werden. TLS ist ein Betriebssystemmechanismus, bei dem Variablen spezifisch für einen Thread-Kontext sind, das heißt der Zugriff aus dem gleichen Thread immer denselben Wert der Variablen liefert. Ein wichtiger Vorteil von *ReaderWriterLocks* ist, dass keine Betriebssystemsynchronisationmechanismen benötigt werden, um eine Anforderung für einen Lesezugriff zu verarbeiten, wenn keine Anforderung für einen Schreibzugriff vorliegt. In diesem Fall wird nur eine atomare *Compare/Exchange*-Operation, die von modernen Prozessoren in den Befehlssatz integriert wurden, sowie ein Zugriff auf den TLS benötigt.
In dieser Arbeit werden *ReaderWriterLocks* für die Synchronisierung von Anwendungslogik und Rekonfigurationsoperationen verwendet. Kommunikationstransaktionen werden als Anforderung für Lesezugriffe verstanden, wobei der Beginn einer Kommunikationstransaktion die Anforderung des Ressourcenzugriffs darstellt und die Beendigung die Ressourcenfreigabe. Auf der anderen Seite wird beim Aufruf der *BlockConnection*-Methode eine Anforderung für einen Schreibzugriff auf den gewünschten Konnektor ausgelöst. Der große Vorteil ist auch hier wiederum, dass bei Abwesenheit einer Blockierungsaktion nur geringe Zusatzkosten in Form einer atomaren Operation und einem TLS-Zugriff verursacht werden.
Durch die Verwendung eines *ReaderWriterLocks* wird in Adapt.Net auch die dynamische Rekonfiguration von Anwendungen mit multiplen Threads und rekursiven Methodenaufrufen unterstützt. Die verwendete *ReaderWriterLock*-Implementierung unterstützt die Aktivierung von geschachtelten Anforderungen für Lesezugriffe. Dabei können Threads mit gewährten Lesezugriffen neue Lesezugriffe gewährt werden, trotz existierender Anforderungen für Schreibzugriffe. Eine anstehende Anforderung für einen Schreibzugriff wird erst gewährt, wenn die laufenden Kommunikationstransak-

tionen aller Threads beendet wurden. Auf die dynamische Rekonfiguration übertragen, können also Threads mit aktiven Kommunikationstransaktionen weitere Kommunikationstransaktionen starten, die zur Beendigung von aktiven Kommunikationstransaktionen benötigt werden. Die Blockierung eines Konnektors wird verzögert, bis alle zu diesem Zeitpunkt offenen Kommunikationstransaktionen beendet wurden.
Der erweiterte Synchronisationsalgorithmus unterstützt nun auch rekursive Funktionsaufrufe, die eine Teilmenge der verschachtelten Kommunikationstransaktionen darstellen.

3.6.4 Synchronisation bei zyklischen Verbindungsgraphen

In diesem Abschnitt wird ein neues Synchronisationsverfahren für zyklische Verbindungsstrukturen beschrieben werden, bei denen die Bestimmung einer Blockierungsreihenfolge nicht möglich ist. Eigentlich handelt es sich bei verschachtelten Kommunikationstransaktionen schon um einen Sonderfall zyklischer Verbindungsgraphen, der aber erfolgreich durch den Einsatz von *ReaderWriterLocks* behandelt werden kann. Aber auch Zyklen über mehr als eine Kaspel können erfolgreich mit Hilfe von *ReaderWriterLocks* behandelt werden, wenn die Anwendung auf einem Rechner ausgeführt wird. Im Falle verteilter Anwendungen kann das Verfahren nicht mehr verwendet werden, da *ReaderWriterLocks* nicht über Rechnergrenzen hinweg funktionieren, weil für die Implementierung TLS benötigt wird, der in den eingesetzten Betriebssystemen nicht für verteilte Anwendungen verfügbar ist. Zusätzlich verursacht der Einsatz von *ReaderWriterLocks* zwar relativ geringe Zusatzkosten, aber dennoch konnten diese durch einen in dieser Arbeit neu entwickelten Algorithmus (ReDAC) optimiert werden.
ReDAC basiert auf der Speicherung aktiver Threads und deren Rekursionstiefe innerhalb der Kapseln. Bei jedem Eintritt eines Threads in eine Kapsel wird ein Zähler für diesen Thread inkrementiert, beim Verlassen dekrementiert. Beim Blockieren eines Konnektors kann für jeden Thread selektiert werden, ob er blockiert werden muss oder nicht, weil er noch auf einer Kapsel des *BlockSet* aktiv ist. Der Vorteil dieses Algorithmus ist, dass er mit Hilfe von logischen Thread-IDs auch über Rechnergrenzen hinweg funktioniert und zusätzlich mit sehr geringen Zusatzkosten in die Kapseln integriert werden kann.

```
1  LogicalThread curThread = LogicalThread.CurrentThread;
2  Counter countRef = null;
3
4  regetentry:
5  if (!_threadList.TryGetValue(curThread, out oldCount)){
6    lock (_threadList){
7        if (!_threadList.TryGetValue(curThread, out countRef)){
8              countRef = new Counter(curThread);
9              _threadList.Add(curThread, countRef);
10       }
11   }
12 }
13 if (!oldCount.thread.Equals(curThread))
14     goto regetentry;
```

Abbildung 3.29: Hinzufügen neuer Threads

Abbildung 3.29 zeigt einen ersten Quellcodeausschnitt von ReDAC. Für jede Kapsel existiert eine Datenstruktur *_threadList* in Form einer Hash-Tabelle (*Dictionary<Thread,Counter>*)[3]. In der Hash-Tabelle werden die Zähler in Form eines *Counter*-Exemplars für jeden Thread gespeichert. Als Schlüssel für den Zugriff auf den Zähler wird eine logische Thread-ID verwendet (Zeile 1). Für den Zähler wurde eine zusätzliche Indirektion gewählt. In einem *Counter*-Exemplar existiert ein Element vom Typ *int*, mit dem eigentlichen Zählerwert. Diese Indirektion wurde verwendet, um teure Synchronisationsoperationen beim Zugriff auf die Hash-Tabelle zu vermeiden. Diese wären bei direkter Benutzung des Zählers nötig, da der Zähler zunächst aus der Datenstruktur gelesen, inkrementiert bzw. dekrementiert werden muss und daraufhin zurück in die Datenstruktur geschrieben wird. Dies müsste synchronisiert werden, da potentiell konkurrierende Threads auf die Datenstruktur zugreifen. Durch die Verwendung der Indirektion wird der Zugriff auf den Zähler zu einer reinen Leseoperation.
Tritt ein Thread das erste Mal in eine Kapsel ein, muss ein neuer Eintrag in der Hash-Tabelle erzeugt werden (Zeile 5-11). Für die Synchronisation von konkurrierenden Eintritten in diesen Abschnitt wird eine Sperre (*lock*) für das *_threadList*-Objekt akquiriert, da es bei gleichzeitigem Einfügen und Lesen in der Hash-Tabelle, wegen einer potentiellen Vergrößerung des Hash-Tabellen-Feldes, zum Auslesen fehlerhafter Einträge kommen kann. Von den Zusatzkosten her ist dies unproblematisch, da dieser Code nur im Zusammenhang mit der Erzeugung neuer Threads ausgeführt wird, einer im Verhältnis zu einem Eintrag in eine Hash-Tabelle ohnehin teuren Aktion.
Auf Grund der nicht Thread-sicheren Implementierung der Hash-Tabellen des .NET Frameworks, die an Hand des verfügbaren Quellcodes untersucht wurde, kann es vorkommen, dass beim Zugriff auf die Tabelle ein falscher Eintrag ermittelt wird. Deshalb wird in jedem *Counter*-Exemplar zusätzlich die logische Thread-ID des zugehörigen Threads gespeichert. Eine Überprüfung wird am Ende des gezeigten Quellcodeausschnitts (Zeile 13) vorgenommen, indem die ermittelte Thread-ID mit dem Eintrag im *Counter*-Exemplar verglichen wird. Im äußerst unwahrscheinlichen Fall einer Ungleichheit wird das Auslesen wiederholt. Die durch den gezeigten Abschnitt verursachten Zusatzkosten für den typischen Fall eines wiederholten Methodenaufrufs beläuft sich auf das Ermitteln der logischen Thread-ID (letztendlich ein Registerzugriff und einer Indirektion der .NET TLS-Implementierung), einen Zugriff auf die Hash-Tabelle (bei Kollisionsfreiheit in konstanter Zeit möglich) sowie zwei Vergleichsoperationen.
Der zweite Teil der Blockierungslogik (Abbildung 3.30) zeigt die Verwendung des zuvor ermittelten Zählers sowie die entsprechende Logik für den Fall eines Blockierungsvorgangs. Der obere Teil der Logik (Zeile 4-20) wird nur im Falle einer Blockierungsanfrage abgearbeitet, verursacht im Normalbetrieb also keine Zusatzkosten. In Zeile 22 wird der Zähler inkrementiert, bevor in Zeile 28 die funktionale Methode der verwobenen Klasse aufgerufen würde (nicht dargestellt). Der Ausschnitt zeigt die Implementierung der Sychronisationslogik als dynamischen Aspekt. In der Transaktionsbibliothek sind der Teil vor dem Aufruf der funktionalen Methode und das Dekrementieren in Zeile 29 in der *Begin*- und *End*-Methode getrennt. Nach dem Aufruf der funktionalen Methoden wird der Zähler dekrementiert. In Zeile 24 wird überprüft, ob zwischen der Überprüfung einer aktiven Blockierungsanfrage und dem Inkrementieren des Zählers eine Blockierungsanfrage eingetroffen ist. In diesem Fall wird zum Beginn der Blockierungslogik gesprungen, da auf Grund des nicht inkrementierten Zählers inzwischen

[3]Seit der Einführung von generischen Typen mit dem .NET Framework 2.0 sind Hash-Tabellen mit der Klasse *Dictionary* verfügbar.

```
1 reprocess:
2   bool initialBlockMode = blockMode;
3
4   if (blockMode){
5     int initialUpdateNr = updateNr;
6     lock (globalLock){
7       if (NoActiveThreadOnBlockSet()){
8         if (!blockMode)
9           goto reprocess;
10        if (initialUpdateNr != updateNr)
11          goto reprocess;
12
13        quiecenseReached.Set();
14        updateNr++;
15      }
16    }
17    if (!isThreadActiveOnBlockSet(curThread)){
18        blockEnd.WaitOne();
19    }
20  }
21
22  countRef.count++;
23
24  if (initialBlockMode != blockMode){
25    countRef.count--;
26    goto reprocess;
27  }
28  // Invoke target method
29  countRef.count--;
```

Abbildung 3.30: Blockierungslogik Rekonfigurationsalgorithmus

eine dynamische Rekonfiguration aktiv sein könnte. Auf dessen Beendigung müsste dieser Thread dann warten. Vor dem Rücksprung wird der Zähler wieder dekrementiert. Dieses Verfahren ist sehr typisch für die Realisierung von Thread-Sicherheit und wird u.a. auch bei der Implementierung der .NET-Klassenbibliotheken verwendet. In Abbildung 3.30 nicht dargestellt ist die zusätzliche Synchronisation nach dem Methodenaufruf (ab Zeile 29) die im Abschnitt 3.3.2 schon abstrakt beschrieben wurde. Hier ist zusätzliche Logik zum Auslösen der Rekonfiguration (siehe Zeile 13) nötig, falls ein Thread einen Methodenaufruf beendet und das Dekrementieren des Zählers genau zum rekonfigurierbaren Zustand führt.

Im Falle einer Blockierungsanfrage wurde die boole'sche Variable *blockMode* auf den Wert *true* gesetzt. In diesem Fall würde der obere Block der bedingten Anweisung ausgeführt. Da die im Algorithmus verwendete Synchronisation und die prinzipielle Struktur schon im Abschnitt 3.3.2 ausführlich beschrieben wurden, soll an dieser Stelle nur auf spezielle Sonderfälle bei der Sychronisation konkurrierender Threads eingegangen werden. Bei einer aktiven Blockierungsanforderung können potentiell mehrere Threads das Erreichen des rekonfigurierbaren Zustands feststellen (*NoActiveThreadOnBlockSet* - es ist kein Thread auf dem BlockSet aktiv). Hat der erste Thread durch die Signa-

lisierung des Ereignisobjekts (*quiecenseReached*) z.B. eine Aktualisierung ausgelöst, könnte ein zweiter Thread bei einer späteren Blockierungsanfrage das Erreichen des rekonfigurierbaren Zustands signalisieren, wenn er z.B. in Zeile 5 unterbrochen wurde, inzwischen aber wieder andere Threads auf den Kapseln aktiv sind. Aus diesem Grund wurde ein Rekonfigurationszähler eingeführt, der einzelne Blockierungsanfragen unterscheidet und durch nochmalige Überprüfung vor der Signalisierung (Zeile 10) das beschriebene Problem verhindert. Die Zusatzkosten für den zweiten Teil der Blockierungslogik beschränken sich im Normalfall auf den Vergleich bzw. den Test von Variablen an zwei Stellen sowie das Inkrementieren bzw. Dekrementieren des Zählers. Das erforderliche atomare Setzen der Variable *blockMode* kann mit Hilfe eines Synchronisationsmanagers innerhalb des Konfigurationsmanagers implementiert werden. Die Akquirierung von *globalLock* wird dann zentral durch diesen Manager verwaltet, allerdings nur im Falle einer Rekonfiguration. Für die normale Ausführung einer Anwendung wird der zentrale Synchronisationsmanager nicht benötigt. Soll nur eine Kapsel aktualisiert werden, können die Strukturen dezentral synchronisiert werden, da in diesem Fall nur auf die lokale Hash-Tabelle der Kapsel zugegriffen werden muss. In Abbildung 3.30 ist zu beachten, dass *globalLock* in Zeile 6 erst akquiriert werden kann, wenn die Variable *blockMode* in allen beteiligten Kapseln gesetzt wurde.
Kapseln können potentiell mehrere Wurzelobjekte besitzen, die jeweils mit der beschriebenen Aspektlogik verwoben werden. Für das Blockieren der eingehenden Konnektoren einer Kapsel existieren pro Kapsel die Variable *blockMode*, die Synchronisationsobjekte und auch die Thread-Liste genau einmal. Im Prinzip besteht bei der beschriebenen Synchronisationslogik kein Unterschied, ob sie konkurrierend von multiplen Threads über ein Wurzelobjekt oder über mehrere verarbeitet wird; sie funktioniert in beiden Fällen wie beschrieben. Abschließend soll an dieser Stelle noch einmal kurz beschrieben werden, wie multiple Wurzelobjekte einer Kapsel entstehen. Kapseln werden durch die Erzeugung von Exemplaren einer Hauptklasse gebildet. Bei der Ausführung von Instruktionen der Methoden dieser Klasse können weitere Objekte und auch Wurzelobjekte, durch die Verwendung einer speziellen Fabrikmethode (*UpdateManager.CreateRootObject*), entstehen.

3.6.5 Implementierung logischer Thread-IDs

Die Implementierung von rechnerübergreifenden Thread-IDs lässt sich mit dem *Interceptor*-Konzept von .NET *Remoting* sehr effizient umsetzen. Die Architektur von .NET *Remoting* erlaubt die Client- und Server-seitige Integration von zusätzlicher Logik die bei jedem entfernten Aufruf ausgeführt wird. Mit Hilfe dieses Mechanismus können logische Thread-IDs implementiert werden, indem die folgende Logik integriert wird:

Logik auf Client-Seite

- Bei der Thread-Erzeugung: Erzeugen einer neuen ID und Speicherung im TLS
- Vor dem Ausführen eines entfernten Methodenaufrufs: Auslesen der ID aus dem TLS und Übertragung der ID im Kontext der Aufrufnachricht (als impliziter Parameter)

Logik auf Server-Seite

- Auslesen der ID aus der Nachricht vom Client und Schreiben in den TLS

Der Zugriff auf die Thread-ID kann dann einfach durch das Auslesen des entsprechenden Wertes aus dem TLS implementiert werden. Als Thread-ID müssen eindeutige Werte verwendet werden. Dies lässt sich z.B. durch die Verwendung einer eindeutigen Rechner-ID (z.B. die MAC-Adresse) und einem pro Rechner eindeutigen Zähler implementieren. Der Vorteil dieses Verfahrens ist, dass die logischen Thread-IDs dezentral erzeugt werden können.
Eine Optimierung dieses Algorithmus wäre die Einführung einer zusätzlichen Indirektion für die Speicherung der logischen Thread-IDs. Da der Zugriff auf den TLS im .NET Framework im Gegensatz zum Zugriff auf die normale Thread-ID recht imperformant ist, kann die logische Thread-ID durch das Auslesen einer Hash-Tabelle mit der logischen Thread-ID als Wert realisiert werden. Die Hash-Tabelle kann dabei in einer statischen Variable pro *Application Domain* gespeichert werden und beim Eintritt in den Prozess aktualisiert werden. Im Vergleich zu einem damit verbundenen Remoting-Aufruf ist der nötige TLS-Zugriff vernachlässigbar (siehe Abschnitt 7.1). Bei normalen Methodenaufrufen bzw. Transaktionsbeginn und -ende kann der Zugriff auch die logische Thread-ID durch Benutzung der Hash-Tabelle effizient realisiert werden.

3.7 Migration

Die dynamische Rekonfiguration einer Kapsel mit verändertem Ausführungsrechner wird als Migration bezeichnet. Neben der Herstellung eines rekonfigurierbaren Zustands muss im Falle einer Migration auf dem entfernten Rechner eine Kopie der Kapsel erzeugt und ihr Zustand auf den neuen Rechner übertragen werden.
Im Falle einer Migration überträgt der primäre Konfigurationsmanager vor der Initiierung der Unterbrechungsphase die entsprechenden Assemblies der Kapsel an den neuen Ausführungsrechner, wo die Kapsel auch noch vor Beginn der Unterbrechungsphase erzeugt wird. Befindet sich die Kapsel nach dem Blockieren aller beteiligten Konnektoren in einem rekonfigurierbaren Zustand, überträgt der Konfigurationsmanager mit Hilfe der Methoden *GetState* und *SetState* der *IStateTransfer*-Schnittstelle den Zustand der Kapsel auf den neuen Ausführungsrechner.
Die Methode *GetState* traversiert jedes Attribut des Hauptobjekts und speichert eine Referenz in einem Feld von Objekt-Referenzen. Dieses wird als Rückgabewert der *GetState*-Methode an den entfernten Konfigurationsmanager per .NET-Remoting-Aufruf übertragen und dort an die entsprechende *SetState*-Methode übergeben. Die Übertragung des Zustands wird hierbei vom .NET Laufzeitsystem übernommen. Beim Verpacken (*Marshalling*) der Methodenparameter wird der Zustand eines jeden Objekts serialisiert, also in flache Strukturen überführt, und so je nach gewähltem *Formatter* übertragen. Die beteiligten Typen müssen durch das *[Serializable]*-Attribut zu erkennen geben, dass eine Serialisierung des Zustands möglich ist. Die Migrationslogik kann als Aspekt an die funktionale Kapsellogik gewoben werden. Dabei wird auch das *[Serializable]*-Attribut hinzugefügt (siehe Konfigurationsaspekt).
Im Adapt.Net-Projekt stand die Übertragung von externen Betriebsmitteln nicht im Mittelpunkt. Bei einer Migration müssen z.B. benutzte Dateien oder Datenbankverbindungen an den entfernten Rechner übertragen werden. Für die Übertragung von externen Betriebsmitteln eignen sich die von Peter Tröger entwickelten *Post-Migration*-Handler [Tröger und Polze 2003]. Dabei werden für Elemente bestimmter Typen auf der Ursprungsseite Handler ausgeführt, die z.B. zusätzliche Logik installieren, welche Aufrufe an den neuen Ausführungshost der Kapsel weiterleiten oder Dateien auf den

Zielrechner übertragen. Im Gegensatz zur Arbeit von Peter Tröger ist es mit dem vorgestellten Verfahren möglich, Kommunikationsendpunkte effizient zu migrieren, da diese von der vorgestellten Aspektlogik verwaltet werden und keine Weiterleitung von Nachrichten vom Ursprungsrechner notwendig ist.

3.7.1 Migration von Kapseln mit eigenen Threads

Die Übertragung des Ausführungszustands von Threads ist mit den verfügbaren Basistechnologien nicht möglich, da der direkte Zugriff auf den Kontext (Stack, Register, Befehlszähler) nicht ohne eine Modifikation der Laufzeitumgebung realisierbar ist. Kapseln, die migriert werden sollen, sind deshalb an wenige Einschränkungen gebunden. Alle Threads einer Kapsel müssen, während sie nicht auf der Kapsel aktiv sind, unabhängig vom Ausführungskontext sein. Dies ist bei einem zyklischen und ereignisbasierten Abarbeitungsmodell der Fall, wie sie bei den in dieser Arbeit untersuchten Steuerungssystemen und interaktiven, graphischen Anwendungen mit der .NET-*WinForms*-Bibliothek Verwendung finden.

3.8 Dynamische Integration von Diensten

Im Rahmen der Diplomarbeit von Sebastian Tiedt [Tiedt 2007] wurde die dynamische Dienstintegration in mobilen Ad-hoc Netzwerken untersucht. Das Dienst-Erkundungs-Protokoll (*service discovery protocoll*) UPnP wurde erweitert, um die Auswahl alternativer Dienstimplementierungen in Abhängigkeit gemessener Umgebungseigenschaften zu bestimmen. Hierfür wurde die XML-basierte Schnittstellenbeschreibung um Profile erweitert, welche die Einsatzbedingungen der Dienste beschreiben. Durch die Unterstützung externer Referenzen in Adapt.Net ist es mit dieser Technik möglich, zuvor nicht bekannte Dienste in eine Anwendungskonfiguration zu integrieren. Durch die Nutzung der Protokollerweiterung können gefundene Dienste entsprechend der ermittelten Umgebungssituation selektiert werden.

3.9 Zusammenfassung

Im Rahmen dieses Kapitels wurde ausgehend von einem Modell der Laufzeitumgebungen moderner Komponentenplattformen wie der Java- und .NET-Plattform die Konfiguration verteilter, komponentenbasierter Anwendungen eingeführt. Eine Konfiguration beschreibt die Zusammensetzung einer Anwendung aus Kapseln, die auf verschiedenen Rechnern eines verteilten Systems ausgeführt werden und über Konnektoren mittels Objektfernaufrufen kommunizieren. Der unabhängige Austausch einzelner Komponenten zur Aktivierung alternativer Implementierungen (mit gleicher Schnittstelle) mit angepassten nicht-funktionalen Eigenschaften wird durch das eingeführte Konzept der Kapseln unterstützt. Eine Kapsel fasst eine Menge von Objekten gleicher Deployment-Einheiten zusammen. Wurzelobjekte definieren Kapselgrenzen, über die Threads in die Verarbeitung von Objektmethoden eintreten. Das eingeführte Modell unterstützt dabei auch mehrere Threads pro Kapsel.
Die Konfiguration einer Anwendung bestimmt wesentlich ihre nicht-funktionalen Eigenschaften - die Änderung einer Konfiguration während der Anwendungslaufzeit ist damit ein wichtiger Baustein adaptiver Anwendungen. Da ohne eine Manipulation der eingesetzten Laufzeitumgebungen kein Zugriff auf den Stack aktiver Threads möglich

ist, wurde im Rahmen dieser Arbeit ein Verfahren entwickelt, bei dem die dynamische Rekonfiguration durchgeführt wird, wenn kein Thread auf den beteiligten Elementen aktiv ist. Im Rahmen der Arbeit wurden verschiedene Synchronisationsverfahren vorgestellt, mit denen die Erkennung und auch die Herbeiführung eines rekonfigurierbaren Zustands möglich ist. Mit Hilfe der Wurzelobjekte sind die Integrationspunkte für die Synchronisationslogik gegeben, da über sie Threads in eine Kapsel eintreten. Besonders hervorzuheben ist ein neu entwickeltes Verfahren, welches laufende Methodenaufrufe von Threads durch spezielle Zähler in den Kapseln speichert, die verwendet werden, um zu erkennen, ob ein Thread während der Herbeiführung des rekonfigurierbaren Zustands blockiert werden darf oder weiterlaufen muss, um laufende Methodenaufrufe in den von der Rekonfiguration betroffenen Kapseln zu beenden. Im Vergleich zu existierenden Ansätzen hat das neue Verfahren geringere Zusatzkosten für funktionale Methodenaufrufe und unterstützt zusätzlich Anwendungen mit mehreren Threads und zyklischen Aufrufbeziehungen. Das Verfahren funktioniert auch für verteilte Anwendungen, da für die Identifizierung von Threads logische Thread-IDs eingesetzt werden, die systemweit eindeutig sind.
Der entwickelte Rekonfigurationsansatz wurde exemplarisch auf der .NET-Plattform im Rahmen des Adapt.Net-Projekts implementiert. Unterstützte Konfigurationsänderungen umfassen unter anderem das Hinzufügen neuer Kapseln, die Änderungen von Parametern, aber auch die Migration von Kapseln sowie die Änderung der Verbindungsstrukturen einer Anwendung. Im Rahmen des Adapt.Net-Projekts wurde ein neues Programmiermodell für adaptive Anwendungen entworfen, das die Erstellung von Anwendungskonfigurationen erleichtert. Der Entwickler annotiert bei der Implementierung von Komponenten Kommunikationsendpunkte und Komponentenparameter, die später über ein graphisches Werkzeug verwendet werden, um aus den Komponenten eine Anwendungskonfiguration zu erstellen. Auch das Deployment im Falle verteilter Komponenten wird von den entwickelten Infrastrukturkomponenten realisiert.
Eine weitere wichtige Leistung dieser Arbeit ist der Einsatz der aspektorientierten Programmierung (AOP) zur Integration konfigurationsspezifischer Logik in die funktionalen Komponenten. Die komplexe Implementierung von Konfigurationsschnittstellen für die Erstellung von Kommunikationsverbindungen und den Zugriff auf Komponentenparameter, aber vor allem die Integration der für die dynamische Rekonfiguration nötigen Synchronisationslogik konnte im Rahmen dieser Arbeit wiederverwendbar als Aspekt implementiert werden und kann so die Entwicklung adaptiver Anwendungen beschleunigen. Zusätzlich ist mit dem Einsatz von AOP im Rahmen dieser Arbeit ein komplexes Einsatzszenario für AOP gefunden.

4 Dynamische Aktualisierung - Spezialfall der dynamischen Rekonfiguration

Nachdem im vorangegangenen Kapitel elementare Rekonfigurationsoperationen wie das Hinzufügen und Entfernen von Kapseln, Änderungen der Verbindungsstruktur und auch die Migration von Kapseln vorgestellt wurden, soll in diesem Kapitel die komplexe Rekonfigurationsoperation der dynamischen Aktualisierung diskutiert werden. Bei einer dynamischen Aktualisierung liegen die Assemblies, in denen die Typen der Objekte einer Kapsel definiert sind, in einer neuen Version vor und müssen in die Laufzeitstrukturen integriert werden. Änderungen können das Verhalten (Instruktionen der Methoden, Methodensignaturen, neue und entfernte Methoden), Typbeziehungen (Vererbung, Implementierung von Schnittstellen) und das Datenlayout (hinzugefügte, entfernte, geänderte Attribute) betreffen.

Im Rahmen dieser Arbeit wurde eine neue Technik entwickelt, welche die dynamische Aktualisierung portabel, also ohne die Manipulation der unterliegenden Laufzeitumgebung mit Hilfe der Reflexionsfunktionalität moderner Komponentenplattformen ermöglicht. Ein besonderer Vorteil des im vorangegangenen Kapitel vorgestellten Rekonfigurationsalgorithmus ist, dass im rekonfigurierbaren Zustand kein Kapselzustand auf den Stacks der Threads existiert. Auf den Stacks können potentiell Referenzen auf Objekte einer Kapsel gespeichert sein, die bei einer Aktualisierung untersucht werden müssten. Da der Zugriff auf den Stack in den eingesetzten Laufzeitumgebungen nicht effizient unterstützt wird, wird die Aktualisierung im rekonfigurierbaren Zustand ausgeführt, in dem alle Objekte einer Kapsel von den Wurzelobjekten aus erreichbar sind. So können die Typen aller Objekte untersucht werden und im Falle einer neuen Version ein neues Objekt erzeugt und der Zustand des alten Objekts kopiert werden. Das Verfahren ähnelt den bei der *garbage collection* eingesetzten Techniken. Das *root-sets* wird durch den eingesetzten Rekonfigurationsalgorithmus nur durch die Wurzelobjekte gebildet, womit der Zugriff auf den Stack entfällt.

Für die Realisierung der dynamischen Aktualisierung wurde das Projekt DSUP (*dynamic software update platform*) als Ergänzung zu Adapt.Net geschaffen. Dort wurde entkoppelt ein Algorithums für die Traversierung des Objektgraphen einer Kapsel implementiert und entsprechende Techniken für den Zustandstransfer untersucht. Schlussendlich wurden die in DSUP implementierten Mechanismen in Adapt.Net integriert, um dort adaptive Anwendungen mit Hilfe alternativer Versionen von Softwarekomponenten erstellen und ausführen zu können.

Dieses Kapitel folgt im logischen Aufbau dem vorangegangenen, in dem zunächst grundlegende Modellelemente eingeführt werden, gefolgt von einer abstrakten Beschreibung der entwickelten Algorithmen und einer abschließenden Diskussion der Implementierung auf Basis der *Common Language Infrastructure*.

4.1 Modellerweiterung

Für die Beschreibung der Aktualisierungsoperation wird zunächst der Versionsoperator eingeführt. Dieser bildet Objekte und Typen auf die Version der Assembly, in der sie definiert wurden, ab. Der in der Definition des Versionsoperators verwendete **Typoperator** $\mathfrak{T}$ bildet ein Objekt $\in$ O auf seinen definierenden Typ $\in$ T ab ($\mathfrak{T} : O \rightarrow T$). Zusätzlich existiert für jedes Attribut eines Objekts eine Typinformation, welche bei der Attributdeklaration definiert wurde (z.B. *MyClass myRef*). Im Falle von Refenztypen können diese Typen in separaten Assemblies definiert worden sein. Auch für die Typinformation eines Attributs lässt sich die Version der definierenden Assembly ermitteln. In diesem Fall bestimmt der Operator $\mathfrak{T}$ den Deklarationstyp eines Attributs (Attr) eines Typs $\in$ T, wenn dieses einen Referenztyp besitzt ($\mathfrak{T} : (T, Attr) \rightarrow T$). Das Attribut wird bei der Typdeklaration durch eine Zeichenkette eindeutig identifiziert.
Im vorangegangenen Kapitel wurde die Projektion **aname** definiert, die den Namen einer Assembly beschreibt, welcher eine Zeichenkette und eine Versionsnummer enthält. Die Projektion **types** enthält die Menge der in einer Assembly definierten Typen. Mit den beschriebenen Relationen kann der Versionsoperator V definiert werden:

Definition 4.1.1 (Versionsoperator) *Sei A eine Menge von Assemblies, O eine Menge von Objekten und Version eine Zeichenkette, welche die Versionsnummer enthält. Dann ist der Versionsoperator V definiert durch:*

- $V : A \rightarrow Version$ *mit* $V(A_x) = version$ *für* $(name, version) \in aname(A_x)$
- $V : O \rightarrow Version$ *mit* $V(O_x) = V(A_x)$ *mit* $\mathfrak{T}(O_x) \in types(A_x)$
- $V : T \rightarrow Version$ *mit* $V(t) = V(A_x)$ *mit* $t \in types(A_x)$
- $V : (T, Attr) \rightarrow Version$ *mit* $V(T, Attr) = V(\mathfrak{T}(T, Attr))$

mit $A_x \in A$ *und* $O_x \in O$.

Eine dynamische Aktualisierung wird auf der Basis von Assemblies durchgeführt, weil sie in der eingesetzten Laufzeitumgebung die Einheit der Versionierung darstellen. Über die Assemblies sind die in ihnen definierten Typen und damit die daraus gebildeten Objekte versioniert, was durch den Versionsoperator formalisiert wurde. Die Typen der Objekte des Objektgraphen einer Kapsel werden in einer Reihe von Assemblies definiert. Von einer dynamischen Aktualisierung wird gesprochen, falls mindestens eine dieser Assemblys in einer neuen Version vorliegt. Eine Assembly kann in einer neuen Version in einer zweiten Datei (z.B. in einem weiteren Verzeichnis) vorliegen oder im *Global Assembly Cache*, einer Datenstruktur, welche die Speicherung von Assemblies in mehreren Versionen unterstützt. Eine dynamische Aktualisierung wird durch die Abbildung einer Menge von Assemblies der Ausgangskonfiguration auf eine zweite Menge, den Assemblies der neuen Konfiguration, formuliert. Diese wird als **Aktualisierungszuordnung** (*update-mapping*) bezeichnet.
Die Kardinalität der Mengen der Definitions- und Wertebereiche einer Aktualisierungszuordnung kann im Prinzip verschieden sein. Ein Beispiel hierfür ist eine Hilfsmethode, die aus einer zweiten Assembly aufgerufen wird, aber in der neuen Konfiguration nicht mehr benötigt wird. Da es sich bei einer dynamischen Aktualisierung um eine Laufzeitmanipulation von Daten handelt, können sich Fehler bei der Definition der Aktualisierungszuordnung fatal auswirken. Eine Aktualisierungszuordnung sollte deshalb

statisch überprüfbar sein. Um dies zu ermöglichen, wird an dieser Stelle zunächst davon ausgegangen, dass sämtliche Typen der Assemblies der alten Konfiguration auch in der neuen definiert sind. Dies ist nötig, da zunächst nicht entschieden werden kann, welche Typen für die Exemplarerzeugung von Wurzelobjekten benötigt werden und ob potentiell sämtliche Typen im Objektgraphen Verwendung finden. Diese Einschränkung kann später aufgehoben werden, da durch die vollständige CIL-Code-Analyse (Finden der Wurzeltypen) anhand der Abhängigkeitsbeziehung von Typen nicht verwendete Typen ermittelt werden können. Die Kardinalität von Definitions- und Wertbereich der Aktualisierungszuordnung ist also zunächst gleich.
Für die Spezifikation einer dynamischen Aktualisierung wird die Aktualisierungszuordnung eingeführt, die für die statische Überprüfbarkeit der Korrektheit zusätzliche Anforderungen erfüllen muss. Für die Definition werden einige aus dem vorangegangenen Kapitel eingeführten Definitionen verwendet: Die **Typabhängigkeitsrelation** $\triangleright$ beschreibt die Abhängigkeit zweier Typen z.B. durch eine Vererbungsbeziehung. Die Projektion **tname** bestimmt den Namen eines Typs, **aname** den Namen einer Assembly und **refs** beschreibt die Menge der Namen der von einer Assembly referenzierten Assemblies.

Definition 4.1.2 (Aktualisierungszuordnung) *Sei A eine Menge von Assemblies der Ausgangskonfiguration und B eine Menge von Assemblies der Zielkonfiguration. Dann wird eine dynamische Aktualisierung durch die injektive Abbildung AkZu: $\mathcal{P}(A) \rightarrow \mathcal{P}(B)$ formuliert. Die Abbildung ist an eine Reihe von Einschränkungen geknüpft:*

- $\forall A_x \in A\ \forall t \in types(A_x)$ *mit* $B_x = AkZu(A_x) : \exists s \in types(B_x)$ *mit tname(t) = tname(s) (**Typ-Existenz**)*
- $\forall t \in types(B_x), s \notin types(B_x)$ *mit* $t \triangleright s : \exists B_y$ *mit* $aname(B_y) \in refs(B_x)$ *und* $s \in types(B_y)$ *für alle* $B_x, B_y \in B$. *(**Typ-Integrität**)*
- *Die Typ-Integrität gilt auch für A. Die Formulierung ist analog zur Typ-Integrität von B mit B=A.*

Zusammengefasst bedeuten die Einschränkungen an die Aktualisierungszuordnung, dass jeder Typ der alten Konfiguration in der neuen Konfiguration vorhanden sein muss (Typ-Existenz). Die Typ-Integrität formuliert die versionskorrekte Referenzierung der Assemblies der beiden Mengen. Dies bedeutet, wenn ein Typ einen aktualisierten Typen aus einer zweiten Assembly direkt verwendet, muss auch die Assembly dieses Typs in der Aktualisierungszuordnung eingeschlossen werden, da die neue Version der zweiten Assembly eine neue Version dieser Assembly bedingt (Es muss neu gelinkt werden.). Die Versionsnummern der referenzierten Assemblies innerhalb des Assembly-Manifests und der vorliegenden Datei(en) müssen für eine Kapsel immer übereinstimmen. Eine statische Überprüfung der genannten Einschränkungen ist leicht implementierbar, indem Definitions- und Wertbereich der in der Abbildung spezifizierten Mengen separiert überprüft werden.
Ein Aktualisierungsalgorithmus muss sicherstellen, dass die referenzielle Integrität innerhalb einer Kapsel nicht verletzt wird. Darunter ist konkret zu verstehen, dass jedes Attribut mit Referenzsemantik innerhalb einer Kapsel auf ein versionskompatibles Objekt verweist. Ein Konflikt kann durch das dynamische Nachladen von Assemblies einer neuen Version entstehen. Wird z.B. in einer Klasse eine Objektreferenz als Attribut

definiert mit einem Typ aus einer zweiten Assembly, z.B. *MyClass myObjRef*, wird bei Zuweisungen an dieses Attribut eine Typüberprüfung vorgenommen, die auch die Version der Typen einschließt. Die Typinformation des Attributs wird statisch bei der Übersetzung mit der Originalversion der Assembly versehen. Wird nun eine neue Version der Implementierung von *MyClass* geladen und der Zustand der alten Version übertragen, kommt es bei einer Zuweisung von *myObjRef* mit dem Objekt der neuen Version zu einer strukturierten Ausnahme auf Grund des Typkonflikts. Der Typ einer Referenz muss also immer mit dem Typ des Ziels übereinstimmen. Diese Bedingung soll zunächst formalisiert werden mit der im vorangegangenen Kapitel eingeführten Projektion **attributes** als der Menge der Attribute eines Typs, dem Versionsoperator V und dem Typoperator $\mathfrak{T}$ aus diesem Kapitel.

Definition 4.1.3 (Referenzielle Integrität) *Sei O die Menge der Objekte einer Kapsel, dann muss zu jedem Zeitpunkt der Ausführung gelten: $\forall O_1, O_2 \in O$ mit $attr \in attributes(\mathfrak{T}(O_1))$ und attr verweist auf $O_2 : V(\mathfrak{T}(O_1), attr) = V(O_2)$.*

Nach der Einführung des Versionsoperators und der Aktualisierungszuordnung wird nun das entwickelte Aktualisierungsverfahren vorgestellt. Ein wichtiges Korrektheitskriterium des Verfahrens ist der Erhalt der referenziellen Integrität.
Eine dynamische Aktualisierung einer Kapsel kann durch eine Traversierung des Objektgraphen einer Kapsel ausgehend von den Wurzelobjekten implementiert werden. An Hand einer Aktualisierungszuordnung kann für jedes Objekt überprüft werden, ob sich die Version eines Typs (ausgehend von der Assembly) geändert hat. Ggf. kann eine neue Version des Objekts erzeugt und der Zustand des alten Objekts übertragen werden. Die Übertragung des Zustands kann durch eine Kopie aller Attribute eines Objekts, welche den Objektzustand repräsentieren, erfolgen. Zusätzlich müssen die Identitäten der Objekte durch den Erhalt der Referenzierungsbeziehungen der Objekte innerhalb der Kapsel erhalten bleiben.
An dieser Stelle soll das entwickelte Aktualisierungsverfahren in algorithmischer Darstellung beschrieben werden, um einige elementare Eigenschaften diskutieren zu können. Die komplexe Implementierung wird im nächsten Abschnitt genauer beschrieben. Der nachfolgende Algorithmus zeigt die vereinfachte Version einer dynamischen Aktualisierung eines Objektgraphen. Dieser wird in der Funktion *AktualisiereObjektGraph* mit dem Parameter eines Wurzelobjekts implementiert. Der Rückgabewert ist eine Referenz auf den aktualisierten Objektgraphen. Die Aktualisierungszuordnung liegt in Form einer Hash-Tabelle in der Variablen *AktualisierungsZuordnung* vor. Die dynamische Aktualisierung einer Kapsel erfolgt durch den Aufruf dieser Funktion für jedes Wurzelobjekt.
Im Wesentlichen zeigt der Algorithmus das zuvor beschriebene Verfahren der Traversierung des Objektgraphen. Die Methode *AktualisiereObjektGraph* wird für jedes Attribut der Menge der Wurzelobjekte aufgerufen. Im Falle des Vorhandenseins einer neuen Version wird ein neues Objekt erzeugt (Zeile 5). Ab Zeile 10 wird jedes Attribut der Objekte untersucht und Referenzen durch einen rekursiven Aufruf verfolgt, was letztendlich die Traversierung des Graphen realisiert. Das Ergebnis der Untersuchung wird in dem Attribut des Ausgangsobjekts gespeichert. Im Falle primitiver Daten, die den eigentlichen Zustand der Objekte ausmachen (neben den Objektidentitäten), wird der Wert des Attributs des alten Objekts in das entsprechende Attribut des neuen Objekts kopiert. Um die Terminierung des Algorithmus auch im Falle zyklischer Objektreferenzen gewährleisten zu können, wird jedes untersuchte Objekt in

Algorithmus 6 Dynamische Aktualisierung: object AktualisiereObjektGraph(object o)

```
1: if (o, x) ∈ TraversierteObjekte then
2:    return x
3: end if
4: if V(o) ∈ AktualierungsZuordnung then
5:    o_neu ← ErzeugeNeueVersion(o)
6: else
7:    o_neu ← o
8: end if
9: TraversierteObjekte ← (o, o_neu)
10: for all a ∈ Attribute(o) do
11:    if a ist Objektreferenz then
12:       o_neu(a) ← AktualisiereObjektGraph(a)
13:    else
14:       o_neu(a) ← a
15:    end if
16: end for
17: return o_neu
```

einer Hash-Tabelle gespeichert (Zeile 9). Wird ein Objekt ein zweites Mal untersucht, wird das Ergebnis der ersten Untersuchung ermittelt und zurückgegeben (Zeile 1,2).

Der vorgestellte Algorithmus **terminiert**, weil jede Kapsel eine endliche Menge von Objekten besitzt (auf Grund des beschränkten Hauptspeichers) und jedes Objekt nur genau einmal die Untersuchung anderer Objekte veranlasst. Vor dem einzigen rekursiven Aufruf - also dem Veranlassen einer neuen Untersuchung - wird in Zeile 9 garantiert das aktuelle Objekt gespeichert. Wird später das Objekt noch einmal untersucht, wird die Methode *AktualisiereObjektGraph* in Zeile 2 beendet. Damit terminiert der Algorithmus auf Grund der endlichen Zahl an Objekten in endlicher Zeit.

Der Algorithmus **erhält die referenzielle Integrität**. Um diese Eigenschaft zu beweisen, soll zunächst das Erreichen aller Objekte einer Kapsel gezeigt werden. Der Algorithmus traversiert ausgehend von den Wurzelobjekten rekursiv alle Objektreferenzen. Die Definition einer Kapsel fordert genau, dass im Zustand der Inaktivität alle Objekte einer Kapsel durch abgeschlossene Referenzierung erreicht werden können. Folglich werden durch den Algorithmus alle Objekte einer Kapsel erreicht. Kombiniert man die Einschränkung an die Aktualisierungszuordnung und die Erreichbarkeit aller Objekte, kann der Erhalt der referenziellen Integrität leicht gezeigt werden. Da in der Aktualisierungszuordnung immer nur zusammenhängende abgeschlossene gültige Mengen von Assemblies enthalten sein können, das heißt neue und alte Mengen der Assemblies referenziell integer sind, können auch nur referenziell integre Objektgraphen entstehen. Die Referenzierung von Objekten ist über die Abhängigkeitsbeziehung von Typen formalisiert worden, die Grundlage für die Formulierung der Einschränkung der Aktualisierungszuordnung ist. Da der Übergang des Objektgraphen einer Kapsel atomar vollzogen wird, ist die strukturelle Integrität vor und nach der Aktualisierung gewährleistet.

4.2 Implementierung

Nach der abstrakten Vorstellung des Aktualisierungsverfahrens, welches auf aktuelle Komponentenplattformen wie die Java-Plattform und die *Common Language Infrastructure* (CLI) anwendbar ist, soll in diesem Abschnitt exemplarisch die konkrete Implementierung auf Basis der CLI vorgestellt werden. Einen Schwerpunkt des DSUP-Projekts, welches als Ergänzung zu Adapt.Net initiiert wurde, lag auf der Untersuchung, inwieweit sich der Zustandstransfer bei der dynamischen Aktualisierung automatisiert, das heißt ohne die Implementierung aktualisierungsspezifischer Logik durch den Entwickler, realisieren lässt. Neben der Integration der entwickelten Techniken in die Adapt.Net Laufzeitumgebung wurden im DSUP-Projekt zusätzliche Laufzeitkomponenten entwickelt wurden, welche die dynamische Aktualisierung von Software unterstützt und damit unter Anderem zum Einspielen neuer Softwareversionen während der Anwendungslaufzeit z.B. zur Fehlerbehebung verwendet werden kann. Diese sollen im Rahmen dieses Abschnitts vorgestellt werden.

Zusätzlich wurden in DSUP die dynamische Aspektaktualisierungen und -aktivierung untersucht. Hierbei sollen Aspekte während der Anwendungslaufzeit aktiviert bzw. deaktiviert, aber auch ihre Implementierung geändert werden können. Dies eröffnet die Möglichkeit, Alternativverhalten von Kapseln mit Hilfe der aspektorientierten Programmierung zu realisieren. Ein Beispiel hierfür ist die Einführung von *Caches* für durchgeführte Methodenaufrufe, um die Antwortzeiten einer Kapsel zu reduzieren.

In DSUP wurde das Laufzeitmodell von Adapt.Net übernommen. Kapseln interagieren über Schnittstellen und ihr Zustand repräsentiert sich durch einen Graph von Objekten, der aus einer Menge von Wurzelobjekten hervorgeht. Ein Element dieser Menge, das Hauptobjekt, exportiert die Schnittstelle der Kapsel. Für die Herstellung eines rekonfigurierbaren Zustands wurde der im vorherigen Kapitel beschriebene Algorithmus verwendet. Hierfür konnten die Aspektimplementierungen aus dem Adapt.Net-Projekt direkt verwendet werden.

4.2.1 Auslösen dynamischer Aktualisierungen

Zunächst implementiert die Klasse *CapsuleUpdater* (Aktualisierungsmanager) die Methode *object UpdateObjectGraph(object node)*, welche die Traversierung des Objektgraphen und den Zustandstransfer enthält. Die Implementierung wird in den folgenden Abschnitten ausführlich vorgestellt. Der Adapt.Net-Konfigurationsmanager benutzt diese Funktionalität, die in einer Bibliothek bereitgestellt wird, um Kapseln ggf. zu aktualisieren.

Dynamische Aktualisierungen können in DSUP durch das Kopieren der Assemblies der neuen Version in das Anwendungsverzeichnis ausgelöst werden. Durch das Erzeugen der Wurzelobjekte einer Kapsel mit der *CreateRootObject*-Methode des Aktualisierungsmanagers wird auch die Überwachung des Anwendungsverzeichnisses veranlasst. Mit Hilfe der Klasse *FileSystemWatcher* wird ein Ereignis ausgelöst, wenn sich der Inhalt des überwachten Verzeichnisses ändert. Die Adapt.Net-Assembly-Bibliothek *AssemblyCache* sorgt dafür, dass Assemblies zunächst in den Hauptspeicher geladen werden und von dort die Kapseln erzeugt werden. Dadurch ist es möglich, die Assemblies einer Anwendung auch während der Laufzeit einer Anwendung zu überschreiben, da die entsprechende Datei nach dem Einlesen in den Speicher nicht mehr geöffnet ist und damit keine Sperre auf der Datei existiert.

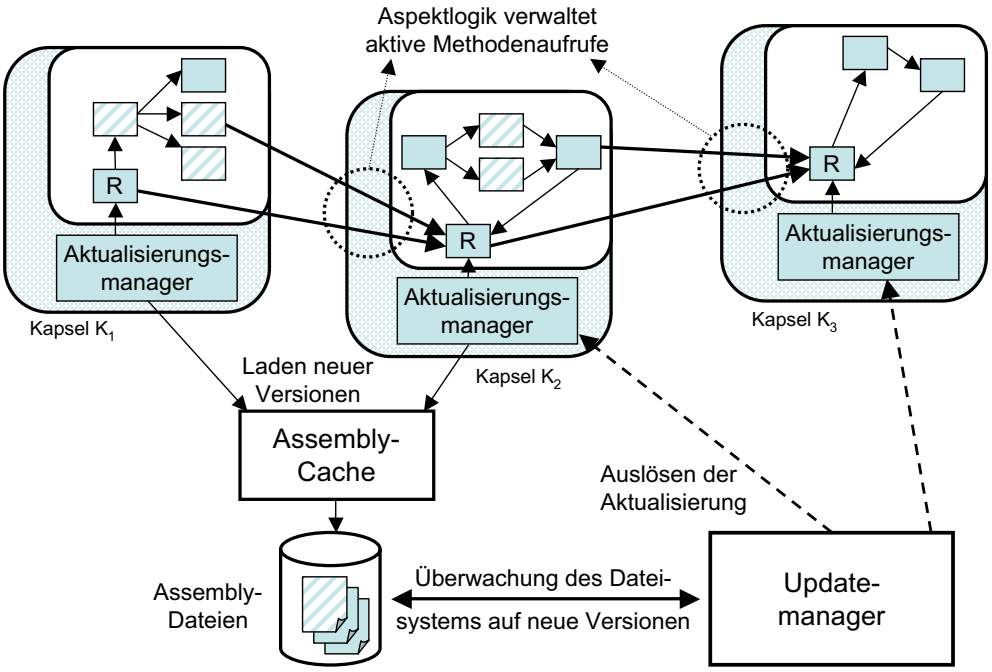


Abbildung 4.1: DSUP Architektur

Abbildung 4.1 zeigt die Architektur der DSUP-Laufzeitumgebung. Der Objektgraph einer jeden Kapsel wird durch den Aktualisierungsmanager traversiert und der Zustand ggf. transferiert. Die Abbildung zeigt die Überwachung des Anwendungsverzeichnisses durch den Updatemanager. Dieser löst die dynamische Aktualisierung einzelner Kapseln über ein Exemplar des Aktualisierungsmanagers aus. Die Synchronisation von Rekonfigurations- und Anwendungslogik wird in die Kapseln durch den *SychronizationAspect* realisiert, wie er auch in Adapt.Net bei Kapseln mit eigenen Threads zum Einsatz kommt (siehe Abschnitt 3.6.1).

4.2.2 Die Traversierungsphase

Nachdem in der Synchronisationsphase die Kapseln in einen rekonfigurierbaren Zustand überführt wurden, kann die eigentliche Aktualisierung ausgeführt werden. Die Klasse *CapsuleUpdater* implementiert die Traversierung des Objektgraphen einer Kapsel. Dynamische Aktualisierungen werden in Form von Aktualisierungszuordnungen beschrieben. In einer Hash-Tabelle werden Assembly-Namen (als Schlüssel) einer neuen Assemly-Version (in Form eines Assembly-Namens) zugeordnet. Während der Graphtraversierung wird der Typ jedes Objekts an Hand seines Assembly-Namens auf das Vorhandensein einer neuen Version geprüft. Details zum Erzeugen der Objekte in der neuen Version sowie dem Zustandstransfer folgen in diesem Abschnitt. Die Aktualisierung des Objektgraphen einer Kapsel wird durch die Methode *static object UpdateObjectGraph(object node,Hashtable updMapping)* des *CapsuleUpdaters* implementiert. Sie wird für jedes Wurzelobjekt einer Kapsel aufgerufen.
Eine stark vereinfachte Version des Traversierungsalgorithmus ist in Abbildung 4.2 dargestellt. Zunächst wird in Zeile 2 geprüft, ob eine neue Version des Objekttyps vorliegt. Ist dies der Fall, wird ein Exemplar des neuen Typs erzeugt. Dabei muss beachtet werden, dass kein Konstruktor ausgeführt werden darf, da dies unerwünschte Seiteneffekte haben könnte (z.B. Anlegen neuer Dateien, Netzwerkkommunikation).

```
1 object UpdateObjectGraph(object node, Hashtable updMapping) {
2   Type newType = GetUpdate(node.GetType(),updMapping);
3   if(newType!=null)
4     object updNode=FormatterServices.GetUninitializedObject(
        newType);
5
6   foreach(FieldInfo _field in type.GetFields())
7   {
8     if(LeafNode(_field.GetValue(node)) {
9       if(updNode!=null)
10        _updatedField.SetValue(updNode,_field.GetValue(node));
11    }
12    else
13    if(updatedNode!=null) {
14      _updatedField.SetValue(updatedNode,
15          UpdateObjectGraph(_field.GetValue(node),updMapping);
16    }
17    else {
18      _field.SetValue(node,
19        UpdateObjectGraph(_field.GetValue(node),updMapping);
20    }
21  }
22  return (updNode==null):node?updNode;
23 }
```

Abbildung 4.2: Attributweise Traversierung

Der *new*-Operator der CLI übernimmt nicht nur die Allokation von Speicher für den Objektzustand und die Initialisierung der Objektdatenstrukturen (Virtuelle Methodentabellen, Interface-Tabellen, Vererbungsstrukturen, Typinformationen und Metadaten), sondern gleichzeitig den Aufruf eines Konstruktors des neuen Typs. Deshalb wird für die Objekterzeugung die Methode *FormatterServices.GetUninitializedObject* der CLI-Klassenbibliothek verwendet, welche die Anforderungen erfüllt. Diese Methode existiert für die Serialisierung von Objekten in der CLI aus dem gleichen Grund, warum sie in dieser Arbeit verwendet wurde. *FormatterServices.GetUninitializedObject* erzeugt ein Objekt mit initialisierten Laufzeitstrukturen ohne gültigen Objektzustand. Dieser wird im zweiten Teil des Algorithmus durch eine attributweise Kopie des alten Objekts realisiert.

Die Traversierung des Objektgraphen erfolgt mit Hilfe der CLI-Reflexionsfunktionen. Ausgehend von einem Objekt wird jedes Attribut der Objekte wiederum traversiert. In Zeile 6 wird über alle Attribute des aktuellen Objekts iteriert, welche über die Methode *GetFields* des Typobjekts abgefragt werden. Dabei wird unterschieden zwischen primitiven Typen und weiteren Objektreferenzen. Ein Attribut primitiven Typs (Blatt im Objektgraphen - engl. *leaf*) wird im Falle einer Aktualisierung, durch Aufruf der *SetValue*-Methode, in das neue Objekt kopiert (Zeile 10). Für Attribute, die wiederum Objektreferenzen sind, wird die Methode *UpdateObjectGraph* rekursiv aufgerufen und der Objektgraph damit weiterverfolgt. Dabei sind wieder zwei Fälle zu unterscheiden: Nach erfolgter Aktualisierung wird der Rückgabewert, der potentiell auch in einer neuen Version vorliegt, in das uninitialisierte Objekt injiziert. Im anderen Fall wird der

Rückgabewert in das untersuchte Objekt kopiert. Dieser Schritt wird im vollständigen Algorithmus nur ausgeführt, wenn eine Aktualisierung des referenzierten Objekts vorliegt.

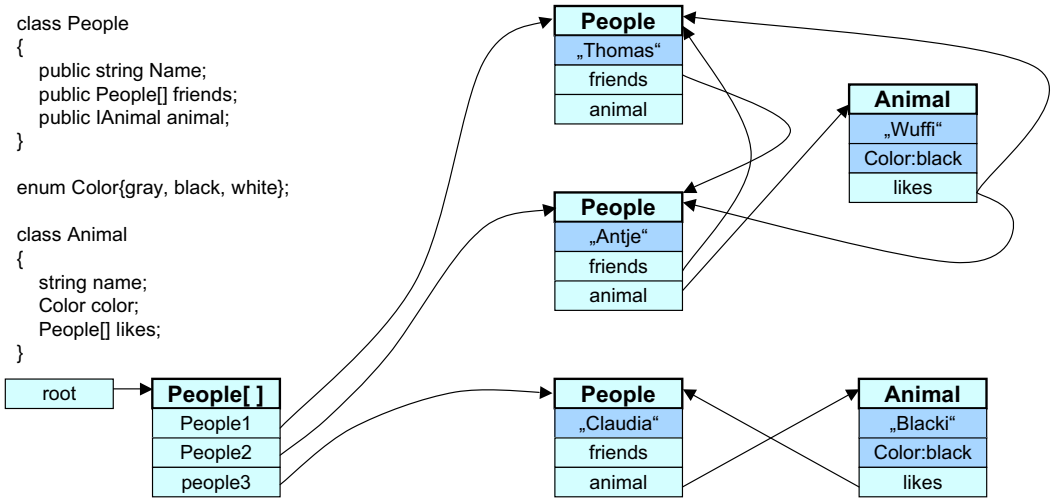


Abbildung 4.3: Beispiel: Traversierungsphase

Die Traversierung des Objektgraphen soll nun an einem kurzen Beispiel erläutert werden. Abbildung 4.3 zeigt einen Objektgraphen und die Deklaration der verwendeten Typen, die als Beispiel dienen sollen. Wurzelobjekte sind in diesem Fall Objekte vom Typ *People*, welche als Elemente in einem Feld gespeichert sind. In Abbildung 4.2 nicht dargestellt wird nun jedes Element des Feldes traversiert. Als Erstes wird die Traversierungsmethode mit dem ersten Objekt vom Typ *People* aufgerufen. Dort werden wieder alle Attribute dieses Typs traversiert. Zunächst wird die Zeichenkette „Thomas“ im Falle einer Aktualisierung in das neu erzeugte Objekt kopiert und nicht weiter untersucht. Als Nächstes wird das Attribut *friends* untersucht, wobei es sich wieder um eine Referenz auf ein Feld handelt. Es wird zunächst eine Referenz auf ein weiteres Objekt vom Typ *People* verfolgt. Hier werden wieder alle Attribute traversiert. Nach der Zeichenkette „Antje“ folgt eine Referenz auf das als erste untersuchte Objekt vom Typ *People*. Um an dieser Stelle eine Endlosschleife im Algorithmus zu vermeiden, müssen Zyklen im Objektgraphen erkannt und behandelt werden.

Die **Zyklenerkennung** wurde durch die Speicherung der Referenzen aller untersuchten Objekte in einer Hash-Tabelle (*visistedNodes*) implementiert. Als Schlüssel muss dafür ein eindeutiger Identifikator für jedes Objekt erzeugt werden. Der Identifikator muss dabei eindeutig für verschiedene Objektreferenzen sein und darf auch bei inhaltsgleichen Objekten denselben Wert besitzen. Die *GetHashCode()*-Methode, die jedes Objekt in der CLI implementiert, kann hierfür nicht verwendet werden, da der Rückgabewert nicht eindeutig ist (Quelle: MSDN *GetHashCode()*-Dokumentation). Die CLI-Klassenbibliothek stellt die Klasse *ObjectIDGenerator* bereit, die eindeutige Identifikatoren für Objekte generieren kann. Der in der Hash-Tabelle gespeicherte Eintrag für eine Objekt-ID (Schlüssel) ist eine Referenz auf das Objekt des Objektgraphen bzw. auf ein neu erzeugtes Objekt im Falle einer Aktualisierung. Der Quellcodeausschnitt in Abbildung 4.4 zeigt die Implementierung der Zyklenerkennung als Erweiterung zum zuvor vergestellten Quellcodeausschnitt (Attributweise Traversierung). Wurde ein Knoten des Graphen schon untersucht, wird die gespeicherte Referenz

zurückgegeben und somit die Rekursion beendet. Die Einordnung in die Hash-Tabelle erfolgt vor den rekursiven Aufrufen der Traversierung der Objektattribute und nach dem Test auf eine neue Version. Im Falle einer Aktualisierung wird eine Referenz auf das noch uninitialisierte Objekt in die Hash-Tabelle aufgenommen, andernfalls die originale Objektreferenz.

```
1 long id = idGenerator.GetId(node,out firstTime);
2 if(!firstTime)
3 {
4     return visitedNodes[id];
5 }
6    Type newType = GetUpdate(type.GetType(),updateMapping);
7    if(newType!=null)
8    {
9        object updatedNode = FormatterServices.
            GetUninitializedObject(newType);
10       visitedNodes.Add(id,updatedNode);
11   }
12   else
13      visitedNodes.Add(id,node);
14 ...
15 // Attributweise Traversierung
16 ...
```

Abbildung 4.4: Zyklenerkennung

Nachdem alle Attribute eines Objekts untersucht wurden, wird im Falle einer Aktualisierung das neue nun mit einem gültigen Zustand initialisierte Objekt zurückgegeben. Wenn alle Attribute des Wurzelobjekts untersucht wurden, ist der Aktualisierungsalgorithmus beendet.
Bei der Traversierung des Objektgraphen müssen auch **Delegates** genauer betrachtet werden. Sie sind Funktionszeiger in der CLI, die Objektmethoden und statische Methoden als Ziel enthalten können. Auch diese müssen potentiell aktualisiert werden. Delegates sind wiederum Objekte, die unter anderem die Attribute *target* und *method* besitzen.
Ist das Ziel eines Delegates eine Objektmethode, wird eine Referenz auf das Zielobjekt im *target*-Attribut sowie die Zielmethode in Form einer *MethodInfo* im *method*-Attribut gespeichert. In diesem Fall muss die Referenz des *target*-Attributs weiter traversiert werden und im Falle einer Aktualisierung im Delegate gespeichert werden. Da in der CLI keine Manipulation der Delegate-Daten unterstützt ist, wird im Falle einer Aktualisierung ein neues Delegate mit Hilfe der Methode *Delegate.CreateDelegate* erzeugt. Ist das Ziel eines Delegates eine statische Methode, wird diese im *method*-Attribut des Delegate-Objekts gespeichert. Das *target*-Attribut ist in diesem Fall *null*. Aber auch hier muss auf eine neue Version geprüft werden, da die statische Methode aus einer aktualisierten Assembly stammen könnte. In diesem Fall wird ein neues *MethodInfo*-Objekt erzeugt und im Delegate gespeichert. Bei der Erzeugung der *MethodInfo*-Objekte ist zu beachten, dass diese passend zu den aktualisierten Typen ermittelt werden, da sich potentiell auch die Version der Parametertypen einer Methode ändern kann. Deshalb musste ein entsprechender Abbildungsmechanismus implementiert werden, der die neuen Parametertypen zuordnet und ein neues *MethodInfo*-

Objekt erzeugt. Vom C#-Compiler werden Delegates in ihrer Reinform nicht erzeugt. Andessen statt werden Multicast-Delegates generiert. Multicast-Delegates sind Container, die mehrere Delegates enthalten können. Während der Traversierungsphase werden alle enthaltenen Delegates untersucht und im Falle einer Aktualisierung zu einem neuen Multicast-Delegate zusammengefügt.
Eine weitere Herausforderung bei der Traversierung stellen **generische Typen** (*generics*) dar, die mit dem .NET Framework 2.0 eingeführt wurden und inzwischen auch in der CLI spezifiziert sind. Generische Typen sind Typen, die durch Parametrierung mit weiteren Typen spezialisiert werden (Schreibweise: $T_{generischerTyp} < T_{gebundenerTyp}, ... >$). Der entstehende Typ wird dann als gebundener generischer Typ bezeichnet. Objekte des Objektgraphen können einen gebundenen generischen Typ besitzen. Bei der Überprüfung einer neuen Version muss zunächst der ungebundene generische Typ überprüft werden, der mit Hilfe der Metadaten ermittelt werden kann. Die ungebundenen generischen Typen werden wie gewöhnliche Typen in Assemblies gespeichert und unterliegen der Versionierung. Zusätzlich muss beachtet werden, dass auch die durch die Spezialisierung gebundenen Typen selbst in einer neuen Version vorliegen können und auch selbst wieder gebundene generische Typen sein können. Die Bestimmung des neuen Typs erfolgt deshalb rekursiv. Exemplare des neuen Typs werden wie zuvor beschrieben mit Hilfe der entsprechenden Methode der *FormatterServices* erzeugt.
Einige **Optimierungen** beschleunigen die Untersuchung eines Objektgraphen. Objekte bestimmter Typen werden nicht weiter untersucht, wenn sichergestellt ist, dass keine Referenzen auf andere Objekte mehr erreicht werden können. Zu diesen Typen gehören einige Klassen aus der .NET Klassenbibliothek, wie z.B. *Reflection.RuntimePointer*.
Einige Exemplare von **Typen der Relexionsfunktionalität** der CLI können aber im Objektgraphen eingebunden sein und bedürfen spezieller Behandlung. Dazu zählen *System.Type*-Objekte, weitere Metadatentypen wie *MethodInfo*, *FieldInfo* usw., aber auch Objekte vom Typ *System.Reflection.Assembly*, welche unter anderem durch das dynamische Laden von Assemblies erzeugt werden können. All diese Metadatentypen können Typen beschreiben, welche in aktualisierten Assemblies definiert wurden. Sie werden nach einem speziellen Verfahren und nicht durch Traversieren des Objektgraphen behandelt. Für *System.Type*-Objekte und die Objekte der anderen Metadatentypen werden neue Exemplare durch die entsprechenden Reflexionsschnittstellen erzeugt und direkt in den aktualisierten Objektgraphen integriert. Assembly-Objekte werden durch das Laden der Assembly in der neuen Version und dem Austausch der Referenz auf das neue Assembly-Objekt behandelt. **Statische Attribute** von Typen, die nicht im Objektgraphen vertreten sind, wurden bis jetzt nicht betrachtet. Bei der Erzeugung der Typen der neuen Version werden auch Klassenattribute neu initialisiert. Um den Zustand dieser Attribute auf die neue Version zu übertragen, müssen alle Typen der ersetzten Assemblies gefunden (mit Hilfe von *Assembly.GetTypes()*) und die Klassenattribute der neuen nicht im Objektgraphen vertretenen Typen korrekt initialisiert werden.

4.2.3 Vorabgenerierung der Aspektlogik

Als ein Problem bei der Leistungsanalyse des beschriebenen Verfahrens hat sich herausgestellt, dass die Verwebung der Wurzelobjekte mit den entsprechenden Sychronisationsaspekten sehr zeitaufwendig ist, da der Aspektweber Rapier-Loom.Net komplexe Compiler-Logik enthält. Die Verwebung wurde bei Aktualisierungen der Wurzelobjekte zunächst während der Unterbrechungsphase vor der Objekterzeugung durchgeführt.

Um die Unterbrechungsphase zu verkürzen, wurde der verwendete Aspektweber vom Autor dieser Arbeit um die Möglichkeit der Vorabgenerierung von verwobener Aspektlogik erweitert. Dabei werden die von Rapier-Loom.Net erstellten Proxy-Klassen in separaten Assemblies gespeichert und können vor der Unterbrechungsphase wieder geladen werden. Die Verwebung der Aspekte ist dann nicht mehr während der Unterbrechungsphase notwendig, da die generierte Logik schon innerhalb der erstellten Caches in Rapier-Loom.Net vorliegt. Im Rahmen dieser Arbeit wurden entsprechende Werkzeuge implementiert, um komfortabel die benötigten Aspekt-Assemblies zu erzeugen. Vor allem bei der adaptiven Rekonfiguration wie in Adapt.Net, bei der zwischen bekannten Konfigurationen umgeschaltet wird, können die vorab generierten Assemblies die Unterbrechungszeit signifikant verkürzen. Aber auch für das Einspielen dynamischer Aktualisierungen zur Fehlerbehebung laufender Anwendungen kann das beschriebene Verfahren verwendet werden, indem vor der Initiierung der Unterbrechungsphase die Aspekt-Proxy-Typen erzeugt werden. Die Vorabgenerierung kann dabei auch auf einem dritten Rechner ausgeführt werden und dann zusammen mit der aktualisierten Logik z.B. als „*hot-fix*“ oder zusammen mit einer adaptiven Anwendung ausgeliefert werden.

4.2.4 Typspezifische Zustandsanpassung

Das vorgestellte Aktualisierungsverfahren setzt zunächst voraus, dass sich die Attribute der aktualisierten Typen nicht verändern. Unterstützte Änderungen umfassen das Hinzufügen, Entfernen und Verändern von Instruktionen, aber auch das Hinzufügen und Entfernen von Methoden, die Veränderungen von Methodensignaturen und auch die Implementierung neuer Schnittstellen. Änderungen am Objektlayout können in bestimmten Fällen zu Inkonsistenzen führen, weshalb in diesem Abschnitt eine Erweiterung des Aktualisierungsverfahrens vorgestellt wird.

Bei den Änderungen des Objektlayouts können drei Fälle unterschieden werden: hinzugekommene und entfernte Attribute sowie Attribute mit geändertem Typ. Jeder dieser Fälle kann Auswirkungen auf die Konsistenz der Daten einer Anwendung haben. Zunächst soll die Auswirkung der Änderungen mit dem vorgestellten Verfahren diskutiert werden.

Das **Entfernen von Attributen** kann in vielen Fällen unterstützt werden, wenn dies nicht die Gültigkeit des resultierenden Kapselzustands verletzt. Das vorgestellte Verfahren würde dann einfach das entfernte Attribut beim Zusandstransfer auslassen. **Neue Attribute** müssen bei der Aktualisierung mit einem Wert initialisiert werden, da kein Ausgangswert der vorangegangenen Version vorhanden ist. Gewährleistet die Initialisierung der neuen Attribute mit Standardwerten (*default values*) die Konsistenz, kann diese Art der Änderung vom Verfahren unterstützt werden. Die Entscheidung, wann eine Aktualisierung mit verändertem Objektlayout zu Inkonsistenzen führt, ist anwendungsspezifisch, kann aber vom Entwickler in vielen Fällen leicht entschieden werden. Die Verifikation von Aktualisierungen ist nicht Fokus dieser Arbeit und muss in zukünftigen Arbeiten untersucht werden.

Eine Möglichkeit, anwendungsspezifische Änderungen zu unterstützen, sind spezielle Initialisierungsroutinen, die im Folgenden vorgestellt werden sollen. **Attribute mit geändertem Typ** können durch eine Transformation des alten Wertes initialisiert werden. Verletzt dies die Konsistenz, muss auch in diesem Fall der Zustand typspezifisch angepasst werden.

Kann der Zustandstransfer bei Änderungen am Objektlayout nicht generisch wie beschrieben realisiert werden, muss der Zustand während der dynamischen Aktualisierung typspezifisch angepasst werden. Ein Beispiel für solche typspezifischen Anpassungen sind graphische Nutzerschnittstellen. Sollen durch eine Adaption vorgenommene Änderungen einer Anwendungskonfiguration z.B. eine eingeschränkte Funktionalität widerspiegeln, kann dies durch eine Anpassung der Elemente bzw. des Layouts der Anwendungsschnittstelle realisiert werden. Diese Möglichkeit soll im folgenden Abschnitt an Hand der dynamischen Aktualisierung von graphischen Komponenten in Form der *Windows.Forms*-Bibliothek diskutiert werden. Zunächst sollen Verfahren für die typspezifische Anpassung allgemein vorgestellt werden.

Da bei der Erzeugung von Typen in der neuen Version die Ausführung der Konstruktoren unterdrückt wird, kann es zu ungültig initialisiertem Zustand kommen. Deshalb ist es notwendig, durch die Ausführung spezieller Logik die Anpassung des Zustands während der Aktualisierung zu ermöglichen. In DSUP wurden zwei verschiedene Verfahren für die Integration dieser Logik implementiert. Zum einen können sogenannte **UpdateHandler** registriert werden, die für bestimmte spezifizierte Typen den generisch auf die neue Version übertragenen Zustand transformieren. Zum anderen kann bei der Entwicklung neuer Versionen Logik direkt in die Assemblies integriert werden, die dann beim Laden der aktualisierten Typen ausgeführt wird. Diese werden als **Aktualisierungskonstruktoren** bezeichnet.

UpdateHandler müssen die Schnittstelle *IUpdateHandler* implementieren, die in Abbildung 4.5 dargestellt ist. Die Schnittstelle enthält die Methode *TransformState*, die für jedes Objekt eines registrierten Typs im Objektgraphen einer Kapsel aufgerufen wird.

```
public interface IUpdateHandler
{
  object TransformState(object oldNode, object newNode);
}
```

Abbildung 4.5: Die Schnittstelle IUpdateHandler

Anhand des Typnamens wird nach dem attributweisen Kopieren geprüft, ob ein *UpdateHandler* ausgeführt werden muss. Diesem werden als Parameter Referenzen auf die neue und alte Version des Objekts übergeben. Innerhalb des *UpdateHandlers* kann mit Hilfe der Reflexionsschnittstellen der Zustand gelesen und verarbeitet werden. Das Ergebnis der Methode *TransformState* ist eine Referenz auf ein Objekt mit angepasstem Zustand, welches im Objektgraphen der Kapsel installiert wird. Mit Hilfe der *UpdateHandler* konnte die Anpassung von *UserControls* der *Windows.Forms*-Bibliothek realisiert werden, die in einem folgendem Abschnitt 4.2.5 vorgestellt werden soll.

Eine zweite Möglichkeit der typspezifischen Anpassung stellen die Aktualisierungs-Konstruktoren dar. Im Gegensatz zu den *UpdateHandlern*, die in DSUP registriert und in separaten Assemblies gespeichert sind, werden Aktualisierungskonstruktoren in den Assemblies definiert, die auch die aktualisierten Typen enthalten. Bei der Implementierung neuer Versionen kann der Entwickler die Zustandsanpassung innerhalb spezieller Methoden realisieren, die über CLI-Attribute für den Aktualisierungsalgorithmus annotiert werden. Die Methoden müssen dieselbe Signatur wie die *TransformState*-Methode der *UpdateHandler* besitzen und bekommen auch als Parameter eine Referenz auf neue und alte Version des entsprechenden Objekts. Der Rückgabewert wird in den

Objektgraphen der Kapsel installiert. Eine Besonderheit bei dieser Variante der typspezifischen Anpassung ist die Möglichkeit zwischen verschiedenen Versionsübergängen unterscheiden zu können. Über das CLI-Attribut *[UpdateConstructor(string fromVersion)]*, mit dem die entsprechenden Methoden annotiert sein müssen, kann über einen Parameter eine Versionsinformation spezifiziert werden, der festlegt, bei welcher Version des Ausgangsobjekts die Methode während der Aktualisierung aufgerufen wird. Die Angabe der Versionsinformation erfolgt in Form einer Zeichenkette als Assembly-Versionsnummer. Ein Beispiel für einen Aktualisierungskonstruktor ist in Abbildung 4.6 dargestellt.

```
[UpdateConstructor("3.334.34.3")]
object UpdateConstructor_3D_to_2D(object oldNode, object
   newNode)
{
  int depth = (int) oldNode.GetType().GetField("depth");
  newNode.GetType().GetField("height").SetValue(newNode,Math.
     Sin(depth)*Math.Sqrt(2));
  return newNode;
}
```

Abbildung 4.6: Beispiel eines versionspezifischen Aktualisierungskonstruktors

Während der dynamischen Aktualisierung wird bei der Erzeugung aktualisierter Typen die Existenz der vorgestellten CLI-Attribute geprüft und entsprechend der Version der Assembly des alten Objekts die annotierte Methode per Reflexion aufgerufen.

4.2.5 Anpassung graphischer Nutzerschnittstellen

Die Verwendung der vorgestellten *UpdateHandler* soll an Hand eines komplexeren Beispiels, der Anpassung graphischer Nutzerschnittstellen, demonstriert werden. Diese werden notwendig, wenn Änderungen der Umgebung die Adaption der Anwendung erfordern und die Änderungen an der Anwendungskonfiguration dem Nutzer durch angepasste Interaktionselemente visualisiert werden sollen.

Aktualisierung von Windows.Forms-Anwendungen

Graphische, interaktive Anwendungen werden in der .NET-Plattform mit Hilfe der *Windows.Forms*-Bibliothek implementiert, die das Ereignissystem (*events*) der CLI verwendet. Die *Windows.Forms*-Bibliothek wurde nicht im Rahmen der CLI standardisiert und ist deshalb spezifisch für Microsofts Implementierung - die .NET-Plattform. Bausteine der Oberfläche werden als *Controls* bezeichnet. Für bestimmte Ereignisse können Behandlungsroutinen registriert werden, die dann entsprechende Logik als Reaktion auf Aktionen eines Nutzers ausführen. Soll in die Oberfläche durch eine Aktualisierung ein neues *Control* integriert werden oder die Position bzw. Größe eines *Controls* geändert werden, ist dies mit dem generischen Aktualisierungsverfahren nicht möglich. Darstellungsinformationen und enthaltene *Controls* werden als Zustand eines jeden *Controls* gespeichert. Bei der Aktualisierung auf eine Version mit angepassten *Controls* wird zwar existierende Logik aktualisiert, das neue Aussehen aber auf Grund des unterdrückten Konstuktorlaufs, in dem die *Controls* normalerweise initialisiert werden, nicht aktiviert. Hierfür müssen *Control*-spezifische Änderungen bei

der Aktualisierung vorgenommen werden, die mit Hilfe der *UpdateHandler* realisiert wurden.
Im DSUP-Projekt wurden *UpdateHandler* für *Forms* und *UserControls* implementiert. Die beiden von *System.Windows.Forms.Control* abgeleiteten Klassen stellen die wesentlichen Bausteine für graphische Benutzerschnittstellen dar. Eine *Form* ist ein Container für *Controls* die wiederum *Controls* enthalten können. *UserControls* sind spezielle, nutzerdefinierte *Controls*. Die implementierten *UpdateHandler* erzeugen beim Aufruf der *TransformState*-Methode zunächst ein Exemplar des neuen Typs, inklusive der Ausführung des parameterlosen *Default*-Konstruktors. Dabei wird ein Referenz-*Control* erzeugt, welches das neue Layout und die Darstellungsinformationen der Nutzerschnittstelle enthält. Die *UpdateHandler* iterieren über alle vorhandenen Sub-*Controls* und führen entsprechende Anpassungen durch, wobei selektiv der Zustand vom *default*-initialisierten *Control* zur neuen Version übertragen wird. Dazu gehören die Position (*location*), die Größe (*size*), Hinter-/ Vordergrundfarbe, die Schriftart (*Font*) und weitere. Zusätzlich werden in der *default*-initialisierten Version nicht vorhandene Sub-*Controls* entfernt und neue hinzugefügt, indem die Liste der Sub-*Controls* entsprechend verglichen wird. Nach der dynamischen Aktualisierung wird damit die neue Darstellung der *Controls* der Anwendung sichtbar.

Hinzufügen und Entfernen von Ereignisbehandlungsroutinen

Ereignisbehandlungsroutinen der *Controls* werden durch den Aktualisierungsalgorithmus auf die Methoden der neuen Version abgebildet. Wurden allerdings neue Ereignisbehandlungsroutinen registriert bzw. alte entfernt müssen diese Änderungen innerhalb des *UpdateHandlers* erkannt und in der neuen Version installiert werden. Für die Behandlung dieser Änderungen war eine komplexe Analyse der *Windows.Forms*-Bibliothek notwendig.
Abbildung 4.7 zeigt einen Ausschnitt des Objektgraphen einer *Windows.Forms*-Anwendung. Auf der linken Seite ist die dynamisch aktualisierbare Kapsel (ein *UserControl*) dargestellt. Neben den funktionalen Methoden existieren die durch den dynamischen Weber erzeugten Proxy-Methoden. Ein *UserControl* hat als Attribute eine Reihe weiterer *Controls*. Dabei kann es sich um einfache *Controls* wie Knöpfe (*buttons*) und Beschriftungen (*labels*), aber auch um komplexe Tabellen und Listen handeln. In Abbildung 4.7 ist ein *Button*-Objekt dargestellt. Jedem *Control* ist ein natives Fenster (*NativeWindows*) des Windows-Subsystems zugeordnet, welches durch ein *Windows*-Handle repräsentiert wird. Zusätzlich sind jedem *Control* eine Reihe von Ereignissen (hier *ClickEvent* und *KeyPressEvent*) zugeordnet, die wiederum eine Liste von Delegates enthalten, die unter anderem auf Methoden des aktualisierten *UserControls* verweisen können.
Beim Erzeugen eines *Controls* wird neben der Erstellung eines nativen Fensters des *Windows*-Subsystems auch ein *Callback* registriert, welches im Falle des Auftretens von Ereignissen für das native Fenster aufgerufen wird. Die Methode *WndProc* ist für die Zuordnung von Ereignissen des nativen Windows-Subsystems an die CLI-Ereignisobjekte zuständig. Auf der rechten Seite von Abbildung 4.7 ist der Kontrollfluss von der Nachrichtenschleife der Anwendung bis zu den einzelnen Ereignisobjekten dargestellt. Beim Auftreten eines Ereignisses werden alle registrierten Ereignisbehandlungsroutinen innerhalb der Anwendung ausgeführt.
Um das Hinzufügen und Entfernen von Ereignisbehandlungsroutinen zu realisieren, müssen die Listen von Ereignisbehandlungsroutinen der *Control*-Ereignisse angepasst

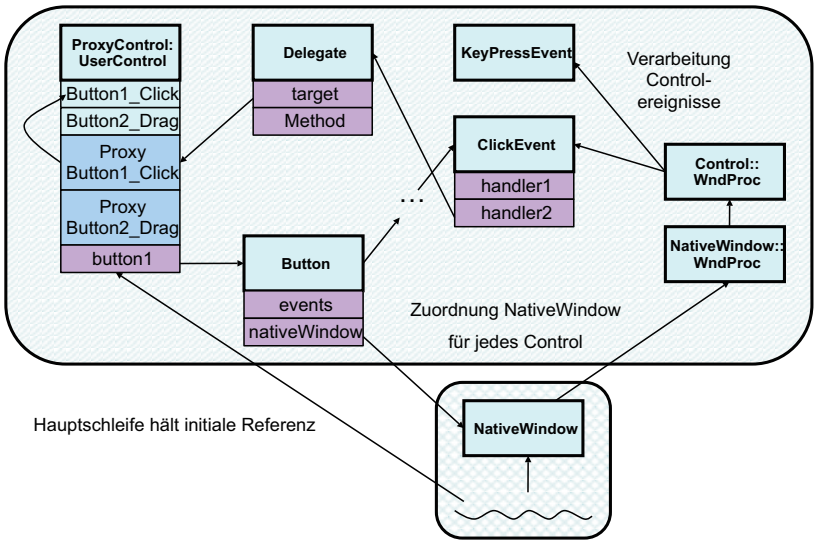


Abbildung 4.7: Dynamische Aktualisierung von *Windows.Forms.Controls*

werden. Da die .NET-Bibliotheken keinen direkten Zugriff auf diese verkettete Liste ermöglichen, kam an dieser Stelle Reflexion zum Einsatz. Dabei wurde die verkettete Liste der alten Version eingelesen und Änderungen entsprechend auf das Exemplar der neuen Version übertragen.

Verzögerte Aktualisierung

Eine Herausforderung bei der Aktualisierung von *Controls* sind Thread-übergreifende Aufrufe von *Control*-Methoden, die bei der Implementierung der *UpdateHandler* benötigt werden. Es ist nicht ohne Weiteres möglich, Daten eines *Controls* von einem anderen Thread aus zu manipulieren als von dem, mit dem das *Control* erzeugt wurde, da das *Windows*-Subsystem bestimmte Daten Thread-spezifisch speichert. Da der Aktualisierungsalgorithmus in einem anderen Thread ausgeführt wird als die Anwendungslogik, wurde die Möglichkeit der verzögerten Aktualisierung entwickelt. Hierbei wird die Aktualisierung verzögert ausgeführt, indem beim Aufruf der Methode *DeferredUpdate* im Aktualisierungsmanager zunächst nur die aktualisierungsspezifischen Daten gespeichert werden. Erst beim nächsten Aufruf einer Methode des *Controls* wird die Aktualisierung dann im Kontext des Anwendungs-Threads, von dem das *Control* erzeugt wurde, ausgeführt. Dies wird innerhalb der Aspektlogik realisiert, indem eine boole'sche Variable vor jedem Methodenaufruf überprüft wird und die entsprechend vom Aktualisierungsmanager gesetzt wird.

Mit dem vorgestellten *UpdateHandler* für die Anpassung graphischer Nutzerschnittstellen in der .NET-Plattform wurde die Arbeitsweise der typspezifischen Zustandsanpassung demonstriert. Mit Hilfe dieser wiederverwendbaren Logik ist es möglich, graphische Bestandteile einer Anwendung durch die Änderung der Darstellung und damit des Layouts, das Hinzufügen, Entfernen und Ändern von Ereignisbehandlungsroutinen sowie der Aktualisierung der Struktur der verwendeten *Controls* zu adaptieren.

4.2.6 Dynamische Aspekte

Eine wichtige Möglichkeit, alternative Varianten von Softwarekomponenten zu erstellen, ist die Nutzung dynamischer Aspekte. Mit Hilfe dynamischer Aspekte können Aspekte an den Objekten einer Kapsel aktiviert bzw. deaktiviert werden und auch die Implementierung der Aspektlogik selbst angepasst werden. Mit Hilfe der Aspekte kann das Verhalten und auch die nichtfunktionalen Eigenschaften einer Kapsel und damit der Anwendung durch das Abfangen und die Manipulation von Parametern bei Methodenaufrufen innerhalb und außerhalb der Kapsel adaptiert werden. Adaptivität durch das dynamische Hinzuschalten von Aspekten zu realisieren wurde zuerst von [Hirschfeld und Kawamura 2006] beschrieben. Dieser Arbeit liegt ein neues Programmiermodell zu Grunde, bei dem die Aspekte sehr leicht durch die Nutzung von Annotationen direkt im Anwendungscode verwaltet werden können.
In Rapier-Loom.Net werden Aspekte mit Hilfe von CLI-Attributen bei der Typdeklaration konfiguriert. Die Annotation der Metadaten wird bei der Erzeugung neuer Exemplare durch den Aspektweber überprüft. Um die dynamische Aspektaktivierung in Rapier-Loom.Net verwenden zu können, musste die Fabrik-Methode *Weaver.CreateInstance* für die Erzeugung der aktualisierten Typen (anstatt *new*) verwendet werden. Wird während des zuvor beschriebenen Traversierungsprozesses im Objektgraphen ein zu aktualisierendes Objekt gefunden, wird es mit Hilfe der Methode *Weaver.CreateInstance* in der neuen Version erzeugt, wobei auch evtl. hinzugefügte bzw. entfernte Aspektannotierungen und aktualisierte Aspektimplementierungen berücksichtigt werden. Rapier-Loom.Net implementiert die Verwebung der Aspekte und auch die Objekterzeugung gemeinsam in der Methode *Weaver.CreateInstance*. Der Aktualisierungsalgorithmus erfordert aber die Erzeugung von uninitialisierten Objekten. Um Verwebung und Exemplarerzeugung zu trennen, wurde ein *UninitializedObject*-Aspekt implementiert, welcher in Abbildung 4.8 dargestellt ist.

```
public class UninitializedObjectAspect:Aspect{
    [Create(Invoke.Instead)]
    public object CreateUninitializedObject(Type t,object[]
       args{
      return FormatterServices.GetUninitializedObject(t);}
}
```

Abbildung 4.8: Trennung von Verwebung und Objekterzeugung

Das CLI-Attribut *[Create(Invoke.Instead)]* und die Signatur der Methode *CreateUninitializedObject* veranlassen den Aspektweber, die Objekterzeugung an diese Methode zu delegieren. Dabei wird der verwobene Typ als Parameter übergeben. Dieser wird benutzt, um ein uninitialisiertes Objekt von diesem Typen mit Hilfe der Methode *FormatterServices.GetUninitializedObject* zu erzeugen, der wiederum als Rückgabewert der Fabrikmethode an den Aktualisierungsmanager übergeben wird.
Beim Traversieren des Objektgraphen wird jedes Objekt hinsichtlich verwobener Aspekte überprüft. Wird eine Aktualisierung des Objekts selbst oder der Aspektimplementierung erkannt, wird die neue Version mit Hilfe der Fabrikmethode des Aspektwebers erzeugt und der Aspektzustand der alten Version auf die neue übertragen. Aspekte werden in Rapier-Loom.Net durch Objekte repräsentiert. Eine Referenz auf die Aspektobjekte wird im Attribut *_aspect_* für jeden Aspekt auf einer bestimmten Ver-

erbungstiefe des verwobenen Typs gespeichert. Das Aspektobjekt selbst wird wieder als Teilgraph der Kapsel über die rekursive Methode *UpdateObjectGraph* traversiert und entsprechend aktualisiert. Neue Aspekte werden über den Standardkonstruktor erzeugt und in das neue Objekt übertragen. Hierbei muss beachtet werden, dass durch das Hinzufügen und Entfernen von Aspekten die Vererbungsstruktur der generierten Typen verändert wird. Bei der Übertragung der Aspektobjekte auf eine aktualisierte Version müssen diese Änderungen bei der Zuweisung der *_aspect_*-Attribute entsprechend zugeordnet werden.

4.3 Einschränkungen

Das entwickelte Verfahren schränkt mögliche Anwendungen durch wenige Bedingungen ein, die für die meisten Anwendungen unproblematisch sind, aber der Vollständigkeit halber diskutiert werden sollen. Der Aufruf nativer Methodenaufrufe (*P/Invoke*) darf keine Referenzen auf Objekte der Anwendung beinhalten, da diese nach einer Aktualisierung ungültig sein können. Aus demselben Grund darf kein *unsafe code* verwendet werden. Auf Grund des eingesetzten Aspektwebers dürfen die Typen von Wurzelobjekten nicht *sealed* oder *internal* sein. Änderungen der Schnittstelle der Wurzelobjekte werden nicht unterstützt. Auf die CLI-*Property System.Runtime.Assembly.Location* darf nicht zugegriffen werden, da diese durch das eingesetzte Ladeverfahren für Assemblies verändert wird.

4.4 Zusammenfassung

Im Rahmen dieses Kapitels wurde ein neues Verfahren zur dynamischen Aktualisierung von Kapseln vorgestellt, welches portabel ohne plattformspezifische Manipulationen der virtuellen Maschine in moderne Komponentenplattformen integriert werden kann. Durch den Einsatz des zuvor vorgestellten Rekonfigurationsalgorithmus kann ein rekonfigurierbarer Zustand der Kapseln herbeigeführt werden, in dem alle Objekte einer Kapsel über die Wurzelobjekte erreichbar sind, weil keine Referenzen auf den Stacks der Threads existieren, auf die in den eingesetzten Laufzeitumgebungen nicht effizient zugegriffen werden kann. Die dynamische Aktualisierung wurde durch die Traversierung über alle Attribute der Objekte einer Kapsel implementiert, wobei der Typ eines jeden Objekts auf eine Aktualisierung überprüft wird und ggf. ein neues Objekt erzeugt werden kann. Der Zustand des alten Objekts wird durch eine attributweise Kopie auf das neue Exemplar übertragen.
Besonders hervorzuheben ist, dass durch die Umsetzung auf Basis der CLI ein sehr komplexes Verfahren implementiert wurde, bei dem eine ausführliche Kenntnis der Laufzeitumgebung nötig war. Zu den Besonderheiten der Implementierung zählen die Erkennung von Zyklen im Objektgraphen, die Analyse von Feldern, Delegates und generischen Typen, die Erzeugung neuer Objekte ohne Konstruktorlauf und auch die Behandlung von Typen der Reflexionsfunktionalität. Das entwickelte Verfahren ist leicht auf die Java-Plattform übertragbar. Die Umsetzung ist weniger komplex, da in der Java-Plattform keine Delegates existieren.
Das entwickelte Verfahren unterstützt beliebige Änderungen der Instruktionen von Methoden, das Hinzufügen und Entfernen von Methoden und die Implementierung neuer Schnittstellen innerhalb einer Kapsel. Für Änderungen des Objektlayouts wurden zwei Verfahren vorgestellt, bei denen typspezifische Transformatoren die Konsis-

tenz des Anwendungzustands gewährleisten können. Das entwickelte Verfahren wurde bei der komplexen Aktualisierung von graphischen Nutzerschnittstellen eingesetzt, bei der adaptive Änderungen der Anwendungskonfiguration in den graphischen Elementen der Nutzerschnittstelle visualisiert werden können. Mit Hilfe dieser Technik können z.B. verschiedene in der Entwicklungsumgebung Visual Studio entwickelte Varianten einer Nutzerschnittstelle während der Laufzeit aktiviert werden, ohne die Anwendung neu starten zu müssen. Das entwickelte Aktualisierungsverfahren konnte mit wenigen Änderungen in die Adapt.Net-Ausführungsumgebung integriert werden und ermöglicht so die Erstellung alternativer Anwendungskonfigurationen durch den Einsatz alternativer Algorithmen bei der Implementierung angepasster Versionen einer Softwarekomponente. Aber auch entkoppelt konnte das Verfahren im DSUP-Projekt für die Aktualisierung verteilter Anwendungen eingesetzt werden. Ein wichtiger Anwendungsfall ist das Einspielen neuer Softwareversionen, um auf bekannt gewordene Sicherheitslücken oder andere Fehler einer Software zu reagieren. Durch die Möglichkeit der dynamischen Aktualisierung ohne einen Neustart der Anwendung kann die Verfügbarkeit erhöht werden.

5 Profile, Werkzeuge und Muster

In diesem Kapitel soll demonstriert werden, wie mit Hilfe der vorgestellten Ausführungsplattform adaptive Anwendungen realisiert werden können. Adaptive Anwendungen sollen in der Lage sein, bei Änderungen der Umgebung Anpassungen vorzunehmen, um bestimmte nichtfunktionale Eigenschaften innerhalb eines durch den Anwender definierten Bereiches zu halten . Die nichtfunktionalen Eigenschaften werden neben den Umgebungseigenschaften wesentlich von der Anwendungskonfiguration bestimmt. Durch die Auswahl und die Aktivierung einer geeigneten Anwendungskonfiguration bei Veränderungen der Umgebung können die nichtfunktionalen Eigenschaften im vorgegebenen Bereich gehalten werden. Mit der dynamischen Rekonfiguration wurde ein wichtiger Ausgangspunkt für die Adaption geschaffen.
Ein weiterer Bestandteil adaptiver Anwendungen ist die wiederholte Beobachtung von Parametern der Umgebung und die Auswahl entsprechender Anwendungskonfigurationen. Im Rahmen dieser Arbeit wurde eine Technik entwickelt, bei der die Auswahl einer Anwendungskonfiguration über ein Profil erfolgt, welches gemessenen Umgebungseigenschaften und dem Anwendungszustand eine entsprechende Konfiguration zuordnet. Hierfür wurde eine neue Komponente, ein Adaptationsmanager, in die Ausführungsplattform Adapt.Net integriert.
Im Rahmen dieses Kapitels sollen weitere wichtige Aspekte bei der Entwicklung adaptiver Anwendungen diskutiert werden. Vor allem die Entwicklung alternativer Anwendungskonfigurationen steht dabei im Vordergrund. Ein wichtiger Bestandteil des Adapt.Net-Projekts ist ein Werkzeug für die graphische Erstellung von Anwendungskonfigurationen sowie der Generierung von Installationspaketen auslieferbarer adaptiver Anwendungen. Mögliche Architekturmuster, die zur Erstellung alternativer Anwendungskonfigurationen verwendet werden können, werden zum Abschluss des Kapitels beschrieben.

5.1 Auswahl geeigneter Anwendungskonfigurationen

In diesem Abschnitt soll die Zuordnung von Kontext und Anwendungskonfigurationen diskutiert werden, die letztendlich die Adaptivität von Anwendungen beschreibt. An dieser Stelle muss bemerkt werden, dass die Auswahl und Komposition geeigneter Anwendungskonfigurationen nicht im Mittelpunkt dieser Arbeit stand. Für die Realisierung des Gesamtansatzes zur Entwicklung adaptiver Anwendungen sollen diese Aspekte aber in das entworfene Modell aufgenommen und die profilbasierte Konfigurationszuordnung formalisiert werden.

5.1.1 Dienstgüte und Ressourcen

Nichtfunktionale Eigenschaften, oft auch synonym mit dem Begriff der Dienstgüte (*quality of service*)(QoS) betitelt [Zinky und Bakken 1995], lassen sich in einem verteilten System in verschiedenen Facetten finden. Im weitesten Sinne beschäftigt sich

QoS mit der Vielzahl von Parametern, abseits des anwendungsspezifischen funktionalen Verhaltens. Elemente nichtfunktionaler Eigenschaften finden sich auf der Ebene unterliegender Netzwerke, der Middleware, aber auch auf Betriebssystem- und der Anwendungsebene selbst. Beispiele für nichtfunktionale Eigenschaften umfassen Leistungscharakteristika, Sicherheit oder die Verfügbarkeit. Dienstgüteeigenschaften auf Anwendungsebene werden in der Literatur auch oft als *application-level* QoS [Woodside und Menascé 2006] bezeichnet.

Während in statischen Systemen die Sicherstellung von bestimmten QoS-Parametern auf Anwendungsebene oft durch geeignete Ressourcenreservierungsstrategien (RSVP [Braden 1997]) durchgesetzt werden kann, ist die Verwaltung von QoS-Parametern auf Anwendungsebene in dynamischen Infrastrukturen von zahlreichen Parametern abhängig. Die Komposition von Software aus alternativen Softwarekomponenten bietet eine hohe Flexibilität bzgl. der nichtfunktionalen Eigenschaften einer Anwendung. Die nichtfunktionalen Eigenschaften berechnen sich während der Laufzeit im Wesentlichen aus den Eigenschaften ihrer Teile - den Kapseln - sowie aus den der Anwendung zur Verfügung stehenden Ressourcen, die wiederum QoS-Parameter der unterliegenden Infrastruktur sind. Durch die Auswahl geeigneter Softwarekomponenten können die resultierenden Eigenschaften des Gesamtsystems bestimmt werden. Die Berechnung nichtfunktionaler Eigenschaften hat sich als komplexes Problem bewiesen [Rajkumar u. a. 1997]. So hängen die Eigenschaften der einzelnen Kapseln von einer Vielzahl von Parametern wie ihrer Implementierung, der Vergangenheit der Eingabedaten oder den potentiell schwankenden Ressourcen ab.

Ziel bei der Entwicklung adaptiver Software ist die Optimierung nichtfunktionaler Eigenschaften auf Anwendungsebene. Die funktionalen Anforderungen einer Anwendung können oft mit einer Vielzahl möglicher Konfigurationen realisiert werden. Jede Konfiguration kann bei der Ausführung in Abhängigkeit von ihrem Kontext unterschiedliche nichtfunktionale Eigenschaften vorweisen. Die Auswahl geeigneter Konfigurationen lässt sich im Wesentlichen durch zwei Ansätze realisieren. Der erste Ansatz, die Berechnung der Anwendungseigenschaften aus den verfügbaren Ressourcen sowie den Kapseleigenschaften, erweist sich auf Grund der Vielzahl von Parametern als sehr komplex. Im zweiten Ansatz werden nichtfunktionale Anwendungseigenschaften durch Messung bestimmt. Für typische Einsatzszenarien werden Eigenschaften von Testkonfigurationen bewertet und eine optimale Konfiguration für bestimmte Einsatzsituationen (Kontext) bestimmt. Die dynamische Rekonfiguration ermöglicht dabei die Umschaltung der aktiven Anwendungskonfigurationen während der Laufzeit und realisiert damit das adaptive Verhalten.

5.1.2 Dienstgüte und Ressourcen im Modell adaptiver Anwendungen

Jede Anwendung wird in einer Umgebung ausgeführt, die der Anwendung **Ressourcen** (R) zur Verfügung stellt. Diese Ressourcen werden von einer Anwendung konsumiert. Werden mehrere Anwendungen auf einem System ausgeführt, konkurrieren die Anwendungen um die verfügbaren Ressourcen. Ressourcen sind durch Vektoren skalarer Werte beschreibbar und beeinflussen die Eigenschaften einer Anwendung. Beispiele für Ressourcen sind eine Netzwerkverbindung mit einer gegebenen Bandbreite, Übertragungsverzögerung und *Jitter* oder eine Batterie in einem mobilen Gerät.

Zusätzlich zu den Ressourcen können weitere Parameter die Umgebung einer Anwendung bestimmen. Beispiele sind die Position eines Nutzers oder der Lebenszykluszu-

stand einer Kapsel (z.B. Ausfall - siehe Kapitel 6). Als **Umgebungsparameter** (U) werden Eigenschaften der Umgebung bezeichnet, die keine Ressourcen darstellen.
Jede Anwendung hat während ihrer Ausführungszeit beobachtbare Eigenschaften, die als **Dienstgüteeigenschaften auf Anwendungsebene** (Q) bezeichnet werden. Die Eigenschaften sind durch Messung bestimmbar und besitzen einen skalaren Wertebereich. Beispiele für nichtfunktionale Eigenschaften sind Ende-zu-Ende-Antwortzeiten oder die Bildwiederholrate einer Videoabspielsoftware.

5.1.3 Adaption im Modell

Die sichtbaren nichtfunktionalen Eigenschaften einer Anwendung und auch das funktionale Verhalten selbst werden durch eine Reihe von Faktoren bestimmt. Neben der Anwendungskonfiguration gehören zu diesen Faktoren die verfügbaren Ressourcen und Umgebungsparameter sowie potentiell die Historie aller Eingaben. Durch den Nutzer sind eine Reihe von Vorgaben an funktionale und nichtfunktionale Eigenschaften gegeben, die im Softwareentwicklungsprozess spezifiziert werden. Die Ausführung in statischen Umgebungen ermöglicht den Entwurf und die Entwicklung der Software entsprechend den statisch gegebenen Ressourcenvorgaben. In dynamischen Umgebungen ändern sich diese und die Anpassung während der Laufzeit ist nötig. Zunächst sollen die vom Nutzer gegebenen Anforderungen an die Anwendungseigenschaften durch ein Akzeptanzkriterium formalisiert werden.

Definition 5.1.1 (Akzeptanzkriterium) *Sei Q die Menge aller beobachtbaren Eigenschaften einer Anwendung, dann ist die Abbildung AKZ: Q → (wahr, falsch) definiert durch:*

$$AKZ(Q_x) = \begin{cases} wahr & Q_x \in Q \text{ ist in einem akzeptablem Bereich} \\ falsch & \text{sonst.} \end{cases} \tag{5.1}$$

Eine Anwendung funktioniert entsprechend ihrer Spezifikation, wenn das Akzeptanzkriterium zu jeder Zeit der Ausführung den Wert „wahr“ besitzt. Das zentrale Ziel der Adaption ist es, auf Änderungen der Umgebung und damit direkt auf Änderungen der verfügbaren Ressourcen und der Umgebungsparameter zu reagieren, um das vom Anwender vorgegebene Akzeptanzkriterium zu erfüllen. Die Erfüllung des Akzeptanzkriteriums wird bestimmt durch die vom unterliegenden System gegebenen verfügbaren Ressourcen und der veränderlichen Anwendungskonfiguration. Durch eine gültige Belegung der freien Variablen der Anwendungskonfiguration kann das Akzeptanzkriterium erfüllt werden. Dieser Zusammenhang zwischen den Anwendungseigenschaften Q und einer Anwendungskonfiguration sowie den verfügbaren Ressourcen bildet das „Paradigma der Adaptivität“.

Theorem 5.1.1 (Paradigma der Adaptivität) *Sei $\mathcal{E}$ die Menge aller Eingaben, Q eine Menge nichtfunktionaler Anwendungseigenschaften und U eine Menge von Umgebungsparametern. Für eine Menge verfügbarer Ressourcen R wähle KONF, so dass gilt:*

$$\forall Q_x \in Q : AKZ(Q_x) = \text{wahr mit } Q = \text{f(R,U,KONF,E)}. \tag{5.2}$$

Ausgehend vom „Paradigma der Adaptivität“ lässt sich das **Konfigurationsproblem** formulieren: Finde eine Konfiguration KONF, so dass sich alle nichtfunktionalen Eigenschaften im spezifizierten Bereich befinden. Allgemein ist diese Problematik auf

Grund des Halteproblems der Turing-Maschine nicht lösbar. Für die Lösung muss eine Berechnungsvorschrift für die nichtfunktionalen Eigenschaften (Q) aus den verfügbaren Ressourcen, den Eingaben der Anwendung und der Konfiguration gefunden werden. Eine Konfiguration umfasst dabei wieder Kapseln mit Eigenschaften, Parametern und Konnektoren, die wiederum Einfluss auf die Berechnung finden müssen. Die Eigenschaften von Kapseln hängen nun wiederum von der enthaltenen Logik ab, was zum Halteproblem der Turing-Maschine führt, welches nicht entscheidbar ist.

Vereinfachungen erlauben es dennoch, vielversprechende Lösungen für das Konfigurationsproblem zu finden. Setzt man die Unabhängigkeit der Eigenschaften von Kapseln von deren aktuellem Zustand, also der Historie der Eingabedaten voraus, lassen sich die nichtfunktionalen Eigenschaften berechnen, vorausgesetzt der Zusammenhang zwischen verfügbaren Ressourcen, Kapselparametern und Kapseleigenschaften ist beschreibbar. Ein Beispiel ist die Berechnung des Speicherverbrauchs einer Anwendung. Summiert man unter den gegebenen Voraussetzungen den Speicherverbrauch einzelner Kapseln, so kann der Gesamtspeicherverbrauch ermittelt werden. Beim Konfigurationsproblem sind zwei Teilbereiche zu unterscheiden. Zunächst besteht das Problem der Auswahl geeigneter Softwarekomponenten aus einem Komponentenkatalog. Diese Problematik wurde unter anderem im Forschungsbereich des „Ressourcenorientieren Konfigurierens“ [Heinrich 1993], einem Teilgebiet der Künstlichen Intelligenz, untersucht. Zusätzlich müssen Werte für die Parameter der Kapseln gefunden werden, die wiederum die Eigenschaften der Anwendung beeinflussen.

Das Finden geeigneter Kapselparameter (Parametrierung) sowie einer Anwendungskonfiguration ist ein NP-vollständiges Problem. [Rajkumar u. a. 1997] beweist dies durch Abbildung auf ein nachgewiesen NP-vollständiges Problem (*0-1 Knapsack*). [Rajkumar u. a. 1997] führt nutzerdefinierte *Utility*-Funktionen ein, die einen Zufriedenheitswert des Nutzers mit einer jeweiligen nichtfunktionalen Eigenschaft festlegen. Um die optimale Zufriedenheit eines Nutzers für eine Anwendungskonfiguration mit entsprechenden Kaspelparametern zu finden, können diese zunächst diskretisiert werden. Die Verfügbarkeit von Ressourcen schränkt den Suchraum ein, da die Wahl der Kapselparameter den Ressourcenbedarf der Anwendungen definiert. Bei der Abbildung auf *0-1 Knapsack* ist der Gesamtressourcenbedarf äquivalent zur Rucksackkapazität, der Profit gleich der Nutzerzufriedenheit sowie die Kapselparameter äquivalent zum Gewicht der Gegenstände. Entsprechende Softwareprozess- und Werkzeugunterstützung zur teilautomatierten Lösung des Konfigurationsproblems wurde im MADAM-Projekt [Alia u. a. 2006; Geihs u. a. 2006] entwickelt. In MADAM können u.a. mit Hilfe einer UML-Erweiterung Varianten einer Anwendung beschrieben werden, aus denen während der Laufzeit mit Hilfe von *Utility*-Funktionen ein optimales Exemplar ausgewählt wird.

Ein wichtiger Punkt für die Lösbarkeit des Konfigurationsproblems der Parametrierung ist die Diskretisierung der möglichen Anwendungskonfigurationen, da die Kardinalität der Parametermenge den Suchaufwand mit quadratischer Komplexität erhöht. [Rajkumar u. a. 1997] beschreibt einen Approximationsalgorithmus zur Bestimmung einer nahezu optimalen Lösung (*near optimal solution*) mit der Komplexität O(n log n). Bei gegebenem Suchraum lassen sich durch den Algorithmus akzeptable obere Schranken für die Berechnung kleiner Konfigurationsprobleme finden, die auch während der Laufzeit berechnet werden können.

Die Bestimmung der optimalen Konfiguration insgesamt ist ein lineares mehrdimensionales Optimierungsproblem, welches während der Laufzeit der Anwendungen nur

schwer lösbar zu sein scheint. Aus diesem Grund wurde in dieser Arbeit ein anderer Ansatz für die Bestimmung optimaler Anwendungskonfigurationen verwendet, obwohl sich existierende Lösungen des Konfigurationsproblems auf die im Rahmen dieser Arbeit entwickelte Laufzeitinfrastruktur übertragen ließen. Durch die Möglichkeit des Ladens neuer Konfigurationen während der Laufzeit, kann eine zusätzliche Planungskomponente leicht in die vorgestellte Adapt.Net-Infrastuktur integriert werden.

5.2 Profilbasierte Konfigurationszuordnung

Wie schon am Beginn dieser Arbeit motiviert, müssen Anwendungen in dynamischen Umgebungen operieren. Die dynamischen Eigenschaften der Umgebungen, die in den verfügbaren Ressourcen resultieren, wechseln dabei ständig zwischen wiederkehrenden Situationen. Oft sind zuvor mögliche Umgebungsszenarien bekannt, für die optimale Konfigurationen durch den Anwendungsentwickler bestimmt werden können. Ein Beispiel hierfür ist der Wechsel zwischen unterschiedliche Netzwerkklassen (WLAN, LAN, GSM), die wiederkehrend gleiche Ressourcen bereitstellen. Der Entwickler kann verschiedene Situationen simulieren - seine Software in einer bestimmten Umgebung testen - und dabei die gewünschten Eigenschaften der Anwendung durch die Wahl einer geeigneten Anwendungskonfiguration garantieren. Während der Laufzeit muss die entsprechende Situation, z.B. durch Messung, ermittelt und eine geeignete Konfiguration geladen werden. Als Hilfe bei der Erstellung angepasster Konfigurationen können Architekturmuster dienen, von denen einige in diesem Kapitel vorgestellt werden.

Die Abbildung von der ermittelten Umgebungssituation und der geeigneten Konfiguration erfolgt in Form eines Profils. Ein **Profil** ist eine Abbildung von (R,Q,U) → (K,C,P,M) und dient als Eingabe für den Adaptionsmanager, um die Anwendung an die Umgebungseigenschaften anzupassen, und bildet die Grundlage für die Adaptation einer Anwendung. Die Umgebungssituation kann durch Messung der verfügbaren Ressourcen und der Auswertung der Umgebungsparameter bestimmt werden und damit eine Konfiguration mit parametrierten Kapseln, Konnektoren und einer Rechnerzuordnung aktiviert werden. In den Definitionsbereich eines Profils wurden zusätzlich die nichtfunktionalen Eigenschaften der Anwendung aufgenommen. Dies ermöglicht dem Programmierer zusätzlich zu Änderungen der verfügbaren Ressourcen auch auf die Abweichung von nichtfunktionalen Eigenschaften der Anwendung zu reagieren. Ein Beispiel ist das Unterschreiten einer Schranke für eine Ende-zu-Ende-Antwortzeit unter ein bestimmtes Zeitlimit, worauf eine neue Konfiguration geladen werden kann.

Das Konzept der profilbasierten Konfigurationszuordnung stellt mit den vorgestellten Rekonfigurationsalgorithmen eine effiziente Möglichkeit dar, Anwendungen an verschiedene Situationen der Umgebung anzupassen. Auch in diesem Zusammenhang ist die Diskretisierung von Wertebereichen erforderlich, um Profile als endliche Menge beschreiben zu können. Im Adapt.Net-Projekt definieren Profile deshalb Bereiche verfügbarer Ressourcen und Anwendungseigenschaften, das heißt, dass z.B. die verfügbare Bandbreite als Bereich 10 MBit/s - 100 MBit/s angegeben wird.

5.2.1 Reglerbasierte Parameteradaption

Basierend auf den vorgestellten Profilen kann die reglerbasierte Adaption von Kapselparametern verwendet werden (siehe [Li und Nahrstedt 1999]). Während Profile

zunächst relativ grobgranular die Anpassung an Umgebungssituationen durch die Angabe von Wertebereichen definieren, können feingranulare Anpassungen auf der Basis von Kapselparametern vorgenommen werden. Innerhalb eines Bereichs bestimmter Umgebungseigenschaften können Kapselparameter in Abhängigkeit dieser Eigenschaften eingestellt werden. Hierfür können zusätzliche Zusammenhänge zwischen Ressourcen, Umgebungs- und Kapselparametern der Form P = f(R,U) innerhalb der Profile definiert werden. Während der Überwachung der Umgebungseigenschaften können dann mit Hilfe dieser Zusatzinformationen feingranulare Anpassungen vorgenommen werden.
Nachdem nun grundlegende Konzepte der Zuordnung von Umgebungssituation und Anwendungskonfiguration beschrieben wurden, sollen im Folgenden die Implementierung wichtiger Infrastrukturkomponenten und Mechanismen im Rahmen des Adapt.Net-Projekts vorgestellt werden.

5.3 Der Adaptionsmanager

Ein zentraler Bestandteil adaptiver Anwendungen ist der Adaptionsmanager, welcher die Umgebungssituation einer Anwendung bestimmt und an Hand eines Profils eine entsprechende Anwendungskonfiguration auswählt und diese über den Konfigurationsmanager aktiviert. Der Adaptionsmanager, im Rahmen der Arbeit als *Adaptation Engine* bezeichnet, lädt vor dem Start der eigentlichen Anwendung Messfühler, die als **Observer** bezeichnet werden, für die Messung der entsprechenden Umgebungseigenschaften und leitet bei Verlassen der definierten Bereiche die Rekonfiguration der Anwendung über den Konfigurationsmanager ein. Die Funktionsweise der *Adaptation Engine* wird im Folgenden erläutert.

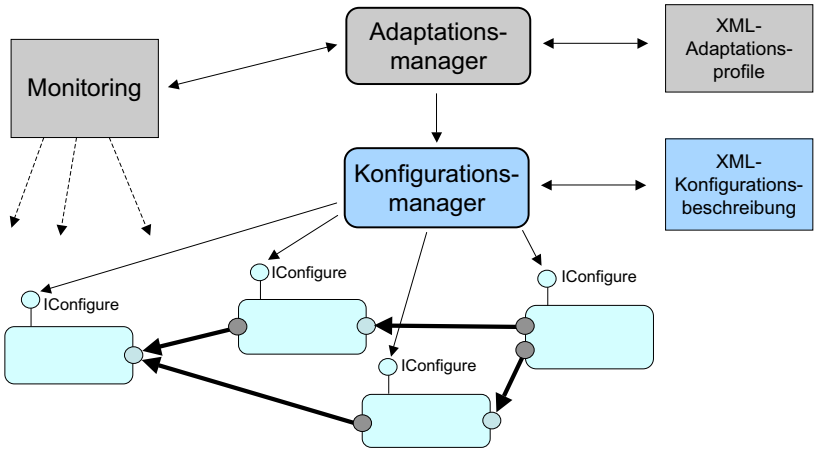


Abbildung 5.1: *Adaptation Engine* und Konfigurationsinfrastruktur

Abbildung 5.1 zeigt die Einordnung der *Adaptation Engine* in das Adapt.Net-Projekt. Die *Adaptation Engine* ist dem primären Konfigurationsmanager vorgelagert und löst die Änderung der Anwendungskonfigurationen über diesen aus. Im Prinzip wird die *Adaptation Engine* entkoppelt von der eigentlichen Anwendung ausgeführt und kann auch erst später bei laufender Anwendung aktiviert werden. Die Rekonfiguration über den Konfigurationsmanager kann manuell über eine Konfigurations-Kommandozeilenschnittstelle ausgelöst werden oder automatisch von der *Adaptation Engine*. Diese

nutzt dafür ein ereignisbasiertes *Monitoring*-System für die Beobachtung von Umgebungseigenschaften sowie der Kapseln der Anwendung.

5.3.1 Monitoring

Beobachtbare Parameter werden in Adapt.Net in vier Kategorien unterteilt:

- verfügbare Ressourcen (z.B. Netzbandbreite, CPU-Auslastung, Batterieleistung)
- Eigenschaften der Umgebung (z.B. Position des Nutzers)
- Laufzeitzustand der Kapseln (geladen, gestartet, abgestürzt, beendet)
- kapselinterne Variablen (z.B. ein Attribut *antwortZeit* der Hauptklasse)

Die vorgestellte Klassifikation resultiert aus den verschiedenen Verfahren für die Messung der einzelnen Parameter. Die allgemeine Bestimmung von Umgebungsparametern wird von gesonderten *Observern* (*environment observer*) realisiert, die separat geladen werden und die Beobachtung beliebiger Umgebungseigenschaften und verfügbarer Ressourcen realisieren können. Einige Parameter lassen sich aber nur innerhalb von Kapseln bzw. mit Unterstützung des Konfigurationsmanagers ermitteln. Darunter sind einige nichtfunktionale Eigenschaften, wie z.B. eine Ende-zu-Ende-Antwortzeit. Da die *Observer* keinen Zugriff auf den internen Zustand der Kapseln haben, musste eine neue Möglichkeit für die Messung kapselinterner Parameter geschaffen werden. Dies wurde durch den möglichen Zugriff auf Attribute des Hauptobjekts der Kapseln realisiert. Um auf den Ausfall von Kapseln reagieren zu können, musste die Überwachung des Laufzeitzustands von Kapseln geschaffen werden. Dieser kann mit Hilfe der schon vorgestellten Kapselkonfiguratoren realisiert werden. Die Behandlung von Ausfällen und ein entsprechendes Fehlermodell werden im anschließenden Kapitel diskutiert.

```
public interface IMonitor{
  event OnChange onChangeEvent;
  void SetBorders(Range[] ranges);
  bool StartObservation();
  bool StopObservation();
  object GetValue();
}
```

Abbildung 5.2: Die *Observer*-Schnittstelle *IMonitor*

Observer sind Objekte eines Typs, welche die Schnittstelle *IMonitor* implementieren, die in Abbildung 5.2 dargestellt ist. Ausgehend vom Profil werden die *Observer* vor dem Anwendungsstart geladen, indem ein Objekt eines spezifizierten Typs aus einer angegebenen Assembly erzeugt wird. Die *Adaptation Engine* kann eine Ereignisbehandlungsroutine bei jedem *Observer* registrieren (*onChangeEvent*), die bei Verlassen der definierten Bereiche aufgerufen wird. Die Bereiche ergeben sich durch die Analyse der Profile. Je nach Implementierung der *Observer* wird beim Aufruf der Methode *StartObservation* ein Thread gestartet, der periodisch einen Messwert überprüft oder eine Ereignisbehandlungsroutine für ein unterliegendes ereignisbasiertes System (z.B. einen *Zeitgeber*) registriert. Die Periodendauer der Überprüfung kann durch die *Observer*-Implementierung oder über *Observer*-Parameter konfiguriert werden.

Observer können als wiederverwendbare Komponenten implementiert werden und anwendungsübergreifend Verwendung finden. Im Adapt.Net-Projekt wurden eine Reihe von Standard-*Observern* implementiert, die dem Entwickler adaptiver Anwendungen zur Verfügung stehen. Darunter sind ein Zeitpunkt- und Zeitintervall-*Observer* sowie *Observer* für die Bestimmung einer Rechnerverfügbarkeit (via *ICMP Ping*) und der CPU-Auslastung. Mit der vorgestellten Architektur ist auch die Beeinflussung von Adaptionsentscheidungen durch den Nutzer realisierbar. In einigen Szenarien ist es sinnvoll, den Nutzer in die Adaptionsentscheidung einzubeziehen und z.B. über separate Menüs die Konfiguration bestimmter Präferenzen zu ermöglichen. Diese zusätzliche Konfigurationsmöglichkeit kann sehr leicht als *Observer* implementiert werden, indem die Interaktion mit dem Menü die entsprechenden „Messwerte“ (den eingegebenen Wert) im *Observer* einstellt.
Bei der Erstellung neuer *Observer* können Entwickler auf die abstrakte Klasse *ObserverImpl* zurückgreifen, die das periodische Überprüfen eines Messwertes implementiert. Der Entwickler muss nur die eigentliche Messung in der Methode *Measure* implementieren und den aktuellen Messwert in einem Attribut des *Observers* speichern. Zusätzlich kann die Periodendauer der Überprüfung in der Variable *sleepInterval* konfiguriert werden. Die Klasse *ObserverImpl* implementiert zusätzlich die Logik zur Überprüfung der Bereichsgrenzen, die damit nicht mehr vom Entwickler implementiert werden muss.
Die Abfrage von **Kapselvariablen** (*CapsuleProperty*) erfolgt über den Konfigurationsmanager, der die Anfrage an den entsprechenden Kapselkonfigurator weiterleitet. Die Kapselkonfiguratoren implementieren die Abfrage je nach Kapseltyp unterschiedlich. Bei *In-Process*-Kapseln erfolgt die Abfrage mit Hilfe der Methoden der Reflexionsschnittstellen. Im Falle von *Remoting*- und *Separate-Process*-Kapseln wird eine Schnittstelle für die Abfrage über Prozessgrenzen mit Hilfe von *.NET Remoting* zur Verfügung gestellt. Diese Funktionalität wurde in den entwickelten Konfigurationsaspekt integriert.
Die Überwachung des **Lebenszykluszustands** (über *CapsuleObserver*) wird in den Kapselkonfiguratoren implementiert. Die *Adaptation Engine* kann den Lebenzykluszustand einer Kapsel über die Schnittstelle *ICapsuleConfigurator* mit Hilfe der *Property State* überprüfen. Der Rückgabewert der Methode ist ein Wert der Enumeration: (*Created,Unloaded,Crashed, Running, UnderReconfiguration*). Bei der Erkennung von Kapselausfällen muss zunächst ein Fehlermodell diskutiert werden. Dieses wird im Kapitel 6 zusammen mit der Implementierung der Erkennung von Kapselausfällen vorgestellt. Die Schnittstelle der Kapselkonfiguratoren enthält auch ein Ereignis (*event*), welches bei einer Änderung des Lebenszykluszustands ausgelöst werden kann. Die *Adaptation Engine* registriert eine Ereignisbehandlungsroutine bei denjenigen Kapseln, die überwacht werden sollen, und kann eine Änderung der Konfiguration über den Konfigurationsmanager auslösen, wenn dies im Profil gefordert wurde.

5.3.2 Definition von Profilen

Profile werden durch den Anwendungsentwickler in einem XML-basierten Dokument definiert. Ein Profil besteht aus zwei Teilen: der Beschreibung der benötigten Messwerte, den zugehörigen *Observern* bzw. Kapselvariablen und Lebenszykluszustand, sowie der Zuordnung von Messwertbereichen zu Anwendungskonfigurationen. Das XML-Wurzelelement eines Profils trägt den Namen *AdaptationDescription*. Mögliche Kindelemente des Wurzelelements sind *Profiles* und *Monitoring*, welche die beiden zuvor erwähnten Teile eines Anpassungsprofils enthalten. Das XML-Element *Profiles*

enthält als Kind-Elemente *Profile*, denen als XML-Attribute ein Name (*name*) sowie eine Konfiguration (*configuration*) in Form eines Namens zugeordnet sind. Ein *Profile* kann mehrere XML-Elemente *ProfileEntry* enthalten, denen folgende XML-Attribute zugeordnet sind:

- *observerName*: der Name des messenden *Observers*
- *min*: die untere Schranke des Bereiches
- *max*: die obere Schranke des Bereiches

Das XML-Element *Monitoring* enthält als Kind-Element ein oder mehrere XML-Elemente namens *Observer*, denen folgende XML-Attribute zugeordnet werden können:

- *name*: der Name des entsprechenden *Observers* für die Zuordnung im Profilteil
- *observedValueType*: der Datentyp der gemessenen Größe
- *monitorType*: die Art der Messung *(CapsuleProperty, CapsuleObserver, EnvironmentObserver)*
- *capsuleName*: der Name der überwachten Kapsel im Fall *CapsuleProperty* oder der Überwachung des Lebenszykluszustands einer Kapsel (Der Name muss mit einem Kapselnamen innerhalb der Konfigurationsbeschreibung übereinstimmen.)
- *capsulePropertyName*: der Name der überwachten kapselinternen Variablen
- *assemblyName*: der Name der *Observer*-Assembly im Fall *EnvironmentMonitor*
- *className*: der Name der zu erzeugenden Klasse im Fall *EnvironmentMonitor*
- *locationName*: der Rechner auf dem der *Observer* geladen werden soll

Zusätzlich kann ein XML-Element *Observer* XML-Elemente namens *Options* enthalten, dessen Kindelemente *OptionEntry*, die XML-Attribute *key* und *value* für die Definition *Observer*-spezifischer Optionen enthalten.

```
<AdaptationDescription Version="1.2">
  <Profiles>
      <Profile profilename="Morning" configuration="configuration1">
        <profileentry observername="HourObserver" min="8" max="12" />
    </Profile>
    <Profile profilename="Evening" configuration="configuration2">
      <profileentry observername="HourObserver" min="13" max="18" />
    </Profiles>
    <Monitoring>
      <Observer name="HourObserver" observedValueType="System.Byte"
        MonitorType="EnvironmentObserver" AssemblyName="StdObs.dll"
        ClassName="AdaptNet.Monitoring.HourObserver"
        LocationName="localname:8017">
      </Observer>
    </Monitoring>
</AdaptationDescription>
```

Abbildung 5.3: Beispiel einer Profil- und Monitoring-Beschreibung

Abbildung 5.3 zeigt ein Beispiel eines Profils. Abhängig von der aktuellen Stunde der Uhrzeit wird entweder *configuration1* oder *configuration2* der Anwendung ausgeführt.

Der Uhrzeit-*Observer* wird aus der Assembly *StdObs.dll* geladen und ist durch den Typ *AdaptNet.Monitoring.HourObserver* implementiert. Die *Adaptation Engine* lädt das Profil einer adaptiven Anwendung vor deren Start, erzeugt die entsprechenden *Observer* und ermittelt den initialen Systemzustand. Nachdem über den Konfigurationsmanager alle im Profil referenzierten Konfigurationen in die Hauptspeicherstrukturen überführt wurden, wird die initiale Konfiguration geladen.

5.3.3 Auswahl der optimalen Konfiguration

Beim Start der Anwendung und beim Verlassen des Wertebereichs eines Profileintrags muss die *Adaptation Engine* eine neue Konfiguration auswählen. Die im Profil innerhalb von Profileinträgen definierten Bereiche werden beim Einlesen in eine Speicherrepräsentation überführt. Nach der Messung der erforderlichen Parameter wird ein Profileintrag ermittelt, dessen enthaltene Bereiche die gemessenen Werte einschließen. Für jeden Profileintrag wird jeder Bereich durch je einen Vergleich mit den min/max-Werten getestet. Im Prinzip kann es bei dieser Suche mehrere passende Lösungen geben, da die Bereiche der Profileinträge überlappen können. Die in der *Adaptation Engine* implementierte Suche ist als lineare Suche über alle Profile implementiert. Die Komplexität ist damit linear bezüglich der überwachten Parameter und Profilanzahl. Durch die Ordnung von Profileinträgen könnte an dieser Stelle die Berechnungsdauer für die Profilsuche optimiert werden. Da allerdings bei den gewählten Anwendungsdomänen typischerweise eine geringe Anzahl von Profilen untersucht wird, wurden keine weiteren Optimierungen implementiert.

5.4 Ein Werkzeug zur Erstellung adaptiver Anwendungen

Um die Entwicklung adaptiver Anwendungen basierend auf der vorgestellten Infrastruktur zu vereinfachen, wurde ein graphisches Werkzeug zur Erstellung von Anwendungskonfigurationen, der Beschreibung des *Monitoring*-Systems und Profilen entwickelt. Hierbei stand nicht das graphische Werkzeug selbst im Vordergrund, sondern die Techniken zur Generierung einer auslieferbaren Anwendung mit Hilfe der Vorgaben des Anwendungsentwicklers. Bei der Entwicklung adaptiver Anwendungen müssen die Entwickler drei Teilaufgaben bearbeiten. Zunächst müssen alternative Anwendungskonfigurationen durch die Auswahl von Assemblies, der Parametrierung, Verbindung und Verteilung von Kapseln erstellt werden. Als Zweites müssen die zu beobachtenden Umgebungsparameter durch die Beschreibung des *Monitoring*-Systems definiert werden und letztlich den gemessenen Werten im Rahmen von Profilen die zuvor erstellten Konfigurationen zugeordnet werden. Die Realisierung dieser drei Aufgabenbereiche wurde in das Werkzeug *AdaptationProfileCreator* integriert.

5.4.1 Der AdaptationProfileCreator

Der *AdaptationProfileCreator* wurde als eigenständige .NET-Anwendung implementiert. Konzeptionell sind die entwickelten Module aber sehr leicht in existierende Entwicklungswerkzeuge wie *Visual Studio.NET* oder *Eclipse* integrierbar. Der *AdaptationProfileCreator* umfasst drei Module, die über eine gemeinsame Navigationsleiste vereinigt werden. Abbildung 5.4 zeigt ein Bildschirmfoto des *AdaptationProfileCreators*. Die Navigationsleiste auf der linken Seite enthält in einer Ordnerstruktur alle definierten Anwendungskonfigurationen, eingerichtete *Observer* sowie alle definierten Profile. Der

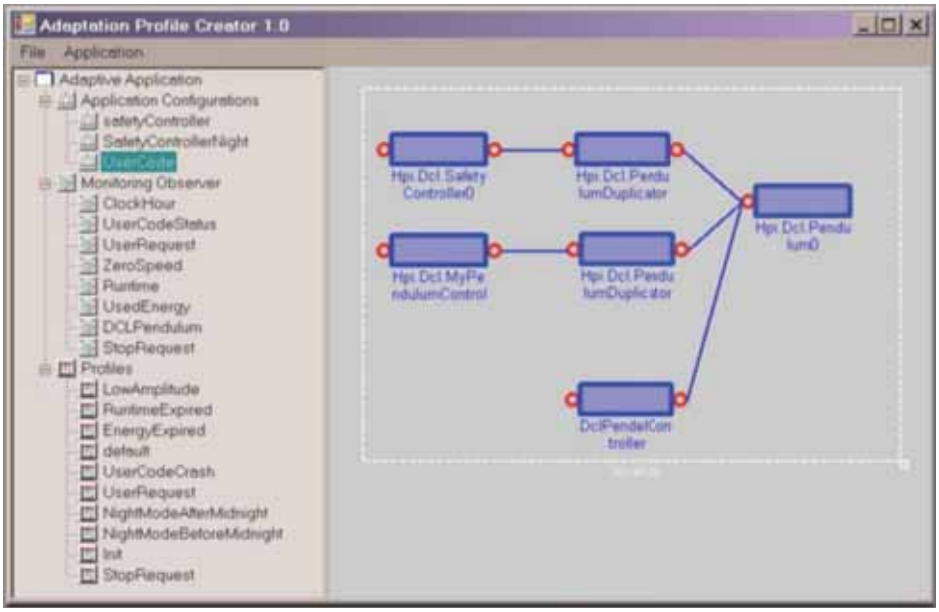


Abbildung 5.4: AdaptationProfileCreator Werkzeug

Anwendungsentwickler kann jedem Ordner neue Elemente hinzufügen und diese bearbeiten. Durch Selektion eines Elements in der Navigationsleiste wird auf der rechten Seite eine Detailansicht eingeblendet, innerhalb derer die Bearbeitung der jeweiligen Elemente erfolgt. Im Beispiel enthält die rechte Seite eine Anwendungskonfiguration. Die jeweiligen Module für die Detailansichten wurden als *UserControls* implementiert und können damit auch in anderen Werkzeugen Einsatz finden. *UserControls* bestimmen nutzerdefinierte Bestandteile einer graphischen Oberfläche in .NET. Während die ersten Versionen des Werkzeugs auf dem statischen Aspektweber *Loom.Net* basierten, wurde später der dynamische Aspektweber Rapier-Loom.Net integriert. Dies ermöglicht die Konfiguration von Aspekten durch Annotationen im Quelltext. Über ein Menü kann aber immer noch einer der beiden Aspektweber für die Generierung ausgewählt werden. Der statische Aspektweber bietet gegenüber dem dynamischen geringe Laufzeitvorteile.

5.4.2 Erstellung von Anwendungskonfigurationen

Anwendungskonfigurationen können mit dem *UserControl ConfigurationCreator* erstellt werden, welches auf der rechten Seite von Abbildung 5.4 zu sehen ist. Der Anwendungsentwickler kann einer Konfiguration neue Kapseln über das Kontextmenü *neu* hinzufügen. Nach der Auswahl einer Assembly aus dem Dateisystem wird die Hauptklasse der Kapsel festgelegt. Über das Menü *Eigenschaften* können die Werte von Parametern sowie die Ausführungsart der Kapsel definiert werden. Konnektoren können durch Anklicken eines Ports, die als Kreise am Rand der rechteckigen Kapseln dargestellt sind, und das Verbinden eines anderen Ports bei gedrückter Maustaste erstellt werden. Der Konnektortyp kann über ein separates Menü definiert werden. Die Ports werden durch eine Analyse der Metadaten der Hauptklasse entsprechend dargestellt. Mit Hilfe von „*Tooltips*“ werden die Namen der Quellattribute bzw. die Typnamen der Schnittstellen angezeigt. Schließlich kann der Anwendungsentwickler die Ausführungsrechner einer Kapsel durch das Hinzufügen neuer Rechner (*locations*) festlegen. Durch

die Platzierung einer Kapsel innerhalb des repräsentierenden Rechtecks wird der Rechner zugeordnet. Neue Kapseln können auch mit der Hilfe *Drag&Drop*-Funktionalität der *Windows*-Betriebssysteme hinzugefügt werden, indem eine Assembly aus einem Dateimanager auf die Fläche des *ConfigurationCreators* gezogen wird.
Zusätzlich können in eine Konfiguration **externe Referenzen** und damit auch SOAP-basierte Web-Dienste integriert werden. Über ein separates Menü können spezielle Eigenschaften wie eine URL spezifiziert werden. Eine weitere wichtige Aufgabe bei der Konfigurationserstellung ist die statische **Überprüfung der Konsistenz**. Im *ConfigurationCreator* wurden einige Einschränkungen für Konfigurationen definiert. Die Typen von Quellattributen und Schnittstellen der Konnektoren müssen kompatibel sein, Konnektoren über Rechnergrenzen Fernaufrufe unterstützen und alle Quellattribute mit entsprechenden Kapseln verbunden sein. Zusätzlich wird überprüft, dass für alle Kapselparameter Werte angegeben wurden.

5.4.3 Konfiguration der Monitoring-Umgebung

Für das Hinzufügen von *Observern* wurden spezielle Dialoge entwickelt, welche die Konfiguration vereinfachen. Der Entwickler kann zunächst zwischen der Überwachung von Kapselvariablen und Umgebungsparametern oder dem Lebenszykluszustand wählen. Bei der Auswahl von Kapselvariablen wird eine Liste sämtlicher in den definierten Konfigurationen vorhandenen Kapseln erstellt, die dann vom Entwickler selektiert werden können. Für jede Kapsel werden beobachtbare Variablen mit Hilfe von Reflexion analysiert und wieder dem Entwickler als Liste präsentiert. Als Ergebnis werden die Namen einer Kapsel und eines Attributes der Hauptklasse vom Entwickler gewählt. Die Kapsel mit dem gewählten Namen kann in mehr als einer Konfiguration vorhanden sein. *Observer* für Umgebungsparameter werden durch zusätzliche Dialoge für die Selektion einer Assembly, eines Typnames und der Konfiguration von Optionen definiert. *Observer* für den Lebenszykluszustand bedürfen nur der Auswahl des Namens der zu überwachenden Kapsel. Zusätzlich wird für jeden *Observer* automatisch mit Hilfe der Metadaten der dem Messwert zu Grunde liegende Datentyp eingestellt.

5.4.4 Erstellen von Profilen

Beim Hinzufügen von Profilen wird eine Auswahlliste aller definierter *Observer* erstellt. Der Entwickler kann für ein Profil *Observer* selektieren und für diese entsprechende min- und max-Werte eingeben. Eine zusätzliche Auswahlbox bietet pro Profil die Auswahl einer zugehörigen Konfiguration aus den definierten Konfigurationen an. Bei der Eingabe der Wertebereiche werden im Werkzeug entsprechende Typüberprüfungen vorgenommen.

5.4.5 Die Generierungsphase

Nachdem alle benötigten Teile einer adaptiven Anwendung erstellt wurden, kann ein auslieferbares „Paket“ in Form eines Verzeichnisses generiert werden. Dabei werden alle benötigten Dateien in einer speziellen Verzeichnisstruktur abgelegt. Als erstes werden die einzelnen Anwendungskonfigurationen in einem vom Entwickler spezifizierten Verzeichnis erzeugt. In dem angegebenen Zielverzeichnis wird für jede Konfiguration ein Unterverzeichnis mit dem Namen *Configuration_Name*, wobei *Name* durch den Konfigurationsnamen ersetzt wird, angelegt.

Generierung der Anwendungskonfigurationen: Ausgehend von der Haupt-Assembly jeder Kapsel werden alle benötigten Assemblies der Anwendung berechnet. Dazu zählen die Haupt-Assemblies selbst sowie alle Assemblies der Abhängigkeitshülle (HUL). Alle benötigten Dateien werden in ein Unterverzeichnis *bin* kopiert. Danach wird für jede Kapsel konfigurationsspezifische Logik in einer separaten Assembly durch den Aspektweber erzeugt. Mit welchen Aspekten eine Kapsel verwoben wird, hängt unter anderem von ihrer Ausführungsart (*In-Process, Remoting, SeparateProcess*) ab. Die verschiedenen Aspekte unterscheiden sich dabei nur in wenigen Details. Im Wesentlichen werden die in Kapitel 3 vorgestellten Aspekte verwendet. Die erzeugten Assemblies werden im Unterverzeichnis *bin* gespeichert. Zusätzlich kann im *AdaptationProfileCreators* ein Zertifikat konfiguriert werden, mit dem alle erzeugten Assemblies signiert werden. Das Sicherheitskonzept in Adapt.Net nutzt einen Mechanismus der CLI, bei dem Sicherheitsrichtlinien an Hand der Signatur einer Assembly festgelegt werden können (siehe Abschnitt 3.5.2).
Im nächsten Schritt wird die XML-basierte Konfigurationsbeschreibung erzeugt. Dabei werden die originalen Namen der Hauptklassen sowie die Namen der Haupt-Assemblies durch die Assemblies mit der verwobenen Logik (den Proxys) ersetzt. Die Konfigurationsbeschreibung wird im Verzeichnis der generierten Anwendung unter dem Namen *config.xml* gespeichert. Aus Leistungsgründen wird für die dynamische Aktualisierung von Kapseln benötigter Code vorabgeneriert (siehe Abschnitt 4.2.3), wenn dies innerhalb der Menge der definierten Konfigurationen erkannt wird (gleicher Assembly-Name, unterschiedliche Versionsnummer).
Zum Abschluss werden die von der CoFRA-Infrastruktur benötigten Assemblies in das Zielverzeichnis kopiert. Zusätzlich wird die Assembly *StartConfiguration.exe* erstellt. Diese ermöglicht den Start jeder einzelnen Konfiguration aus dem Zielverzeichnis heraus und kann für Testzwecke verwendet werden.

Erzeugung der Anwendung: Nachdem alle Konfigurationen einer Anwendung im Zielverzeichnis erstellt wurden, muss die Adaptionslogik erzeugt werden. Zunächst werden benötigte *Observer*-Assemblies in das Zielverzeichnis kopiert, gefolgt von den Assemblies der *Adaptation Engine*. Die im Werkzeug spezifizierten Profile werden in das beschriebene XML-Format persistiert und in einer Datei im Zielverzeichnis gespeichert. Bei der Erstellung der Profil-Datei werden die Konfigurationen entsprechend den zuvor generierten Verzeichnissen zugeordnet. Abschließend wird die ausführbare Datei *StartApplication.exe* in das Zielverzeichnis kopiert. Mit dieser kann die Anwendung gestartet werden.
Das vom *AdaptationProfileCreator* erzeugte Verzeichnis kann durch Kopieren auf einen Zielrechner installiert und dort ausgeführt werden. Dabei muss nur zuvor ein Exemplar des Konfigurationsmanagers auf jedem beteiligten Rechner gestartet werden.

5.5 Architekturmuster für adaptive Anwendungen

Das vorgestellte Adapt.Net-Projekt ermöglicht die Ausführung und Anpassung alternativer Anwendungskonfigurationen in Abhängigkeit verschiedener Umgebungssituationen. Wie alternative Konfigurationen für verschiedene Situationen erstellt werden können, wird im Folgenden durch die Vorstellung von Architekturmustern demonstriert. An dieser Stelle sollen nicht ausschließlich neu entwickelte Architekturmuster

beschrieben werden, sondern an Hand von existierenden Mustern Möglichkeiten beschrieben werden, alternative Anwendungskonfigurationen zu entwickeln. Die vorgestellten Muster beziehen sich auf die Einführung neuer Kapseln und Konnektoren in eine Anwendungskonfiguration. Diese Art von Mustern wird im Bereich der Softwarearchitektur auch als Architekturmuster bezeichnet. Die Erstellung von alternativen Anwendungskonfigurationen unter Verwendung der vorgestellten Muster wird durch das entwickelte Werkzeug unterstützt. Architekturmuster können auf Kapseln oder Konnektoren angewendet werden. Durch Anklicken der entsprechenden Elemente in der Konfigurationsansicht kann aus einer Liste ein Muster ausgewählt werden, für das teilautomatisiert die notwendigen Konfigurationsänderungen erzeugt werden. Nur anwendungsspezifische Anteile müssen vom Entwickler ergänzt werden.
Bei der Vorstellung der Muster werden verschiedene Gesichtspunkte für alle Muster diskutiert. Dazu gehört die graphische Darstellung in Form von Konfigurationen, eine Analyse, welche nichtfunktionalen Eigenschaften bzw. Ressourcenanforderungen verändert werden und in welchem Szenario das Muster am besten Einsatz findet, Voraussetzungen und Einschränkungen für den Einsatz sowie Details zur Integration in das Adapt.Net-Konfigurationserstellungswerkzeug.

5.5.1 Das Architekturmuster Filter

Beschreibung: Eine zusätzliche Kapsel verändert die übertragenen Daten von Methodenaufrufen zwischen zwei Kapseln. Das Architekturmuster ist in Abbildung 5.5 dargestellt.

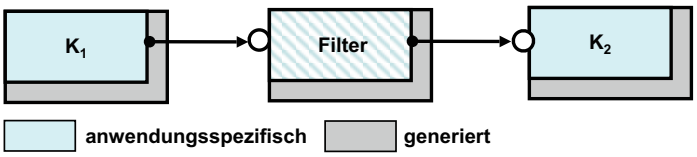


Abbildung 5.5: Architekturmuster Filter

Einsatzszenarien: Bei reduzierter Bandbreite können Daten (Methodenparameter) komprimiert werden. Dies kann anwendungsspezifisch unter Verwendung spezieller Kompressionsverfahren (z.B. JPEG bei Bilddaten) oder durch verlustfreie generische Kompressionsalgorithmen (z.B. ZIP) erfolgen. Diese Filterart wird auch als Kompressionsfilter bezeichnet. Kompressionsfilter verkürzen die Antwortzeit einer Anwendung, wenn einer hohen Rechenleistung für die Kompression eine geringe Übertragungsbandbreite für die Daten gegenübersteht. Filter können auch aus zwei korrespondierenden Kapseln bestehen, von denen eine z.B. Daten komprimiert und die andere auf einem zweiten Rechner wieder expandiert. Verschlüsselungsfilter erhöhen die Sicherheit, erfordern aber zusätzliche Rechenressourcen und Speicher. Typisch für Filter sind Parameter, die je nach Situation feingranulare Anpassungen ermöglichen. Ein Beispiel ist ein Kompressionsfaktor für einen Kompressionsfilter.
Beispiele: Es existieren zahlreiche Beispiele, die dieses Muster verwenden. Typischerweise findet es bei der datenorientierten Adaption Einsatz (siehe Abschnitt 8.1.3). Anwendungsspezifische Filter beginnen bei der Umwandlung von PDF- in Text-Dokumente und der Kompression von Bildern mit Hilfe des JPEG- oder GIF-Formats und enden bei der Anpassung der Auflösung, Framerate und Kompressionsfaktoren bei Videodaten.

Werkzeugintegration: Filter können auf Konnektoren angewendet werden. Als Resultat werden ein neuer Konnektor sowie eine zusätzliche Kapsel, welche die gleiche Schnittstelle wie die Senkenkapsel des Konnektors implementiert, erzeugt. Die Filterlogik kann entweder in Form einer generischen Version in einer Methode gegeben sein, die für jeden Methodenaufruf ausgeführt wird, oder der Entwickler stellt einen generierten Coderumpf fertig. Das Adapt.Net-Werkzeug unterstützt wiederverwendbare Kompressions- und Verschlüsselungskapseln.
Einschränkungen: Die Methodenparameter müssen in der gewählten Form veränderbar sein. Für die Anwendung von Kompressionsfiltern müssen die Methodenparameter bzw. Rückgabewerte komprimierbar sein.

5.5.2 Das Architekturmuster Intelligenter Proxy

Beschreibung: Eine zusätzliche Kapsel ist auf Quellseite eines rechnerübergreifenden Konnektors integriert. Im Falle eines Ausfalls der Verbindung bzw. des entfernten Rechners (*crash fault*) kehrt der Methodenaufruf nicht mit einer strukturierten Ausnahme (*Exception*) zurück, sondern vermerkt den Ausfall und wartet auf die Fehlerbehebung. Bei einem permanenten Ausfall kann eine alternative Konfiguration durch den Konfigurationsmanager aktiviert und der Methodenaufruf transparent für die Quellkapsel fortgesetzt werden. Bei transienten Verbindungsausfällen kann durch eine zusätzliche Kapsel auf Senkenseite der Ausfall durch Wiederholung fehlgeschlagener Aufrufe toleriert werden.
Einsatzszenarien: Dieses Muster kann zur Erhöhung der Zuverlässigkeit einer verteilten Anwendung eingesetzt werden. Bei der Beteiligung mobiler Geräte kann bei längerem Verlust der Konnektivität transparent eine neue Konfiguration aktiviert werden, die lokal arbeitet. Vor allem die Umschaltung zwischen verschiedenen Kommunikationsnetzwerken (z.B. LAN nach UMTS) kann mit Hilfe dieses Musters transparent realisiert werden. Hierfür ist für den allgemeingültigen Einsatz eine Erweiterung des beschriebenen Musters notwendig.
Das beschriebene Verfahren funktioniert uneingeschränkt für Methodenaufrufe, die keinen Zustand verändern. Eine Wiederholung der Aufrufe ist aber nicht in jedem Fall möglich, da nicht klar ist, ob während eines Ausfalls aktive Aufrufe ausgeführt wurden oder nicht. Eine **Erweiterung** des intelligenten Proxys durch eine zusätzliche Kapsel auf Senkenseite, die den Status von Aufrufen speichert, behebt dieses Problem. Ein Aufrufidentifikator kann im Kontext der Aufrufe übertragen werden und diese eindeutig zuordnen. Unter Nutzung der im Rahmen dieser Arbeit schon vorgestellten logischen Thread-IDs kann für jede Aktivität der Rückgabewert in einem Cache mit dem zugehörigen Aufrufidentifikator gespeichert werden. Ist eine temporär ausgefallene Verbindung wieder hergestellt, kann der Proxy für jeden (logischen) Thread auf der Senkenseite überprüfen, ob ein während des Ausfalls aktiver Aufruf ausgeführt wurde oder wiederholt werden muss. Bei schon erfolgter Ausführung während eines Ausfalls wird der Wert aus dem Cache zum Proxy auf Quellseite übertragen, der diesen der Quellkapsel als Rückgabewert übergibt. Zu beachten sind an dieser Stelle mögliche rekursive Methodenaufrufe an der Senkenkapsel des Konnektors. Bei der Abfrage der Werte aus dem Cache muss sichergestellt sein, dass die gespeicherten Werte neben der logischen Thread-ID auch über die aktuelle Rekursionstiefe identifiziert werden. An dieser Stelle können die bei dem vorgestellten Rekonfigurationsalgorithmus eingesetzten Hash-Tabellen zum Einsatz kommen, über welche die aktuelle Rekursionstiefe ermittelt werden kann. Die Zusatzkosten dieser Erweiterung belaufen sich auf einen

zusätzlichen lokalen Methodenaufruf sowie auf die Zuweisung des Rückgabewertes an den Cache.
Werkzeugintegration: Der intelligente Proxy kann vollständig generiert werden. Eine Kapselvariable, welche von der *Adaption Engine* überwacht wird, repräsentiert den Zustand eines Konnektors. Der intelligente Proxy wird als *In-Process*-Kapsel quellseitig mit einem direkten Aufruf verbunden. Dies minimiert die Zusatzkosten für den Einsatz dieses Musters. Innerhalb der Kapsellogik werden Aufrufe an die Senkenkapsel weitergeleitet und strukturierte Ausnahmen gefangen sowie bei Verfügbarkeit einer neuen Senke die Aufrufe wiederholt.
Einschränkungen: Bei permanenten Ausfällen kann der Einsatz dieses Musters keinen potentiellen Zustandsverlust von Kapseln tolerieren. Für die Aktivierung einer alternativen Konfiguration dürfen keine abhängigen Transaktionen in der Quellkapsel existieren, die über Kapseln initiiert wurden, die an der Rekonfiguration beteiligt sind, da sonst die Synchronisationslogik verklemmen kann.

5.5.3 Das Architekturmuster Lastverteilung

Beschreibung: Anfragen einer Quellkapsel werden an replizierte Exemplare der Senkenkapsel eines Konnektors mit Hilfe einer Lastverteilungskapsel verteilt.

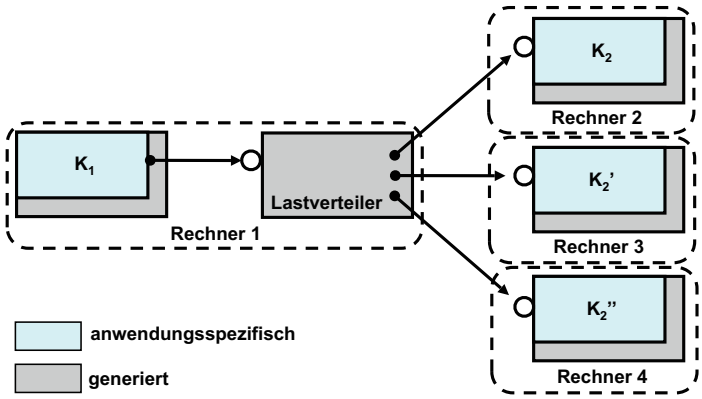


Abbildung 5.6: Architekturmuster Lastverteilung

Einsatzszenarien: Mit Hilfe dieses Musters kann die Antwortzeit einer Berechnung verkürzt werden, indem Anfragen parallel verarbeitet werden. Es werden hierfür zusätzliche Rechenressourcen benötigt. Bei Verfügbarkeit freier Rechenressourcen in einem verteilten System können die replizierten Kapseln auf diesen Rechnern ausgeführt werden.
Werkzeugintegration: Das Architekturmuster kann auf einen Konnektor angewendet werden. Es wird eine Lastverteilungskapsel generiert sowie Replikate der Quellkapsel des Konnektors und zusätzliche Konnektoren zu den replizierten Kapseln erzeugt. Die vom Werkzeug erstellte Lastverteilungskapsel verteilt Anfragen zufällig an replizierte Kapseln, welche über eine feste Menge (3) von Verbindungspunkten in der generierten Logik konfiguriert werden können. Die Aufrufe werden parallel über einen *Threadpool* verarbeitet.
Einschränkungen: Die Verarbeitung der Quellkapsel muss den parallelen Aufruf der entsprechenden Senkenkapsel unterstützen. Die replizierten Kapseln dürfen nicht schreibend auf gemeinsame externe Betriebsmittel, wie z.B. Datenbanken zugreifen.

5.5.4 Das Architekturmuster Replikation

Beschreibung: Aufrufe werden über eine zusätzliche Kapsel, die als *Voter* bezeichnet wird, an replizierte Kapseln weitergeleitet. Der *Voter* vergleicht die Rückgabewerte der Aufrufe. Bei unterschiedlichen Rückgabewerten wird der am häufigsten vorkommende Wert als Rückgabewert bestimmt. Dieses Architekturmuster ist eng verwandt mit der Lastverteilung. Die Lastverteilungskapsel ist durch die *Voter*-Kapsel ersetzt. Dieses Muster ist zusätzlich artverwandt mit dem intelligenten Proxy, welcher die „Replikation in der Zeit“ realisiert, während dieses Architekturmuster „Replikation im Raum“ unterstützt.
Einsatzszenarien: Mit Hilfe der Replikation können Ausfälle von Rechnern oder Kommunikationsverbindungen (crash fault) in einem verteilten System toleriert werden. Zusätzlich können transiente Fehler, die zu einer inkorrekten Berechnung führen (*incorrect computation*), erkannt und toleriert werden. Dies ermöglicht die Erhöhung der Zuverlässigkeit. Durch die Verwendung unabhängig voneinander implementierter Kapseln (*n-version programming* [Avizienis 1995]) kann die Zuverlässigkeit weiter erhöht werden.
Werkzeugintegration: In das Konfigurationserstellungswerkzeug wurde die dreifache Replikation von Kapseln integriert. Das Architekturmuster kann auf einen Konnektor angewendet werden, dessen Senkenkapsel verdreifacht wird. Zusätzlich werden die nötigen Konnektoren erstellt. Auf Grund des angenommenen Fehlermodells müssen die zusätzlichen Kapseln auf verschiedenen Rechnern erstellt werden und entsprechende Konnektoren mit Unterstützung für Fernaufrufe verwendet werden. Rückgabewerte sämtlicher Methoden der Schnittstelle müssen die Schnittstelle *IComparable* implementieren. Zusätzlich müssen alle mit dem CLI-Attribut *[out]* annotierten Methodenparameter diese Bedingung erfüllen. Die mit dem Aspektweber Loom.Net erzeugte *Voter*-Kapsel enthält drei Verbindungspunkte für die replizierten Senkenkapseln und implementiert die entsprechende Schnittstelle der Konnektorsenke. Loom.Net wurde für die Generierung eingesetzt, da sich mit dem Aspektweber sehr komfortabel Proxy-Klassen für eine Schnittstelle erstellen lassen, die automatisch mit einem gegebenen Code-Fragment gefüllt werden. Für die parallele Ausführung der Aufrufe wird ein *Threadpool* verwendet. Die generierte Kapsellogik vergleicht die Rückgabewerte der Aufrufe und ermittelt durch die Bestimmung der Mehrheit den korrekten Rückgabewert. Eine optimierte Version könnte ein Ergebnis schon nach der Rückkehr der Mehrheit der parallel ausgeführten Aufrufe liefern (wenn die Rückgabewerte gleich sind).
Einschränkungen: Es gelten die gleichen Einschränkungen für die Replikation wie bei der Lastverteilung. Zusätzlich müssen die Rückgabewerte der Methoden vergleichbar sein.

5.5.5 Das Architekturmuster Online/Offline Mobilität

Beschreibung: Ein mobiles Gerät nutzt in einem *Offline*-Modus eine Kapsel mit eingeschränkter Funktionalität, die den limitierten Ressourcen des Rechners angepasst ist. Bei Verfügbarkeit eines Netzwerks (*Online*-Modus) kann die Kapsel mit limitierter Funktionalität durch eine komplexe auf einem entfernten Rechner ersetzt werden.
Einsatzszenarien: Der Einsatz bietet sich bei mobilen Geräten mit partiellen und intermittierenden Kommunikationsressourcen an, wenn reduzierte Varianten der Funktionalität sinnvoll einsetzbar sind.

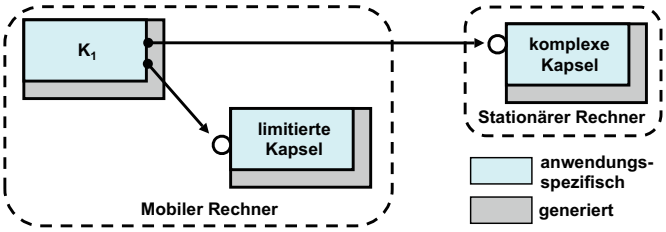


Abbildung 5.7: Architekturmuster Online/Offline Mobilität

Beispiele: Eine Navigationssoftware kann im Offline-Modus mit eingeschränktem Kartenmaterial arbeiten. Im Online-Modus kann die komplexe Kapsel ausführliches Kartenmaterial mit Zusatzinformationen bereitstellen.

5.5.6 Das Architekturmuster Sichere Aktualisierung

Beschreibung: Unter Benutzung des Architekturmusters Replikation wird eine neue Version einer Kapsel parallel zu einem Exemplar einer alten Version ausgeführt. Ist die neue Version fehlerhaft, werden die Ergebnisse der alten Version verwendet.
Einsatzszenarien: Neue Versionen einer Kapsel können potentiell Fehler enthalten. Deshalb kann es sinnvoll sein, für eine gewisse Zeit die als korrekt arbeitend bekannte alte Version einer Kapsel zu verwenden. Die alleinige Aktivierung der neuen Version kann dann erfolgen, wenn die neue Version für eine bestimmte Zeit fehlerfrei funktioniert.
Einschränkungen: Dieses Architekturmuster sollte nicht für das Erreichen adaptiven Verhaltens verwendet werden, da hier die einzelnen Konfigurationen als korrekt angenommen werden. Für die dynamische Aktualisierung von langlaufenden Softwaresystemen kann dieses Muster sehr gut eingesetzt werden. Die neue Version darf aber nicht das Ergebnis der Berechnung verändern, wohl aber z.B. durch den Einsatz verbesserter Algorithmen die Verarbeitungsgeschwindigkeit erhöhen.

5.5.7 Das Architekturmuster Simplex

Beschreibung: Das Architekturmuster Simplex [Sha 1998, 2001] wird ausführlich in Kapitel 6 vorgestellt. Die Verwendung analytischer Redundanz ermöglicht die Toleranz von Fehlern in komplexen Steuerungsanwendungen. Bei Verwendung dieses Musters werden 2 alternative Implementierungen einer Steuerung benötigt. Dabei enthält eine Version, welche als *Safety Controller* bezeichnet wird, einen erprobten Steuerungsalgorithmus, der sich z.B. durch langen erfolgreichen Einsatz als zuverlässig erwiesen hat. Eine zweite Version der Steuerung, die als *High Performance Controller* bezeichnet wird, enthält einen neueren bzw. optimierten, komplexen Steuerungsalgorithmus. Fällt die *High-Performance-Controller*-Kapsel aus, kann die Steuerung von der Logik der *Safety-Controller*-Kapsel übernommen werden.
Einsatzszenarien: Dieses Architekturmuster wurde im Distributed Control Lab zur sicheren Ausführung von Steuerungsexperimenten über das Internet verwendet.

5.5.8 Das Architekturmuster Migration

Beschreibung: Die Migration von Kapseln ist ein grundlegender Bestandteil der Adapt.Net-Ausführungsplattform, bei der die Ausführung einer Kapsel inklusive ihres

Zustands auf einen anderen Rechner verlagert wird.
Einsatzszenarien: Unter anderem im Bereich mobiler Anwendungen kann ein Teil der Funktionalität auf entfernte Rechner verlagert werden, sobald diese verfügbar sind. Im Bereich mobiler Umgebungen ist der Einsatz von Migration problematisch, da bei Verbindungsausfällen eine Rückmigration zu einem mobilen Gerät nicht ohne Weiteres möglich ist. Bestehen Kommunikationsverbindungen zuverlässig mit variierender Bandbreite, kann Migration zur Optimierung von Ende-zu-Ende-Antwortzeiten eingesetzt werden. Migration kann auch für die Mobilität von Nutzern in dem Sinne eingesetzt werden, dass beim Verlassen eines Arbeitsplatzrechners eine aktive Anwendung inklusive der graphischen Nutzerschnittstelle auf ein Laptop migriert wird. Diese Möglichkeit wird von Adapt.Net unterstützt.
Einschränkungen: Die Nutzung externer Ressourcen für migrierbare Kapseln ist eingeschränkt (siehe Abschnitt 3.7). Für den Zustandstransfer müssen zuverlässige Kommunikationsverbindungen zwischen den Rechnern existieren.
Werkzeugintegration: Der *AdaptationProfileCreator* unterstützt den Einsatz von Migration, indem Kapseln mit der Maus in der graphischen Repräsentation von einem Rechner zu einem anderen gezogen werden können.

5.5.9 Das Muster Dynamische Aspektaktivierung

Beschreibung: Durch die Annotation von Typen der Objekte einer Kapsel mit Aspekt-CLI-Attributen werden Aspekte für diese Typen aktiviert.
Einsatzszenarien: Die Einsatzmöglichkeiten dynamischer Aspekte sind vielfältig. Filter lassen sich mit dieser Strategie realisieren, indem die entsprechende Logik mit Hilfe der Aspekte direkt in die Logik der Kapseln integriert wird. Im Prinzip lassen sich beliebige nichtfunktionale Belange mit Hilfe dieser Strategie nutzen.
Werkzeugintegration: In einer ersten Version des *AdaptationProfileCreator* konnten Aspekte für den statischen Aspektweber Loom.Net über ein Menü in der Konfigurationsdarstellung selektiert werden. Seit der Verwendung des dynamischen Webers kann die Auswahl aktiver Aspekte zusätzlich direkt im Quellcode der Kapsel konfiguriert werden.
Beispiel: Ein Beispiel für die dynamische Aktivierung von Aspekten ist das *Logging* anwendungsspezifischer Daten. Im Falle hoher Systemlast ist es dabei nicht wünschenswert, zusätzliche Leistung für die Auswertung der anfallenden Daten zu verlieren. Die dynamische Aspektaktivierung kann benutzt werden, um einen *Logging*-Aspekt nur dann zu aktivieren, wenn keine hohe Systemlast vorliegt.

5.5.10 Das Muster Alternativer Algorithmus

Beschreibung: Die Implementierung der Logik einer Kapsel wird durch den Einsatz eines alternativen Algorithmus, der in einer gegebenen Einsatzumgebung besser geeignet ist, während der Laufzeit ersetzt.
Einsatzszenarien: Die für die Implementierung der Logik einer Kapsel verwendeten Algorithmen bestimmen wesentlich die nichtfunktionalen Eigenschaften der Anwendung. Der Einsatz alternativer Algorithmen für dynamische Umgebungseigenschaften ist damit eine wichtige Technik für die Realisierung adaptiver Anwendungen.
Werkzeugintegration: Die Realisierung alternativer Algorithmen erfolgt auf Basis von Assemblies. Die verschiedenen Algorithmen liegen in Assemblies verschiedener Ver-

sion und gleichen Namens vor. Im Werkzeug werden bei der Konfigurationserstellung unterschiedliche Haupt-Assemblies für die jeweiligen Konfigurationen gewählt.

5.5.11 Das Muster Parameteranpassung

Beschreibung: Über einen Parameter einer Kapsel werden deren nichtfunktionale Eigenschaften angepasst.
Einsatzszenarien: Die Parameteranpassung kann für die Feinjustierung von Algorithmen verwendet werden. Bei dem Einsatz eines Filters können u.a. Kompressionsparameter eingestellt werden. Mit Hilfe von Parameteranpassungen kann in eingeschränktem Rahmen, aber ohne große Beeinflussung der Anwendung adaptiert werden. Der Einsatz von Parameteranpassungen eignet sich zusammen mit Reglern, die gemessene Umgebungseigenschaften oder kapselinternen Variablen direkt auf Kapselparameter abbilden.

5.6 Zusammenfassung

Im Rahmen dieses Kapitels wurde eine Infrastruktur für die Beobachtung von Kontextparametern von Anwendungen vorgestellt, mit deren Hilfe und dem Einsatz von Profilen adaptives Verhalten durch Zuordnung geeigneter Anwendungskonfigurationen realisiert werden kann. Mit dem *AdaptationProfileCreator* wurde ein graphisches Werkzeug zur Erstellung adaptiver Anwendungen implementiert. Besonders hervorzuheben ist an dieser Stelle, dass mit Hilfe des Werkzeugs große Teile adaptionsspezifischer Logik generiert werden können und dadurch die Entwicklungszeit adaptiver Anwendungen verkürzt werden kann. Der Entwickler muss lediglich die Softwarekomponenten mit den mit Verbindungspunkten und Parametern annotierten Klassen in das graphische Werkzeug laden und durch die Verbindung der erzeugten Kapseln in der graphischen Repräsentation und der Zuordnung auf Rechner alternative Konfigurationen für die ermittelten Einsatzsituationen erstellen. Das Werkzeug ist in der Lage, Logik für das Erzeugen von Kapseln, das Setzen von Parametern und das Erstellen von Kommunikationskanälen sowie die für die dynamische Rekonfiguration notwendige Synchronisationslogik mit Hilfe der aspektorientierten Programmierung in die Kapsellogik zu integrieren. Ein großer Vorteil dieses Ansatzes ist auch, dass die erstellten Anwendungskonfigurationen unabhängig voneinander in den jeweiligen Einsatzszenarien getestet werden können und die Einhaltung vorgegebener nichtfunktionaler Anwendungseigenschaften überprüft werden kann. Während der Anwendungslaufzeit kann mit dem entwickelten Rekonfigurationsverfahren die jeweilige Konfiguration bei einem Wechsel der Umgebungssituation aktiviert werden.
Abschließend wurden in diesem Kapitel Muster für Entwicklung die alternativer Anwendungskonfigurationen vorgestellt und die dabei realisierte Werkzeugunterstützung beschrieben. Mit der Beschreibung der Muster konnte gezeigt werden, dass der in dieser Arbeit vorgestellte Ansatz zur Realisierung adaptiver Anwendungen zum einen existierende Ansätze einschließt (u.a. die datenorientierte Adaption - Abschnitt 8.1.3, die Adaption mit Aspekten - Abschnitt 8.1.6), zum anderen aber deren Funktionalität in vielen Bereichen wie der Werkzeugunterstützung, dem Einsatz alternativer Algorithmen oder der Unterstützung der Komponententechnologie erweitert.

6 Adaption in Steuerungssystemen

6.1 Das Distributed Control Lab

Das Distributed Control Lab (DCL) ist eine verteilte Laborinfrastruktur zur entfernten Ausführung von Steuerungsexperimenten. Die DCL-Infrastruktur wird seit dem Jahr 2002 unter anderen vom Autor dieser Arbeit am Hasso-Plattner-Institut entwickelt und eingesetzt. Im Rahmen des DCL wurden die in dieser Arbeit vorgestellten Techniken für die kontrollierte Ausführung potentiell fehlerhafter Steuerungsprogramme eingesetzt. Der Bezug zur Adaptivität ergibt sich über die Beobachtung von Steuerungsprogrammen als Umgebungsparameter und der dynamischen Rekonfiguration im Fehlerfall. Ergebnisse der Entwicklung des DCL schlugen sich in den Veröffentlichungen [Rasche und Polze 2005; Rasche u. a. 2005b, 2004, 2003; Richter u. a. 2007; Tröger u. a. 2008] nieder. Im Rahmen der Vorlesungen „Betriebssysteme für Embedded Computing“ am HPI und „Programming Embedded Systems“ an der BTH, Schweden und zahlreichen weiteren Lehraktivitäten haben sich die in dieser Arbeit entwickelten Techniken im DCL bewiesen.

6.1.1 Die Architektur des Distributed Control Lab

Das DCL besitzt eine erweiterbare Architektur, mit der Steuerungsexperimente über das Internet durchgeführt werden können. Unter einem Experiment wird dabei die Ausführung eines Programms zur Steuerung eines physikalischen Prozesses verstanden. Das Steuerungsprogramm wird auf einem Rechner der Labor-Infrastruktur installiert und ausgeführt und kann dort durch die Verarbeitung von Eingabedaten einer Experiment-Hardware und der Erzeugung von Steuersignalen über entsprechende Schnittstellen einen physikalischen Prozess kontrollieren.
Abbildung 6.1 zeigt einen Überblick der DCL-Architektur. Über verschiedene Nutzerschnittstellen, die auf der rechten Seite dargestellt sind, können über eine *Web-Service*-Schnittstelle (über SOAP [Ryman 2001]) verfügbare Experimente angezeigt, Steuerungsprogramme an Experimente gesendet und Ergebnisse von Experimentläufen betrachtet werden. Experimente können sich an einer zentralen Verwaltungskomponente (Experimentmanager) anmelden, wobei sie durch die Angabe eines Experimenttyps gruppiert werden können. Die Nutzer wählen beim Absenden einer Steuerung einen Experimenttyp, worauf diese an ein verfügbares physikalisches Experiment dieses Typs übermittelt wird. Für jeden Experimenttyp können multiple physikalische Experimente existieren. Durch den Einsatz von Experimenttypen ist eine Lastverteilung auf viele Experimentinstallationen möglich, was den Einsatz des Labors für größere Gruppen von Schülern oder Studenten ermöglicht. Jede Experimentnutzung wird als Auftrag *(job)* vom Experimentdienst entgegengenommen. Beim konkurrierenden Zugriff mehrerer Nutzer muss der Zugriff auf die Experimente synchronisiert werden, da immer nur ein Auftrag zu einem Zeitpunkt ausgeführt werden kann. Sind alle Experimente eines Typs belegt, wird der Auftrag in eine Warteschlange einsortiert und später

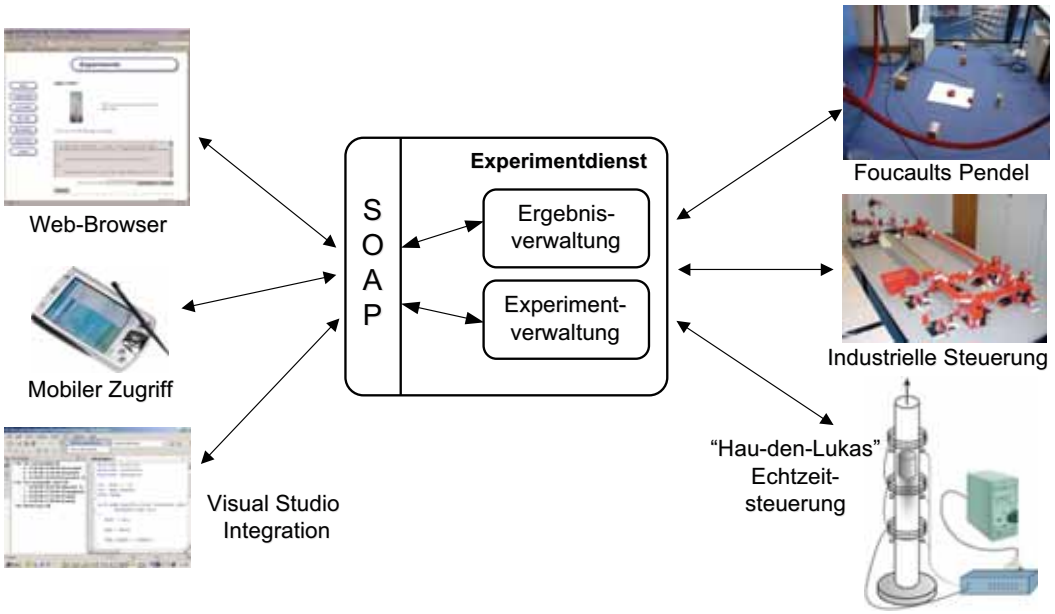


Abbildung 6.1: Architektur des Distributed Control Lab

bei Verfügbarkeit eines Experiments ausgeführt. Über die Web-Schnittstelle kann der Status für jeden Auftrag abgefragt werden und bei erfolgreicher Ausführung (*finished*) können Ergebnisse analysiert werden. Ergebnisse können in Form von Diagrammen, Bildern, Videos, Animationen (z.B. Flash) oder strukturiertem Text in der entsprechenden Endanwendung angezeigt werden.

6.1.2 Experimente im Distributed Control Lab

In das DCL wurden zahlreiche Experimente integriert, von denen nur eine Auswahl im Rahmen dieser Arbeit beschrieben werden soll. Bei der Integration der Lego Mindstorm Roboter musste zunächst die drahtlose Installation von Steuerungsprogrammen und im Rahmen einer Ladestation die Wegfindung und Positionserkennung über Kameras implementiert werden. Später folgten die Experimente „Foucaults Pendel“ und „Hau-den-Lukas“, die im Folgenden ausführlich vorgestellt werden. Eine Modelleisenbahn, das Modell einer Fertigungsstraße mit speicherprogrammierbaren Steuerungen (SPS) [Seitz 2003] der Firma Beckhoff sowie zahlreiche Simulatoren der existierenden Experimente vervollständigte eine umfangreiche Auswahl an Experimenten.

Foucaults Pendel

Dieses Experiment ist angelehnt an einen berühmten Versuch von Leon Foucault, der 1848 im Pantheon in Paris zur Messung der Erdrotation durchgeführt wurde. Eine Kugel an einer langen Schnur wurde in Pendelbewegungen versetzt und nachgewiesen, dass sich die Pendelebene über die Zeit verändert. Die Vielzahl von Installationen auf der Welt machte dieses Experiment zum idealen Kandidaten für das DCL. Eine Aufgabe bei diesem Experiment ist der Ausgleich von Reibungsverlusten durch stete Energiezufuhr. In der Installation am Hasso-Plattner-Institut kann durch einen Elektromagneten im Zentrum der Pendelschwingungsbahn durch elektromagnetische Wechselwirkung das Pendel beschleunigt oder verzögert werden. Zwei orthogonale la-

serbasierte Lichtschranken ermöglichen die Bestimmung der Kugelposition. Die Aufgabe bei diesem Experiment besteht in der Aktivierung des Elektromagneten unter Ausnutzung der ermittelten Positionsinformationen, um z.B. eine bestimmte Amplitude mit minimalem Energieaufwand zu erreichen.
Die Steuerung des Pendels erfolgt u.a. über eine ereignisbasierte Schnittstelle durch eine CLI-Anwendung, die auf einem Steuerungs-PC ausgeführt wird. Die Experiment-Hardware ist über den *Universal Serial Bus (USB)* an den PC angeschlossen. Die Lichtschranken werden über eine separate Schaltung synchron mit einer Frequenz von 23,4 kHz abgetastet und der ermittelte Wert in einem *dual-ported FIFO (DPFIFO)*, einem Speicher mit zwei unabhängigen Transferschnittstellen, gespeichert.

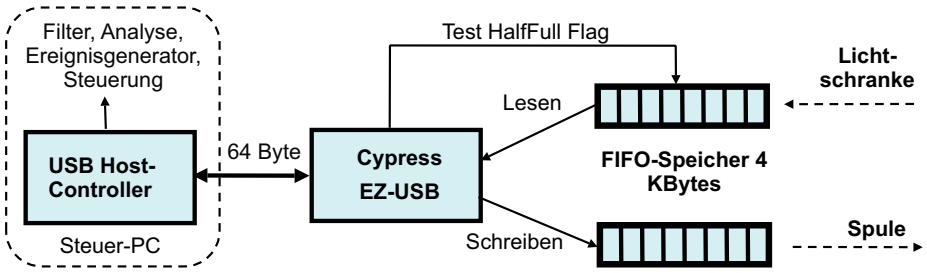


Abbildung 6.2: Architektur der Pendelsteuerung

Ein USB-*Controller-Chip* wertet Flaggen (*full,empty,half-full*) des DPFIFO aus und überträgt nach Aufforderung des *Host-PC* die gewonnenen Daten in 64-Byte-Paketen. Die Ansteuerung der Spule wird in entgegengesetzter Richtung realisiert. Ein Datenstrom wird vom PC über den USB-*Controller-Chip* in den DPFIFO kopiert und eine Spannung an der Spule angelegt, wenn ein aktuelles Datum des Datenstroms einen von 0 verschiedenen Wert besitzt. Die Arbeitsweise der Pendelsteuerung ist in Abbildung 6.2 dargestellt. Auf dem Steuerungs-PC wird *Windows 2000* als Betriebssystem ausgeführt. Daten von der USB-Schnittstelle werden durch einen entsprechenden Treiber als *I/O-Request*-Pakete verarbeitet und über das Ein-/Ausgabe-System Anwendungen zur Verfügung gestellt. Bei der CLI-basierten Steuerung werden die Datenströme über den Aufruf nativer Funktionen (*Plattform Invoke*) in Form der Betriebssystemrufe *ReadFile* und *WriteFile* ausgelesen bzw. geschrieben. Der eingehende Datenstrom wird in einem separaten Thread analysiert mit einem FIR-Filter geglättet, um „flackerndes“ Auslösen der Lichtschranke zu verhindern. Ereignisse bei Änderungen im Datenstrom werden über den Aufruf der Methode (*GetEvent*) der Pendelschnittstelle realisiert, die beim Auftreten eines neuen Ereignisses ein Ereignisobjekt erzeugt. Ein Steuerungsprogramm kann damit auf Lichtschrankenereignisse reagieren und entsprechende Steuersignale über einen Methodenaufruf der Pendelschnittstelle (*SendEvent*) in den ausgehenden Datenstrom installieren.
Der Einsatz hochpriorer Threads (*real-time priority*) und des FIFO-Puffers mit einer Größe von 4 KByte ermöglicht die Einhaltung der vom Experiment gegebenen Zeitschranken, obwohl das eingesetzte Betriebssystem von sich aus keine Echtzeitgarantien erlaubt. Die Anzahl der Steuersignale pro Zeiteinheit musste zusätzlich begrenzt werden, um ein Leerlaufen der Puffer zu verhindern, was zu einem undefinierten Zustand der Hardware führen kann. Hierfür wurden Ring-Puffer eingesetzt, die für einen vorgegebenen Zeitraum der Vergangenheit abgesetzte Steuersignal protokollieren und über die ggf. die Ablehnung von Steuersignalen (*admission control*) erfolgt.

Durch die vorgestellte Treiberarchitektur ist es dem Nutzer dieses Laborexperiments möglich, sich voll auf die Programmierung effizienter Steuerungsalgorithmen zu konzentrieren, da die Experimentschnittstelle in das Programmiermodell der CLI integriert wurde.

Hau den Lukas

Das „Hau den Lukas“-Experiment besteht aus einer Glasröhre, umrandet von 7 äquidistanten Elektromagneten und Lichtschranken. Die Aufgabe bei diesem Experiment besteht darin, einen kleinen Eisenzylinder durch die Aktivierung der Elektromagneten und Auswertung der Lichtschranken zu beschleunigen. Der Aufbau der Steuerung ist der des Pendels sehr ähnlich. Es kommt wieder ein *DPFIFO* zum Einsatz, dessen Daten von einem Steuer-PC über die parallele Schnittstelle verarbeitet werden. Im Gegensatz zum Pendel beträgt die Abtastfrequenz der Analog-/Digitalwandler 38 kHz und die maximale Puffergröße ist auf 256 Byte beschränkt. Die Spulen werden über eine spezielle Hardware über eine Autobatterie aktiviert, die ein stärkeres Magnetfeld aufbauen kann. Aufgrund der hohen erreichten Geschwindigkeiten des Projektils sind den Steuerungsprogrammen harte Zeitschranken für die Reaktion auf Lichtschrankenereignisse gegeben. Der eingesetzte Puffer hat den Vorteil, die Zeitschranke für die Verarbeitung der Pufferdaten, bevor der Puffer voll- bzw. leerläuft, zu erhöhen. Im gleichen Moment entsteht aber eine Verzögerung für die Kontrollsignale der Steuerung, denn je größer der eingesetzte Puffer, desto mehr „Einträge“ werden vor einer notwendigen Reaktion an die Spulen gesendet. Um die Reaktionszeit zu verkürzen, kann beim „Hau den Lukas“-Experiment die Puffergröße konfiguriert werden. Je kleiner dabei der Puffer ist, desto kürzer ist die Reaktionszeit und desto kürzer werden auch die Zeitschranken für die Datenverarbeitung. Bei maximalen Geschwindigkeiten müssen bei diesem Experiment Reaktionen unterhalb des Millisekunden-Bereichs erzielt werden. Für die Ausführung der Steuerungsprogramme wird deshalb der Einsatz von Echtzeitbetriebssystemen erforderlich. Im Rahmen der Lehrveranstaltung „Betriebssysteme für Embedded Computing“ wurden Real-Time-Linux, Windows Ce.NET und QNX-Neutrino für die Steuerung eingesetzt.
Um die Programmierung der Steuerung auf Basis der CLI zu ermöglichen, wurde vom Autor dieser Arbeit das Real-Time.Net-Projekt begründet. Real-Time.Net ermöglicht die vorhersagbare Ausführung von CLI-Programmen und basiert auf dem Lego.Net-*Compiler*, einem CIL-*Frontend* für die GNU Compiler Collection (gcc). Real-Time.Net wird in einem späteren Abschnitt (6.3.1) kurz vorgestellt.

Fischertechnik Fertigungsstraße

Im Rahmen eines vom Autor mitbetreuten Bachelorprojekts wurde eine Windows-Ce.NET-basierte SPS in ein Fischertechnik-Modell einer Fertigungsstraße integriert. Neben der Steuerung von pneumatischen Aktuatoren, Förderbändern, rotierbaren Segmenten und Näherungssensoren über die SPS (nach IEC 61131) kann bei diesem Experiment ein Roboterarm über eine Java-Bibliothek programmiert werden. Neben der Integration in das DCL, die an dieser Stelle nicht näher ausgeführt werden soll, stellt dieses Experiment ein interessantes Szenario für den Einsatz des vorgestellten Konfigurationsmanagers, für die Verwaltung einer heterogenen komponentenbasierten Steuerungs- und Monitoring-Anwendung dar.

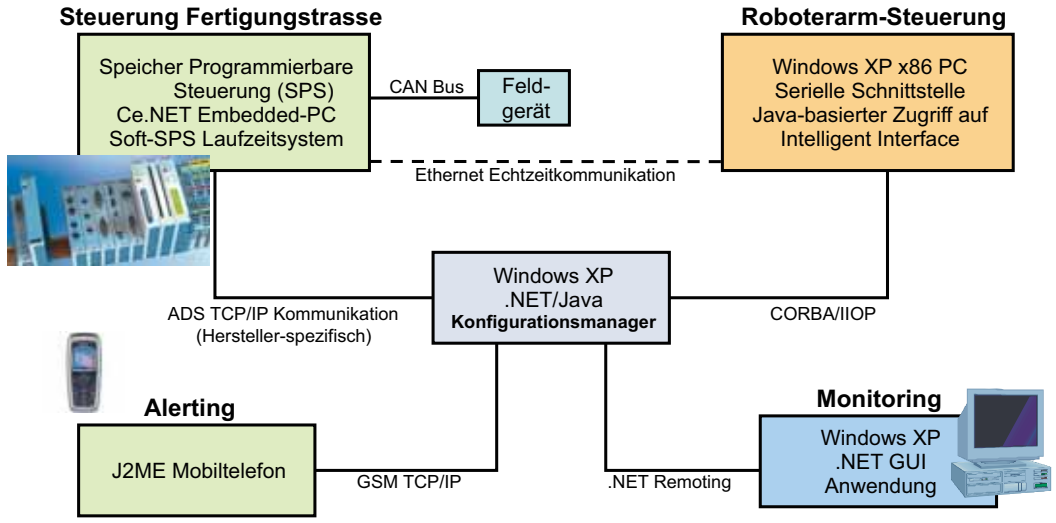


Abbildung 6.3: Heterogene Steuerungsanwendung einer industriellen Anlage

Abbildung 6.3 veranschaulicht die Konfiguration einiger beteiligter Softwarekomponenten. Neben der Erzeugung entsprechender Steuerungskomponenten auf der SPS kann über die entwickelten Kapselkonfiguratoren die Java-basierte Steuerung des Roboterarms sowie die Überwachung der Anlagensteuerung über graphische .NET-Anwendungen oder mobile Geräte integriert werden. Die Fischertechnik-Fertigungstrasse stellt ein interessantes Szenario für die Anwendung der in dieser Arbeit vorgestellten Konfigurationstechniken dar, im Folgenden soll aber die Realisierung von Sicherheit und Fehlertoleranz im DCL an Hand der Experimente „Foucaults Pendel“ und „Hau den Lukas“ diskutiert werden.

6.2 Sicherheit und Fehlertoleranz durch Adaption

Eine Problematik beim Anbieten von Steuerungsexperimenten über das Internet sind fehlerhafte und bösartige Programme, die potentiell teure Experiment-Hardware beschädigen können. Das Fehlverhalten fremder Nutzerprogramme kann zunächst grob in zwei Kategorien eingeteilt werden. Zum einen kann der Rechner, auf dem das Programm ausgeführt wird, kompromittiert werden, zum anderen kann die Experiment-Hardware durch fehlerhafte Steuersignale in einen unerlaubten Zustand überführt werden. Ersteres umfasst z.B. das Löschen von Dateien, den Aufbau von Netzwerkverbindungen zu internen Rechnern oder die unerlaubte Benutzung eines Druckers. Beispiele für Letzteres sind das Abbrennen einer Spule beim „Hau den Lukas“-Experiment durch zu langes Aktivieren oder das Abstoppen des Pendels, wodurch das Experiment für weitere Nutzer nicht mehr zur Verfügung steht.
Der **Schutz vor Kompromittierung** der Ausführungsrechner wurde im Rahmen des DCL durch verschiedene Techniken realisiert, die nicht im Mittelpunkt dieser Arbeit stehen. Der Technische Report „Sichere Ausführung nicht vertrauenswürdiger Programme“ [Nicolai 2005] gibt einen Überblick über anwendbare Techniken. Hervorzuheben ist, dass für die sichere Ausführung von CLI-Steuerungsprogrammen beim Pendelexperiment vom Autor *Code Access Security (CAS)* verwendet wurde, mit der sehr feingranular die Berechtigungen von CLI-Anwendungen konfiguriert werden können. Durch die Einschränkung der Nutzung bestimmter Typen der *Base Class Library* kann

der Zugriff auf nicht erlaubte Betriebsmittel unterbunden werden. Bei der Integration der Lego Mindstorm Roboter in die Laborinfrastruktur wurden zusätzlich die statische Code-Analyse, experimentspezifische Sprachen und die Simulation der Aufträge vor der Ausführung auf dem physikalischen Experiment eingesetzt.
Die **Sicherung der Experiment-Hardware** konnte mit Hilfe der in dieser Arbeit vorgestellten Adaptionstechniken, durch die Beobachtung dynamischer Umgebungseigenschaften und der dynamischen Rekonfiguration vor dem Verlassen eines erlaubten Systemzustands realisiert werden. Bei den beschriebenen Techniken wurde zunächst von der Fehlerfreiheit der involvierten Kapseln ausgegangen. Für die Sicherung der Experiment-Hardware muss deshalb an dieser Stelle das Ausführungsmodell erweitert und ein Fehlermodell eingeführt werden. Dies soll im Rahmen der Betrachtungen zur Behandlung des Zustands von Kapseln bei der dynamischen Rekonfiguration diskutiert werden, da dieser beim Ausfall von Kapseln verloren gehen kann.

6.2.1 Dynamische Rekonfiguration und Zustand

Abbildung 6.4 gibt einen Überblick über mögliche Strategien der Zustandsbehandlung bei der dynamischen Rekonfiguration. Im unteren Bereich sind im Rahmen der Realisierung adaptiver Anwendungen vorgestellte Strategien aufgelistet, während im oberen Bereich der Ausfall von Kapseln eingeschlossen wird. Beim Erzeugen neuer Kapseln bei der adaptiven Rekonfiguration erfolgt kein Zustandstransfer, da in diesem Fall kein gültiger Ausgangszustand vorhanden ist. Die neue Kapsel initialisiert ihren Zustand während der Startphase oder durch die Verarbeitung neuer Eingaben. Vor allem im Bereich der Steuerungssysteme, bei denen eine zyklische Verarbeitung von Eingaben erfolgt [Liu 2000], ist dies ein typisches Szenario. Im Falle der Migration und der dynamischen Aktualisierung von Kapseln ist der Transfer von Zustand notwendig. Ansätze hierfür wurden im Rahmen dieser Arbeit vorgestellt. Im Unterschied zur Migration, bei der der Zustand eins zu eins auf einen entfernten Rechner übertragen wird, kann bei einer dynamischen Aktualisierung eine Anpassung des Zustands durch veränderte Typen notwendig sein. Eine in dieser Arbeit nicht betrachtete Variante für die Rekonfiguration ohne Fehlerfall in Steuerungsanwendungen ist die *transitionale Rekonfiguration*, bei der parallel zu einer aktiven Konfiguration neue Kapseln geladen werden, die durch die Verarbeitung der gleichen Daten ihren internen Zustand aufbauen. Nach erfolgter Initialisierung des Zustands kann die Nutzung des Ausgangssignals der alten Konfiguration zu einer neuen Konfiguration umgeschaltet werden.
Für die Betrachtung der Realisierung von Fehlertoleranz soll zunächst ein **Fehlermodell** definiert werden, bei dem zwischen einer ausgefallenen und einer fehlerhaften Kapsel unterschieden werden soll. Eine **fehlerhafte Kapsel** weicht in ihrem Verhalten von ihrer Spezifikation ab. Im Falle von Steuerungssystemen bedeutet dies, dass fehlerhafte Ausgangssignale produziert werden, die das gesteuerte System in einen ungültigen Zustand und damit zum Ausfall bringen können. Die Feststellung fehlerhafter Kapseln kann durch die Überwachung der Aktivitäten oder durch die Beobachtung des Zustands des gesteuerten Systems erfolgen. Im Rahmen dieser Arbeit wird der **Ausfall einer Kapsel** mit der Terminierung eines umgebenden Betriebssystemprozesses diagnostiziert. Ein typisches Beispiel hierfür sind unbehandelte strukturierte Ausnahmen, welche die virtuelle Maschine zur Terminierung des Betriebssystemprozesses veranlassen. Durch die Überwachung der entsprechenden Betriebssystemobjekte kann der Ausfall von Kapseln detektiert werden. Dies wird durch den Kapselkonfigurator *Separate-Process* implementiert. Im Falle verteilter Anwendungen kann der Ausfall

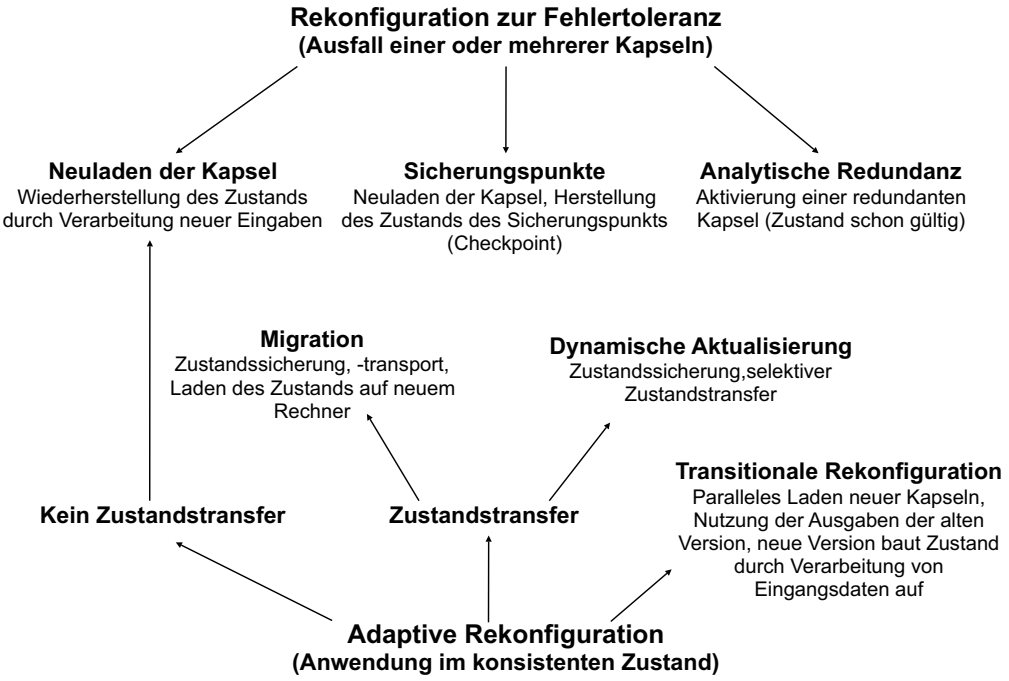


Abbildung 6.4: Dynamische Rekonfiguration und Zustand

von Verbindungen oder beteiligter Rechner (zwischen denen nicht unterschieden wird) (*crash fault*) auf den Ausfall von Kapseln übertragen werden. Vom primären Konfigurationsmanager aus betrachtet zählt eine **entfernte Kapsel als ausgefallen**, wenn der zugeordnete Rechner nicht erreichbar ist. Die im Rahmen des Adapt.Net-Projekts entwickelten Standard-*Observer* der *Monitoring*-Infrastruktur unterstützen die Diagnose ausgefallener Rechner bzw. Verbindungen, mit Hilfe von *ICMP-Ping*-Paketen.

Beim Ausfall und der Diagnose des Fehlverhaltens einer Kapsel kann, um die Zuverlässigkeit der Anwendungen zu erhöhen, durch entsprechende Fehlertoleranzmaßnahmen reagiert werden. Eine einfache Maßnahme ist das Neuladen ausgefallener Kapseln sowie die Terminierung fehlerhafter Kapseln und gleichzeitigem Neuladen. Der neuen Kapsel muss es möglich sein, einen gültigen Zustand zu etablieren, wie dies bei Steuerungsanwendungen oft durch die Verarbeitung neuer Eingaben in einem nächsten Zyklus erfolgen kann. Eine zweite Möglichkeit stellt die Speicherung des Zustands einer Kapsel während ihrer Laufzeit dar. Die gewonnenen Sicherungspunkte (*checkpoints*) repräsentieren einen gültigen Zustand und können bei einem Ausfall in der neu erzeugten Kapsel installiert werden. Die für das Anlegen der Sicherungspunkte benötigten Ressourcen (Speicher, Rechenzeit) sind ein Nachteil dieses Verfahrens. Der gesteuerte physikalische Prozess ändert sich potentiell sehr schnell und bei der Wiederherstellung eines Sicherungspunktes kann der Zustand deshalb veraltet sein. Die Frequenz der Speicherung der Sicherungspunkte muss entsprechend angepasst werden, was die Zusatzkosten erhöht. Eine weitere Möglichkeit Ausfälle von Rechnern zu tolerieren ist der Einsatz von Replikation. Kapseln werden dabei parallel auf verschiedenen Rechnern ausgeführt und der Zustand synchronisiert. Beim Ausfall eines Rechners ist der Zustand der Kapsel im replizierten Exemplar verfügbar. Für die Reaktion auf fehlerhafte und bösartige Algorithmen ist dieses Verfahren aber nicht geeignet, da die Fehler innerhalb der Kapsel auf beiden Rechner zum Ausfall führen können und damit der

Zustand verloren gehen könnte.
Eine leistungsfähige Möglichkeit, die beim „Foucault'schen Pendel“ verwendet wurde, ist der Einsatz **analytischer Redundanz**. Dieses Verfahren wurde im Rahmen der Simplex-Architektur [Sha 1998, 2001] unter der Leitung von Lui Sha vorgestellt. Eine Steuerung S_1 ist analytisch redundant zu einer Steuerung S_2, wenn sich diese bezüglich eines Qualitätsmaßes (z.B. Zuverlässigkeit) zur ersteren unterscheidet, aber trotzdem die minimalen Anforderungen an die Steuerungsaufgabe erfüllt. Ein Beispiel für die Anwendung dieses Verfahrens ist die Steuerung der Boeing 777, die eine komplexe Steuerung besitzt. Bei deren Ausfall kann die Steuerung von der der Boeing 747 übernommen werden, die eine geringere Komplexität besitzt, da nur ein Teil der Steuersignale verarbeitet wird und deren Algorithmen sich durch einen langjährigen Einsatz als zuverlässig erwiesen haben [Sha u. a. 1998].

6.2.2 Analytische Redundanz

Im Rahmen des Pendelexperiments konnte gezeigt werden, dass das Architekturmuster der analytischen Redundanz (Abschnitt 5.5.7) mit Hilfe der im Rahmen dieser Arbeit vorgestellten Adaptionstechniken realisiert werden kann.
Beim Pendelexperiment werden Steuerungsalgorithmen, die vom Nutzer über die Web-Schnittstelle gesendet werden, als Kapseln einer Steuerungsanwendung ausgeführt. Die Nutzersteuerungskapsel wird über eine entsprechende Schnittstelle, mit Methoden zur Verarbeitung der Lichtschrankenereignisse und einer Methode zur (De-)Aktivierung der Spule, mit einer Hardware-Zugriffskapsel verbunden, welche die zuvor beschriebene Analyse und Verarbeitung der strombasierten Experimentdaten realisiert. Die Nutzersteuerungskapseln werden als separater Betriebssystemprozess gestartet. Der Ausfall dieser Kapseln kann damit über den Kapselkonfigurator wie beschrieben implementiert werden. Fehlerhafte Nutzersteuerungskapseln werden durch die Beobachtung bestimmter Umgebungsparameter mit Hilfe der implementierten *Monitoring*-Infrastruktur erkannt. Beim Pendelexperiment werden die folgenden Parameter überwacht:

- die Amplitude des Pendels (Fällt die Amplitude des Pendels unter einen kritischen Wert, kann es zum Abstoppen kommen - da keine Lichtschrankenereignisse durch vollständige Verdeckung der Laser gewonnen werden können - und wäre damit für weitere Nutzer nicht mehr verfügbar. Zu große Amplituden können zur Beschädigung in der Nähe befindlicher Gegenstände führen.)

- die Laufzeit der Steuerung (Für die Realisierung von Fairness für mehrere Nutzer muss die Laufzeit einzelner Experimentläufe beschränkt werden.)

- die verbrauchte Energie (Für bestimmte Aufgaben steht dem Nutzer eine begrenzte Energie für die Aktivierung der Spule zu Verfügung. Diese wird über die Aktivierungsdauer der Spule für jede Experimentnutzung gemessen. Zusätzlich ist die aufgewendete Energie ein Indikator für die Temperatur der Spule, die vor allem beim „Hau den Lukas“-Experiment überwacht werden muss.)

Über entsprechende *Observer* werden die beschriebenen Umgebungsparameter überwacht. Für das Pendel wurde zunächst eine sichere Steuerung (*safety controller*) entwickelt, deren Funktionalität durch intensive Tests überprüft wurde. Die sichere Steuerung wird ausgeführt, wenn keine Nutzersteuerung verfügbar ist oder wenn durch ein

Verlassen des sicheren Bereichs der überwachten Experimentparameter eine Rekonfiguration ausgelöst wurde. Die sichere Steuerung ist in der Lage, eine wohldefinierte Amplitude des Pendels einzustellen und damit einen vergleichbaren Ausgangszustand vor jeder Experimentnutzung herzustellen.

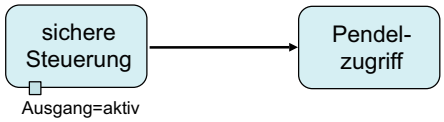


Abbildung 6.5: Konfiguration mit sicherer Steuerung

Abbildung 6.5 zeigt die Konfiguration der Steuerungsanwendung für das Pendelexperiment, wenn keine Nutzersteuerung verfügbar ist. Die sichere Steuerung ist unmittelbar mit der Hardware-Zugriffskapsel verbunden und realisiert direkt die Steuerung. Über einen zusätzlichen Kapselparameter (nicht dargestellt) kann die Amplitude, die durch die Steuerung eingestellt werden soll, reguliert werden. Über diesen Parameter konnte ein Nachtmodus für das Pendel integriert werden, bei dem die Amplitude des Pendels zum Energiesparen in der Nacht reduziert wird. Über einen Uhrzeit-*Observer* kann die aktuelle Stunde für den Nachtmodus als Adaptionsparameter dienen. Für die reduzierte Amplitude wurde eine separate Konfiguration mit einem entsprechenden Wert für den Parameter erstellt.

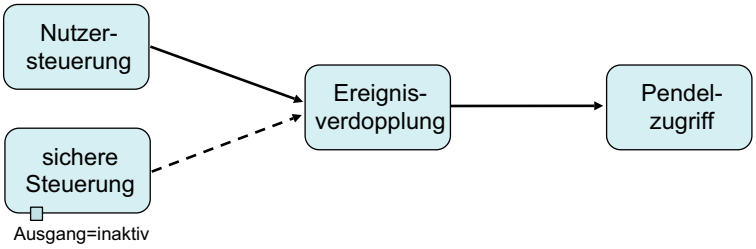


Abbildung 6.6: Konfiguration der Steuerung mit Nutzerkomponente

Abbildung 6.6 zeigt die Konfiguration der Steuerungsanwendung mit aktiver Nutzersteuerung. Die Nutzersteuerungskapsel ist über eine zusätzliche Kapsel mit der Hardware-Zugriffskapsel verbunden und kann auf diese Weise Lichtschrankenereignisse verarbeiten und den Magneten steuern. Parallel zur Nutzersteuerung wird die analytisch redundante sichere Steuerung ausgeführt. Über einen Kapselparameter (*Ausgang*) der sicheren Steuerung können die Steuerungssignale dieser Kapsel deaktiviert werden. Damit verarbeitet die sichere Steuerung nur Eingangssignale und aktualisiert damit den internen Zustand über die aktuelle Position des Pendels. Um den Hardware-Zugriff für zwei parallele Steuerungskapseln realisieren zu können, wurde eine zusätzliche Kapsel integriert, welche die Lichtschrankenereignisse für mehrere Kapseln repliziert.
Wird bei der Ausführung der Nutzersteuerung der Ausfall oder das Fehlverhalten der Nutzersteuerungskapsel erkannt, wird die dynamische Rekonfiguration der Steuerungsanwendung ausgelöst. Als Zielkonfiguration wird dabei die in Abbildung 6.5 dargestellte Konfiguration aktiviert. Da die sichere Steuerung während der Ausführung der Nutzersteuerung ihren internen Zustand aktualisiert, kann sie sofort nach erfolgter Rekonfiguration gültige Steuerungssignale produzieren. Würde die sichere Steuerung

erst nach dem Ausfall der Nutzersteuerung neu geladen werden, müsste sie zunächst die Position des Pendels über mindestens eine Schwingungsdauer ermitteln. Bei der Betrachtung von Kapselausfällen muss zusätzlich beachtet werden, dass das zeitaufwendige Erzeugen neuer Kapseln nicht vor der eigentlichen Rekonfiguration erfolgen kann, da im Prinzip sofort nach dem Ausfall der Kapsel die Unterbrechungsphase beginnt. Deshalb hat die vorgestellte Konfiguration den großen Vorteil, dass die sichere Steuerung nicht neu geladen werden muss.
Die beschriebene adaptive Steuerungsanwendung wurde vollständig mit Hilfe des vorgestellten graphischen Werkzeugs entwickelt und auf der implementierten Plattform ausgeführt. Das Pendelexperiment wurde erfolgreich in verschiedenen Lehrveranstaltungen am Hasso-Plattner-Institut und im Rahmen eines Forschungsaufenthalts des Autors in einer Lehrveranstaltung an der Blekinge Techniska Högskola in Ronneby, Schweden eingesetzt. Vor allem die Überwachung der Pendel-Amplitude und die Verhinderung des Abstoppens des Pendels haben sich beim entfernten Einsatz und beim Betrieb außerhalb der Werktage als nützlich erwiesen.

6.3 Ausblick: Vorhersagbare Dynamische Rekonfiguration

Im Rahmen eines Ausblicks soll die Anwendbarkeit der entwickelten Rekonfigurationsalgorithmen für die Einsetzbarkeit in Echtzeitsystemen analysiert werden. Hierfür wird zunächst ein erweitertes Laufzeitmodell definiert.
In einem Echtzeitsystem werden eine Reihe von Tasks ausgeführt, welche die Verarbeitung von Eingangs- und die Erzeugung von Ausgangssignalen realisieren. Den Tasks können durch physikalische Gegebenheiten feste Zeitschranken zugeordnet werden, die bestimmen, wann die Abarbeitung einer Task beendet sein muss. Die Verarbeitung erfolgt zyklisch [Liu und Layland 1973]. In einer Periode werden Daten von Sensoren eingelesen, verarbeitet und Aktuatoren eingestellt. Die Dauer einer Periode (T) wird wiederum durch die physikalische Umgebung bestimmt. Zusätzlich muss für jede Task eine Ausführungsdauer bekannt sein, welche die maximale Abarbeitungsdauer (C) einer Task innerhalb einer Periode bestimmt.
Aufgabe des Echtzeit-Scheduling ist die Ausführung einer Menge von Tasks unter Einhaltung der Zeitschranken für gegebene Perioden- und Ausführungsdauern. Für das Scheduling von Tasks werden oft Prioritäten verwendet, welche die Reihenfolge der Taskausführung bestimmen. Diese Verfahren werden als prioritätsbasiertes Scheduling bezeichnet. Es wird zwischen statischen mit festen Prioritäten und dynamischen Scheduling-Verfahren mit veränderbaren Prioritäten unterschieden. Tasks werden in dieser Arbeit als unterbrechbar betrachtet. Im Rahmen einer Ausführbarkeitsanalyse (*schedulability analysis*) kann bestimmt werden, ob eine gegebene Konfiguration auf einem System ausführbar ist und dabei die Zeitschranken aller Tasks eingehalten werden. Für statische prioritätsbasierte Scheduling-Verfahren kann im Rahmen einer *Rate Monotonic Analysis (RMA)* [Liu 2000] an Hand der Abarbeitungsdauer, die durch „*Worst Case Execution Time*“-Analyse berechnet werden kann, und der Periodendauer die Ausführbarkeit einer Menge von Tasks berechnet werden, wenn die Zeitschranken gleich der Periode und die Tasks unabhängig sind.
Betrachtet man die in dieser Arbeit vorgestellten Rekonfigurationsalgorithmen, basieren diese im Wesentlichen auf der Synchronisation von mehreren Anwendungs- und einer Rekonfigurationstask. Je nach verwendetem Algorithmus kommen unterschiedliche Synchronisationsprimitive zum Einsatz, die bei einer Ausführbarkeitsanalyse betrach-

tet werden müssen. Angenommen eine Steuerungsanwendung, deren Tasks nach dem statischen Scheduling-Verfahren *rate monotic scheduling (RMS)* ausgeführt werden, soll dynamisch rekonfiguriert werden, dann können die Kommunikationstransaktionen der Anwendung mit der Rekonfigurationstask durch zwei Mutexe sychronisiert (siehe Abschnitt 3.6.3) werden. Bei RMS werden die Prioritäten einer Task invers zu ihrer Periode bestimmt, das heißt Tasks mit einer langen Periodendauer besitzen eine niedrige Priorität. Die Rekonfigurationstask kann als normale zyklische Aktivität in die Menge der Tasks der Anwendung integriert werden und besitzt eine Ausführungszeit (C_R) und eine Periodendauer (T_R). Die Rekonfiguration muss dabei nicht ständig stattfinden, sondern nur wenn nötig. Ansonsten verfällt der eingeplante Zeitschlitz für die Rekonfigurationstask. Es sei zunächst angenommen, dass die Rekonfiguration keine Änderung der Taskmenge verursacht und die Anwendungstasks unabhängig voneinander sind. Es werden nur Konfigurationen mit einem beteiligten Rechner betrachtet. Beim Einsatz von RMS bei einer Taskmenge mit Synchronisation (zwischen Anwendungs- und Rekonfigurationstasks) müssen entsprechende Protokolle zur Verhinderung von Prioritätsinvertierungen eingesetzt werden. Nach [Goodenough und Sha 1988] gilt beim Einsatz von *priority ceiling* ein erweitertes Scheduling-Kriterium für die Ausführbarkeit einer Taskmenge mit RMS:

$$\forall i, 1 \leq i \leq n, \frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1) \tag{6.1}$$

Wenn die Rekonfigurationstask aktiv wird, kann unter Einsatz von *priority ceiling* davon ausgegangen werden, dass keine Kommunikationstransaktion aktiv ist. Im Prinzip laufen alle Kommunikationstransaktionen mit der höchsten Systempriorität, da alle Anwendungstasks während einer Kommunikationstransaktion auf die *ceiling*-Priorität des Synchronisationsobjekts angehoben werden. Das heißt, eine Task kann durch die maximale Ausführungsdauer einer niederprioren Task zusätzlich verzögert werden. U_i sei die maximale Dauer einer Kommunikationstransaktion der Task i mit $U_i \leq C_i$. Für den Rekonfigurationstask gilt $U_R = C_R$. Betrachtet man das Synchronisationsverfahren aus Abschnitt 3.6.3 kann eine höherpriore Task im schlechtesten Fall von genau einer Task mit niedriger Priorität verzögert werden, wenn diese kurz vor Aktivierung der hoch-prioren Task ein Mutex akquiriert. Es kann sich dabei entweder um eine Anwendungs- oder die Rekonfigurationstask handeln. Damit ergibt sich die maximale Blockierungsdauer für die Tasks:

$$B_i = \begin{cases} max(U_{i+1}, .., U_n) & \text{für } i < n \\ 0 & \text{für } i = n \end{cases} \text{ mit } T_i < T_j, \text{ wenn } i < j. \tag{6.2}$$

Betrachtet man das Synchronisationsverfahren mit *ReaderWriterLocks* aus Abschnitt 3.6.3, ist interessant zu bemerken, dass sich in diesem Fall die (*worst case*) Blockierungsdauer erhöht. Akquirieren die Anwendungstasks kaskadierend *Read-Locks*, würde der Rekonfigurationstask beim Anfordern des *Write-Locks* durch die Summe der maximalen Transaktionsdauern aller niederprioren Tasks zusätzlich blockiert. Wird genau nach der Anforderung des *Write-Locks* die höchstpriore Task aktiv, wird diese durch die Summe aller maximalen Transaktionszeiträume aller Tasks und der Rekonfigurationsdauer verzögert. Die Blockierungsdauer beträgt dann in diesem Fall für alle Tasks:

$$B_i = \sum_{j=1}^{j \leq n} U_j \tag{6.3}$$

Mit der Berechnung der Blockierungsdauer der Tasks und dem erweiterten Scheduling-Kriterium ist eine notwendige Bedingung an eine rekonfigurierbare Taskmenge gegeben. Aus den Formeln können einige Schlussfolgerungen gezogen werden. Die maximale Transaktionsdauer einer niederprioren Task darf z.B. niemals größer als die Periodendauer einer höherprioren Task sein. Problematisch ist der Einsatz der vorgestellten Algorithmen vor allem dann, wenn sich die Periodendauern der Tasks stark unterscheiden. Eine Task mit einer Ausführungsdauer von 10 ms und einer Periodendauer von 200 ms verursacht auf einem System nur eine Last von 5 Prozent. Eine zusätzliche Task mit einer Periodendauer von 15 ms würde allein durch die einzurechnende Blockierungsdauer eine Last von rund 67 Prozent (10 ms / 15 ms) verursachen. Für den angestrebten Einsatz beim „Hau den Lukas“-Experiment ist diese Einschränkung kein Problem, da keine großen Unterschiede der Periodendauern der Tasks existieren.

In diesem Abschnitt wurde eine Analyse des „schlechtest möglichen“ Einflusses der Rekonfiguration vorgestellt. Würde die Rekonfiguration bei Verwendung des Algorithmus für zyklische Verbindungsstrukturen zu einem „günstigen“ Zeitpunkt ausgelöst, ist die reale Beeinflussung durch die Rekonfiguration viel geringer. An Hand der vorgestellten Aktivitätszähler jeder Kapsel kann erkannt werden, wann gerade keine Task aktiv ist und damit die Rekonfiguration ohne die Beeinflussung aller Tasks ausgeführt werden kann. Für diesen Fall müsste nur für jede Task eine Blockierungsdauer der Länge der Rekonfiguration eingeplant werden. Im Prinzip hängt die Beeinflussung der Rekonfiguration im Wesentlichen von der Dauer des Wartens auf die Beendigung aktiver Kommunikationstransaktionen ab, die wiederum vom Zeitpunkt des Auslösens der Rekonfiguration bestimmt wird. Muss die Rekonfiguration nicht bis zu einer festen Zeitschranke ausgeführt werden, kann die Unterbrechung durch die Angabe einer maximalen Dauer beschränkt werden. Mit der Möglichkeit des Zurückrollens von Rekonfigurationsaktionen (Abschnitt 3.5.3) kann eine Rekonfiguration abgebrochen werden, wenn die angegebene Maximaldauer überschritten wird. Dieser Ansatz ist vergleichbar mit dem *Non-Blocking Approach* von OSA+ (Abschnitt 8.2.7).

Bis zu diesem Punkt wurden Änderungen der Taskmenge ausgeschlossen. Mit dem Erzeugen neuer Kapseln oder ihrem Entfernen ist aber die Erzeugung und Terminierung von Tasks verbunden. Im Rahmen der „*mode change*“-Protokolle [Sha u. a. 1989] wurden Kriterien für Änderungen einer Taskmenge und der Umsetzung der Änderungen entwickelt, die entsprechend in den Konfigurationsmanager integriert werden können. Beim Einsatz der Algorithmen im DCL sind die Änderungen an der Taskmenge stark eingeschränkt, weshalb „*mode change*“-Protokolle nicht weiter betrachtet werden sollen.

Auch die Ausführung der Rekonfigurationsoperationen erzeugt auf dem System eine zusätzliche Last, die sich aus der Ausführungsdauer geteilt durch die Periodendauer berechnet. Die Dauer einer Rekonfiguration wurde vom Autor dieser Arbeit ausführlich in [Rasche und Polze 2005] analysiert und wird an dieser Stelle nicht genau vorgestellt. Das Einstellen von Parametern und die Konfiguration von Verbindungen ist mit wenigen Zuweisungen in kurzer Zeit realisierbar. Das Erzeugen von Kapseln kann im Prinzip über mehrere Rekonfigurationszyklen verteilt werden und damit parallel zur eigentlichen Anwendung ausgeführt werden. [Rasche und Polze 2005] zeigt, dass die Rekonfigurationsdauer begrenzt ist. Für den Einsatz der Adaptionstechniken beim „Hau den Lukas“-Experiment musste zunächst eine Möglichkeit gefunden werden, CLI-Programme vorhersagbar auszuführen.

6.3.1 Vorhersagbare Ausführung von CLI-Programmen

Im Rahmen des „Hau den Lukas“-Experiments wurde die vorhersagbare Ausführung von CLI-Programmen untersucht. Die Verwendung von typsicheren Hochsprachen wie Java oder C# hat große Vorteile bei der Softwareentwicklung durch die sandboxartige Ausführung der Programme und die Verfügbarkeit umfangreicher Klassenbibliotheken. Problematisch bei der Ausführung der Programme sind aber einige Mechanismen der virtuellen Maschine, welche die Vorhersagbarkeit der Ausführung einschränken. Hierzu zählt die automatische Speicherverwaltung (*garbage collection*), welche die Ausführung von Programmen für die konsistente Analyse aktiver Objekte unvorhersehbar lange unterbricht, und die *just-in-time compilation*, welche die Übersetzung des IL-Code in native Maschineninstruktionen realisiert. Die *Real-Time Spezification for Java (RTSJ)* [Bollella u. a. 2000] definiert Erweiterungen für Java, um Java-Anwendungen vorhersagbar ausführen zu können. Im Rahmen des Real-Time.Net-Projekts wurden einige Techniken der RTSJ übernommen, aber bei einigen Aspekten differenzierte Ansätze für die vorhersagbare Ausführung von CLI-Programmen verwendet. An dieser Stelle sollen kurz einige Besonderheiten des Real-Time.Net-Projekts vorgestellt werden, die es von der RTSJ abgrenzen. Ausführliche Informationen um dieses Projekt finden sich in [von Löwis und Rasche 2006; Richter u. a. 2007].

Im Real-Time.Net-Projekt sollte eine Ausführungsplattform für Steuerungskapseln des „Hau-den-Lukas“-Experiments entwickelt werden. Basierend auf dem am „Fachgebiet für Betriebssysteme und Middleware“ entwickelten Lego.Net-Compiler [von Löwis und Möller 2006], einem *intermediate language front-end* der GNU compiler collection (gcc), konnte die Übersetzung von CLI-Assemblies in native Maschineninstruktionen realisiert werden. Durch den Einsatz von Referenzzählung zur automatischen Speicherverwaltung und der Übersetzung vor der Laufzeit existierte die Grundlage für die vorhersagbare Ausführung von CLI-Programmen. Im Rahmen des Real-Time.Net-Projekts wurden vom Autor Bibliothekserweiterungen für die Entwicklung von echtzeitfähigen Steuerungsexperimenten definiert und eine Laufzeitumgebung für das Echtzeitbetriebssystem Windows CE.NET implementiert. Neben der Unterstützung periodischer Threads, dem Zugriff auf erweiterte Thread-Prioritäten und erweiterter Synchronisationstypen wurde der direkte Speicherzugriff und die Implementierung von Interrupt-Behandlungsroutinen durch CLI-Programme integriert. Bei der Realisierung des direkten Speicherzugriffs kamen im Gegensatz zur RTSJ CLI-Attribute zum Einsatz. Durch die Annotierung von Typattributen mit Speicheradressen konnte der Zugriff auf diese Attribute durch eine von Stefan Richter in seiner Masterarbeit [Richter 2006] implementierte Compiler-Erweiterung direkt auf physikalische Speicherinhalte integriert werden. Zusätzlich können mit Hilfe von *structs* Speicherinhalte strukturiert beschrieben und der Zugriff auf Speicherbereiche realisiert werden. Durch die Entwicklung spezieller CLI-Bibliotheken der Hardwarehersteller mit der Beschreibung ihrer Hardware kann die Portabilität von Steuerungsprogrammen durch die Nutzung der CLI erhöht werden. Die im Rahmen des Real-Time.Net-Projekts entwickelten Werkzeuge wurden erfolgreich für die Steuerung des „Hau den Lukas“-Experiments eingesetzt.

6.4 Zusammenfassung

In diesem Kapitel wurde mit dem Distributed Control Lab (DCL) eine Infrastruktur zur entfernten Ausführung von Steuerungsexperimenten vorgestellt. Im Kontext des DCL konnten die in dieser Arbeit vorgestellten Adaptionstechniken zur kontrollierten Ausführung von Steuerungsanwendungen eingesetzt werden. Durch den Einsatz analytischer Redundanz zur Realisierung von Fehlertoleranz konnten fehlerhafte Steuerungen, die durch Nutzer über das Internet eingegeben werden, während der Laufzeit ersetzt und die Beschädigung von Experiment-Hardware vermieden werden. Die dabei angewendeten Muster und die entwickelte Laufzeitinfrastruktur haben eine hohe Wiederverwendbarkeit für weitere Experimente. Die integrierten Techniken haben sich in zahlreichen Lehrveranstaltungen am Hasso-Plattner-Institut und externen Partnern bewiesen. Vor allem bei der entfernten Ausführung der Experimente im Rahmen der Vorlesung „Programming Embedded Systems“ an der BTH in Ronneby, Schweden hat sich der Einsatz der Adaptionstechniken bewährt, da eine Reinitialisierung der Experimente vor Ort nicht möglich war. Mit dem Real-Time.Net-Projekt wurde ein Grundstein für den Einsatz der entwickelten Techniken für eingebettete Echtzeitsysteme gelegt. Die vorhersagbare Ausführung von Steuerungsalgorithmen in der CLI konnte am „Hau den Lukas“-Experiment erfolgreich demonstriert werden.

7 Fallstudien und Leistungsbewertung

In diesem Kapitel wird die Leistungsbewertung der implementierten Ausführungsplattform für adaptive Anwendungen vorgestellt. Wichtige Maße adaptiver Anwendungen sind dabei die durch die dynamische Rekonfiguration verursachte Unterbrechungsdauer, die potentiell vom Nutzer wahrgenommen werden kann und die Rekonfigurationsdauer selbst. Des Weiteren sind die Zusatzkosten (*overhead*) während der normalen Laufzeitphase, in der keine Rekonfiguration stattfindet, von besonderer Bedeutung, weil diese durch zusätzliche Synchronisationsprimitive verursacht werden.
Während die Leistungsbewertung in einigen Fällen an einfachen Testanwendungen erfolgte, wurden die entwickelten Techniken auch im Rahmen komplexer Fallstudien evaluiert. Hierzu zählt das in Kapitel 6 vorgestellte Distributed Control Lab, in dem die Adapt.Net-Infrastruktur für die sichere Ausführung von Steuerungsexperimenten über das Internet eingesetzt wurde. Des Weiteren wurde die dynamische Aktualisierung von Anwendungen im Rahmen mehrerer Fallstudien evaluiert.
Für die in diesem Kapitel präsentierten Messungen wurde generell der *high resolution performance counter* der *Win32*-Schnittstelle verwendet, der eine Auflösung im unteren Mikrosekundenbereich besitzt. Aufgrund der zu verschiedenen Zeiten während dieser Arbeit vorgenommenen Messungen variieren die verwendeten Testsysteme. Alle Messungen wurden mehrfach durchgeführt. In den Diagrammen werden entweder sämtliche Werte dargestellt oder der Mittelwert und die Standardabweichung angegeben. Sofern nicht anders dargelegt, erfolgten die Messungen auf einem Intel-Xeon-Prozessor (*Single-CPU*) mit 2,8 GHz, 2 GB RAM und Windows XP SP2 als Betriebssystem. Das .NET Framework wurde in der Version 2.0 verwendet.

7.1 Zusatzkosten der Rekonfigurationsalgorithmen

Für die Evaluierung der Zusatzkosten der rekonfigurationsbedingten Synchronisation während der normalen Ausführung wurde die Dauer von Methodenaufrufen mit und ohne verwobene Aspektlogik ausgemessen. Die Zusatzkosten werden durch zusätzliche Instruktionen verursacht, die für die Synchronisation bei der dynamischen Rekonfiguration notwendig sind. Neben den in der Arbeit entwickelten Synchronisationsverfahren mit Hilfe von *ReaderWriterLocks* und Thread-spezifischen Zählern (ReDAC) wurde zum Vergleich die Dauer normaler Methodenaufrufe (über *Interface Calls*), von Reflexionsaufrufen (per *Reflection.Invoke*), Web-Service-Aufrufen (lokal mit Client und Server in separaten Prozessen) und .NET Remoting-Aufrufen (innerhalb eines Prozesses) ermittelt. Während in diesem Abschnitt die direkten die Zusatzkosten für Methodenaufrufe diskutiert werden, wurden in den Fallstudien zur dynamischen Aktualisierung auch die Performanceeinbußen auf der gesamten Anwendungsebene untersucht.
Die nachfolgende Tabelle illustriert die Zusatzkosten der Rekonfigurationsalgorithmen und stellt für jede Messung 100 Messwerte für je 1000 Methodenaufrufe dar. Die Dauer einzelner Methodenaufrufe konnte auf Grund ihrer extrem kurzen Dauer nicht bestimmt werden.

Art des Aufrufs	Aufrufdauer (1000 Aufrufe)
Interface Methodenaufruf	5,96 ± 0,14 μs
Reflection.Invoke-Aufruf	5,05 ± 0,21 ms
.NET-Remoting-Aufruf	346,92 ± 2,11 ms
Web-Service-Aufruf	9,98 ± 0,3 s
ReDAC (statischer Weber)	151,68 ± 2,09 μs
ReDAC (dynamischer Weber)	286,72 ± 5,08 μs
RW-Lock Synchronisation (dynamischer Weber)	469,25 ± 6,29 μs

Die Dauer von 1000 normalen Methodenaufrufen (*interface call*) beläuft sich auf ca. 6 μs. Demgegenüber dauerten 1000 Methodenaufrufe mit aktiver Synchronisation über ein *ReaderWriterLock* im Durchschnitt rund 470 μs. Der neue Algorithmus ReDAC benötigt mit durchschnittlich 287 μs für 1000 Aufrufe deutlich weniger Zeit. Die beiden vorgestellten Messwerte beziehen sich auf die Verwebung mit dem dynamischen Aspektweber Rapier-Loom.Net. Zum Vergleich wurde das Synchronisationsverfahren auch für den statischen Aspektweber Loom.Net umgesetzt. In diesem Fall belaufen sich die Zusatzkosten nur auf 152 μs. Der Aspektweber verursacht u.a. durch die Erzeugung von Proxy-Klassen und einem damit verbundenen zusätzlichen Methodenaufruf die Ausführung zusätzlicher Instruktionen. Im zweiten Teil der Tabelle sind die Messwerte für 1000 Reflexionsaufrufe und 1000 .NET-Remoting-Aufrufe dargestellt. Vergleicht man nun die Zusatzkosten der entwickelten Algorithmen mit denen existierender Lösungen, die zum Teil einen Reflexionsaufruf in die Methodenaufrufe der Anwendung integrieren [Sadjadi u. a. 2004], von denen 1000 durchschnittlich über 5 ms benötigen, sind die Zusatzkosten sogar nahezu vernachlässigbar. Kommunizieren Kapseln mit Hilfe von .NET Remoting, sind die durch den Rekonfigurationsalgorithmus verursachten Zusatzkosten (von 287 μs) gegenüber den rund 347 ms für 1000 .NET-Remoting-Aufrufe kaum von Bedeutung. Verwandte Rekonfigurationsansätze wie [Wegdam 2003] verwenden für die Kommunikation zwischen austauschbaren Anwendungsteilen immer Middleware-basierte Fernaufrufe (ein Vertreter ist .NET Remoting). Mit dem in dieser Arbeit vorgestellten Ansatz können auch innerhalb eines Betriebssystemprozesses über effiziente virtuelle Methodenaufrufe kommunizierende Kapseln unabhängig ausgetauscht werden. Nur wenn Rechnergrenzen überwunden werden müssen, werden aufwendigere Middleware-basierte Fernaufrufe verwendet.

Gegenüber normalen Methodenaufrufen an Kapseln über *interfaces* beträgt die Dauer eines mit dem statischen Aspektweber verknüpften ReDAC-Synchronisationsaspekt rund das 25-Fache, was zunächst recht viel erscheint. Untersuchungen in realen Fallstudien haben aber gezeigt, dass diese Zusatzkosten nur relativ selten auftreten; und zwar genau bei Methodenaufrufen an Wurzelobjekten. Genau hier wird die Leistungsfähigkeit der Kapselabstraktion sichtbar, denn für Methodenaufrufe innerhalb der Kapseln treten die Zusatzkosten durch die Synchronisation nicht auf. Beim später vorgestellten *PaintDotNet* (Abschnitt 7.4.3) ist das Verhältnis zwischen verwobenen und normalen Methodenaufrufen kleiner als 1:7000 - also nur ungefähr jeder siebentausendste Methodenaufruf verursacht die ermittelten Zusatzkosten. Dies wurde im Rahmen einer *Profiling*-Analyse mit Hilfe des Werkzeugs *ANTS-Profiler* bestimmt, ein Bildschirmfoto der Analyse ist in Abbildung 7.1 dargestellt. Zu erkennen ist die Anzahl aller aufgerufenen Methoden in absteigender Wertigkeit. Die Methoden des Rekonfigurationsaspekts (*DSUP.ReconfigurationAspect.InvokeAll()*) wurden nur 7138-mal aufgerufen, während andere Methoden der Anwendung millionenfach aufgerufen wurden.

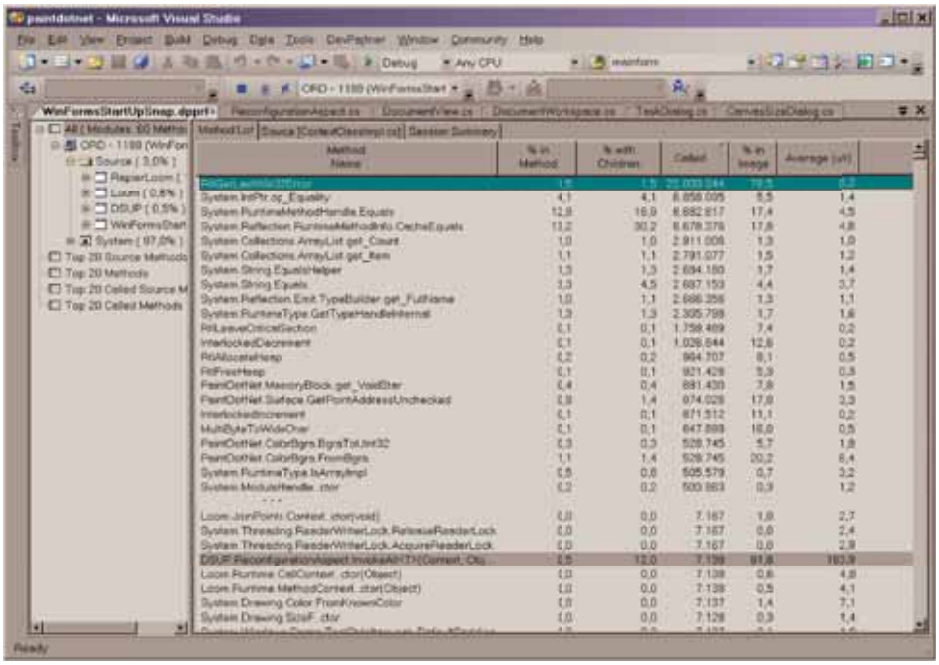


Abbildung 7.1: Anteil von Methodenaufrufen mit Aspektlogik

Abschließend kann festgestellt werden, dass die durch die Synchronisationslogik der Rekonfigurationsalgorithmen verursachten Zusatzkosten im Bereich von 150 ns für jeden Methodenaufruf sehr gering sind. Die in den folgenden Abschnitten (7.4) vorgestellten Fallstudien belegen, dass die Zusatzkosten insgesamt kaum nachweisbar sind.

7.1.1 Unterbrechungsdauer von CLI-Anwendungen

Bei der Herstellung eines rekonfigurierbaren Zustands während einer dynamischen Rekonfiguration werden involvierte Kapseln blockiert und können während dieser Zeit keine Methodenaufrufe von anderen Kapseln bzw. Threads verarbeiten. Dieser Zeitraum kann vom Nutzer als Unterbrechung wahrgenommen werden, da in dieser Zeit potentiell keine Eingaben verarbeitet bzw. Ausgaben erzeugt werden können. Die Minimierung dieses Zeitraums ist deshalb ein wichtiges Leistungsmerkmal dynamischer Rekonfigurationsalgorithmen. Die Unterbrechungsdauer wird auch von der Dauer funktionaler Methodenaufrufe bestimmt, da während der Blockierungsphase auf die Beendigung aktiver Aufrufe gewartet wird. Dieser Anteil der Unterbrechungsdauer ist allerdings stark anwendungsspezifisch und wird im Rahmen der Fallstudien genauer diskutiert. In diesem Abschnitt wird die Evaluation der reinen durch das verwendete Synchronisationsverfahren verursachten Zusatzkosten vorgestellt.
Für die Evaluierung der Unterbrechungsdauer wurde eine Testumgebung implementiert, mit der eine konfigurierbare Anzahl von Kapseln und Threads ausgeführt werden kann. Die Konfiguration der Kapseln wird zufällig bestimmt, indem über eine spezifizierbare maximale Aufruftiefe Methoden der Kapseln (auch rekursiv) aufgerufen werden. Hierdurch ergibt sich durch den Einsatz multipler Threads eine komplexe Aufrufstruktur zwischen den Kapseln, die den Rekonfigurationsalgorithmus vor eine komplexe Synchronisationsaufgabe stellt. Die Unterbrechungsdauer wurde für den Rekonfigurationsalgorithmus ReDAC ermittelt.
Abbildung 7.2 zeigt ein Histogramm der Unterbrechungsdauer von 500 durchgeführten

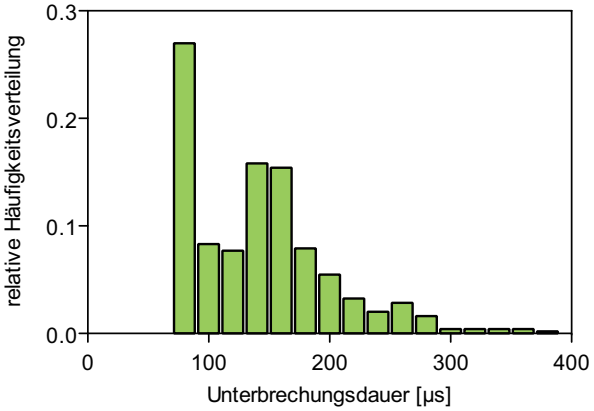


Abbildung 7.2: Unterbrechungsdauer

Rekonfigurationen. Es wurde eine Testkonfiguration mit 20 Kapseln, 10 Threads und einer maximalen Aufruftiefe von 20 ausgeführt. Im Histogramm ist zu erkennen, dass sich die Unterbrechungsdauer in einem Bereich von 80 und 400 μs bewegt. Der größte Teil der Messwerte siedelt sich im Bereich unter 300 μs an. Die gemessenen Werte liegen auf einem sehr niedrigen Niveau und können von einem Nutzer als solche kaum wahrgenommen werden.

7.2 Rekonfiguration heterogener Anwendungen

Die in diesem Abschnitt vorgestellten Messungen wurden auf einem Rechner mit 1.4 GHz Pentium IV Centrino PC mit 1 GB RAM und Windows XP Professional SP2 als Betriebssystem durchgeführt. Die Testanwendung wurde mit dem Microsoft .NET Framework 1.1 und Jacorb 2.5 entwickelt und erstellt. Im Falle heterogener Anwendungen wurden die Verbindungen der Kapseln mit Hilfe von Borlands Janeva 6.0 [Borland 2007] realisiert. Die Messungen wurden zunächst auf einem Rechner durchgeführt.
Die Testanwendung bestand aus zwei Kapseln: einer Nachrichtenquelle und einem Nachrichtenbetrachter, welcher mit einer Periode von 1 Sekunde Nachrichten (in Form einer Zeichenkette) von der Nachrichtenquelle abruft. Die Kodierung der Zeichenkette kann über einen Parameter konfiguriert werden. Zwischen die beiden Kapseln kann eine variable Anzahl von Filterkapseln geschaltet werden, wodurch die Anzahl der beteiligten Kapseln variiert werden kann. Die Messergebnisse für heterogene Anwendungen wurden in Zusammenarbeit mit Marco Puhlmann ermittelt. Das Diagramm in Abbildung 7.3 zeigt die Messung von Rekonfigurations- und Unterbrechungszeit für eine heterogene Anwendung. Bei der dynamischen Rekonfiguration wurde eine neue Kapsel hinzugefügt, welche als CORBA-Objekt geladen wurde. Während der dynamischen Rekonfiguration betrug die Unterbrechungszeit durchschnittlich 10 ms. Auch die Dauer der gesamten Rekonfiguration bewegte sich mit 40-55 ms in einem akzeptablen Bereich. Im Gegensatz zur einer homogenen .NET-Anwendung sind die Messwerte leicht erhöht. Dies ist hauptsächlich durch den Einsatz des Interoperationsframeworks Janeva bedingt, welches durch zusätzlichen Mehraufwand die Interaktionen während der Rekonfiguration verlängert. Zusätzlich wurde die Unterbrechungsphase durch den Verbindungsaufbau einer IIOP-Verbindung verlängert, der mehr Zeit benötigt als der Aufbau einer .NET Remoting oder direkten Verbindung.

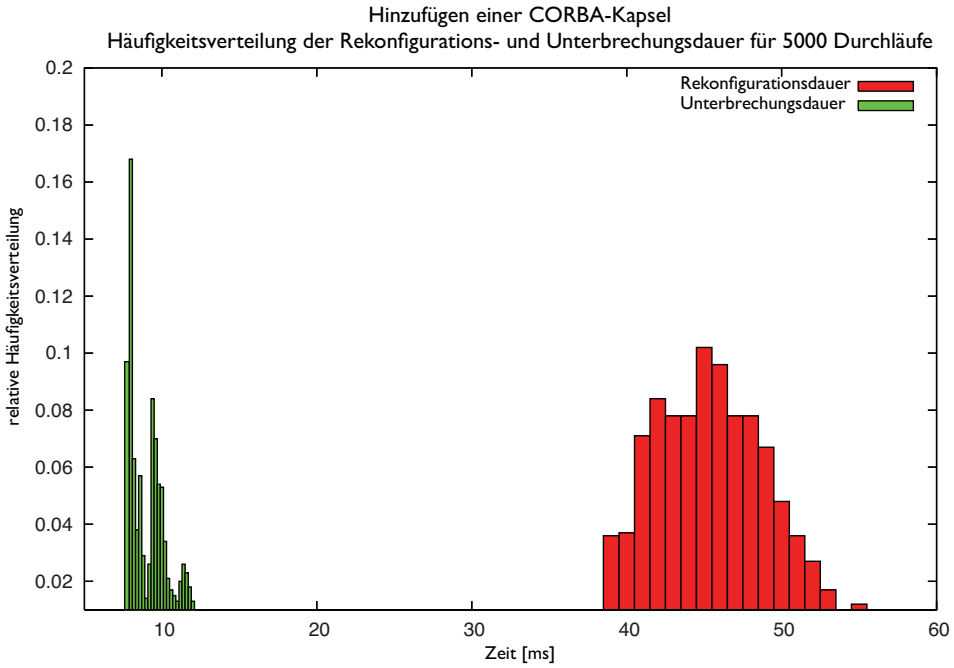


Abbildung 7.3: Rekonfiguration mit CORBA-Kapseln

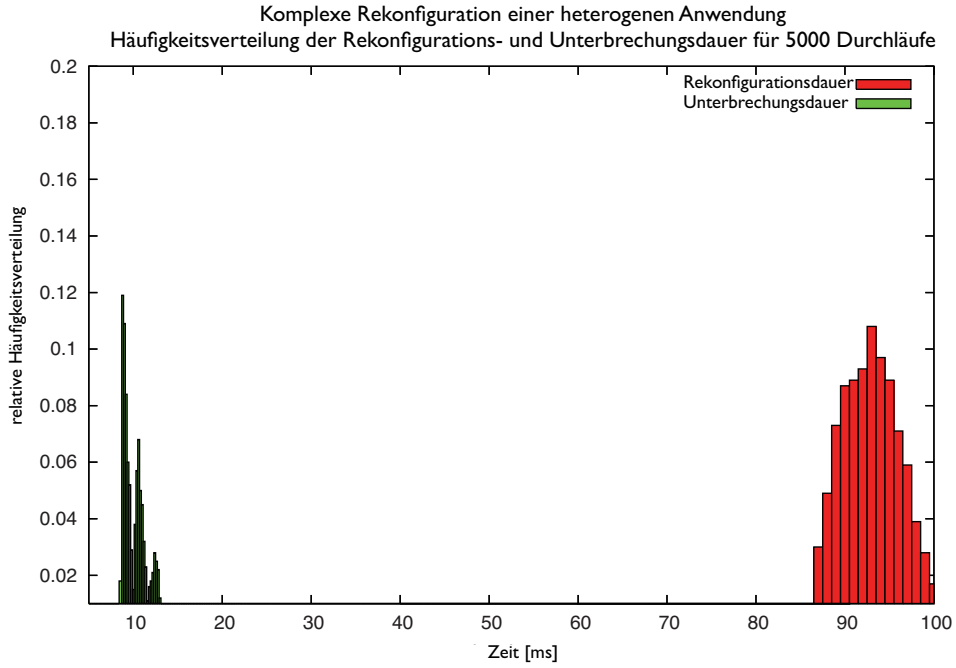


Abbildung 7.4: Rekonfiguration einer heterogenen Anwendung

In einem zweiten Test wurden die Leistungsmaße einer komplexeren dynamischen Rekonfiguration bestimmt. In der beschriebenen Testanwendung wurden die Filter- und Nachrichtenquellkapsel mit CORBA realisiert, während der Nachrichten-Client eine .NET-Kapsel war. Die dynamische Rekonfiguration bestand aus dem Hinzufügen von zwei .NET-Kapseln und einer CORBA-Kapsel, dem Entfernen einer .NET-Kapsel und der Justierung von 15 Kapselparametern. Wie im Diagramm in Abbildung 7.4 illustriert, bewegt sich die Unterbrechungsdauer wiederum im Bereich von 10 ms, während der gesamte Rekonfigurationsprozess 85-100 ms dauerte.

7.3 Fallstudie: Foucaults Pendel

Wie schon im vorangegangenen Kapitel in Abschnitt 6.1.2 vorgestellt, wurde das entwickelte Adaptionsverfahren für die sichere Ausführung von Steuerungsanwendungen im Distributed Control Lab eingesetzt. Bei der Leistungsbewertung sind in diesem Fall vor allem die Reaktionsdauer bei Ausfällen sowie die Rekonfigurationsdauer zu betrachten.

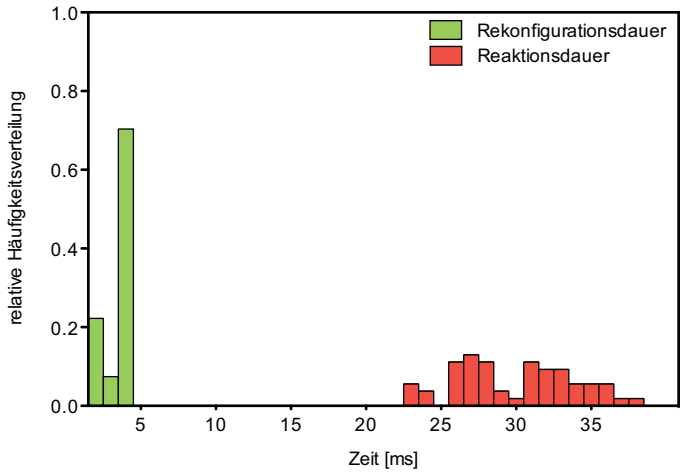


Abbildung 7.5: Foucaults Pendel: Reaktions- und Rekonfigurationsdauer

Abbildung 7.5 zeigt die relative Häufigkeitsverteilung von jeweils 100 ermittelten Messwerten. Die Reaktionszeit wurde als Antwort auf den Ausfall des Betriebssystemprozesses der Nutzersteuerungskapsel durch eine explizite Terminierung (*Process.Kill*) ermittelt und umfasst den Zeitraum vom Erkennen des Ausfalls durch einen entsprechenden Kapsel-Observer bis zur Aktivierung der sicheren Konfiguration, die durch die Verwendung analytischer Redundanz von diesem Moment an sofort gültige Steuersignale produziert. Die Rekonfigurationsdauer ist der Zeitraum vom Auslösen der Rekonfiguration bis zu deren Abschluss.

Die ermittelte Rekonfigurationsdauer beträgt im Mittel 3.31 ± 0.58 ms, wobei sämtliche Messwerte im Bereich unter 4 ms angesiedelt sind. Die Rekonfigurationsdauer ergibt sich im Wesentlichen aus dem Blockieren der Konnektoren, dem Setzen eines Parameters sowie dem Entladen von zwei Kapseln. Die Reaktionszeit liegt im Mittel bei 29.99 ± 3.90 ms und besitzt einen maximalen gemessenen Wert von 38 ms. Neben der Rekonfigurationszeit, die einen direkten Anteil der Reaktionszeit darstellt, wird die Reaktionszeit von der implementierten Monitoring-Infrastruktur und der Auswertung

der Profile beeinflusst. Während die dynamische Rekonfiguration den Fokus dieser Arbeit darstellte und in diesem Bereich viele Optimierungen umgesetzt wurden, sind im Bereich des Monitoring und der Profilanalyse weitere Verbesserungen möglich. Reaktionszeiten im Bereich unter 40 ms beim Pendelexperiment sind aber akzeptabel und erfüllen die gestellten Anforderungen.

7.4 Fallstudien: Dynamische Aktualisierung

Im Rahmen des DSUP-Projekts wurden die zuvor vorgestellten Rekonfigurationsalgorithmen und auch die Verfahren zur Zustandsübertragung in einigen Fallstudien evaluiert. Die Fallstudien und wesentliche Leistungsmerkmale sollen an dieser Stelle vorgestellt werden. Auf der *Homepage* des Fachgebiets für Betriebssysteme und Middleware am Hasso-Plattner-Institut stehen einige Videos [Systems und at Hasso-Plattner-Institute 2008] bereit, welche die dynamische Aktualisierung der untersuchten Fallstudien demonstrieren.

7.4.1 PictureViewer - Ein Bildbetrachter

Mit dem PictureViewer wurde ein einfacher Bildbetrachter entwickelt, der Bilder von der lokalen oder einer entfernten Festplatte anzeigen kann. Die Komponente *PictureSource* kann Bilddateien von der Festplatte einlesen und einer *Viewer*-Komponente zum Anzeigen zur Verfügung stellen. Zusätzlich wurden Filterkomponenten für die Kompression der Bilddaten entwickelt, die zwischen die beiden vorgestellten Komponenten geschaltet werden können.
Für die Leistungsbewertung dieses einfachen Beispiels wurde die *Viewer*-Kapsel mit einem zusätzlichen Effekt bei der Überblendung von Bildern aktualisiert, das heißt es wurde zusätzliche Logik beim Anzeigen der Bilddaten in die existierende Kapsel integriert. In der folgenden Tabelle sind einige Werte der Aktualisierung zusammengefasst.

Aktualisierungsdauer	110 ± 1 ms
traversierte Objektreferenzen	596
aktualisierte Objekte	1

Bei der Aktualisierung wurden 596 Referenzen traversiert und ein Objekt aktualisiert. Die gesamte Aktualisierungszeit fällt mit ca. 110 ms sehr kurz aus. Im Vergleich zur Dauer der Bildbetrachtung durch den Nutzer im Bereich einiger Sekunden ist die verursachte Unterbrechung gering.

7.4.2 Lumisoft-Mail-Server

Für eine zweite Fallstudie wurde eine Kommunikationssuite mit vorhandenem Quellcode herangezogen. Der Lumisoft Mail-Server [LumiSoft 2008] steht als *Freeware* zum Herunterladen bereit und umfasst ca. 61.000 Codezeilen. Für die Evaluierung dynamischer Aspektaktivierungen und -aktualisierungen wurde der IMAP-Server eingesetzt. Für die Integration der notwendigen Rekonfigurationslogik wurde der Quelltext entsprechend angepasst. Dabei mussten nur 2 Zeilen im vorhandenen Code verändert werden, in denen die Erzeugung von Wurzelobjekten über die DSUP-Bibliotheken integriert wurde.

Zunächst wurden die Zusatzkosten für die integrierte Synchronisationslogik evaluiert. Dies wurde durch die Messung des Durchsatzes der IMAP-Server mit und ohne aktivierte Synchronisationsaspekte realisiert. Bei den Messungen wurden wiederholt Nachrichten der Größe 10 kByte von einem Test-Client gelesen. Die Messung ohne aktivierten Synchronsationsaspekt ergab einen Durchsatz von 45,1 ± 3,69 Nachrichten in 10 Sekunden. Mit aktivierter Synchronisationslogik betrug der Durchsatz 45,24 ± 4,49 Nachrichten in 10 Sekunden. Die Zusatzkosten lagen im Rahmen der Messungenauigkeit und können bei dieser Fallstudie als vernachlässigbar betrachtet werden. Zur Leistungsbewertung wurde die dynamische Aktivierung eines *Trace*-Aspekts untersucht, das heißt es wurde während der Laufzeit das *Logging* bestimmter Objektmethoden im IMAP-Server aktiviert. Die folgende Tabelle zeigt die ermittelten Werte.

Aktualisierungsdauer	213 ± 7,5 ms
traversierte Objektreferenzen	1.326
aktualisierte Objekte	3

7.4.3 PaintDotNet 3.0

Im Rahmen der Bildverarbeitungssoftware *PaintDotNet* [dotPDN LLC 2008] sollte die Anwendbarkeit der vorgestellten Techniken an einem sehr komplexen Beispiel gezeigt werden. *PaintDotNet* ist ein freies, quelloffenes Projekt mit mehr als 133.000 Codezeilen.

Die Identifikation von Wurzelobjekten wurde zunächst manuell vom Autor realisiert. Die dafür benötigte Zeit betrug ca. eine Stunde. Es hat sich herausgestellt, dass dieser Arbeitsschritt bei graphischen .NET-Anwendungen automatisiert durchgeführt werden kann. Im Prinzip müssen nur sämtliche von *System.Windows.Forms.Control* abgeleiteten Typen als Wurzelobjekte markiert und damit mit dem Synchronisationsaspekt verwoben werden. Hierfür wurde ein Werkzeug entwickelt, welches die Metadaten der Anwendungs-Assemblies analysiert und in den Quellcode die Objekterzeugung mit dem Aufruf der Fabrikmethode *CreateRootObject* integriert. Die Zusatzkosten konnten bei dieser Fallstudie nicht systematisch untersucht werden. Eine Beeinflussung der Interaktivität war aber nicht bemerkbar.

Aktualisierungsdauer	3,52 ± 0,20 s
traversierte Objektreferenzen	ca. 28.000
aktualisierte Objekte	ca. 200
untersuchte primitive Daten	700.000

Bei der Leistungsbewertung wurde ein kleiner Fehler der Implementierung behoben, der im Zusammenhang mit der Positionierung des Mauszeigers beim Markieren von Objekten stand. Bei der dynamischen Aktualisierung, also dem Einspielen der fehlerbereinigten Version, wurden 28.000 Objektreferenzen traversiert und 700.000 Blattknoten im Objektgraphen gefunden. Ingesamt wurden 200 Objekte aktualisiert. Mit 3,52 Sekunden liegt die Aktualisierungsdauer für das Einspielen einer fehlerbereinigten Version in einem akzeptablen Rahmen. Genauere Untersuchungen haben ergeben, dass sich die Aktualisierungsdauer zu ca. 40 Prozent aus Reflexionsfunktionalität und ca. 40 Prozent aus der Verwaltung der Strukturen zur Zyklenerkennung zusammensetzt.

7.5 Zusammenfassung

Im Rahmen dieses Kapitels wurden komplexe Anwendungsfälle für die in der Arbeit entwickelten Verfahren vorgestellt. Die Leistungsbewertung hat gezeigt, dass die durch die für die dynamische Rekonfiguration notwendige Synchronisationslogik akzeptable Zusatzkosten verursacht, die sich in den beschriebenen Fallstudien zur dynamischen Aktualisierung als vernachlässigbar gezeigt haben. Eine Besonderheit des vorgestellten Rekonfigurationsverfahrens ist die Kapselabstraktion, die es ermöglicht, für die Kommunikation zwischen austauschbaren Einheiten auf einem Rechner virtuelle Methodenaufrufe einzusetzen, während existierende Lösungen immer den Einsatz von Middleware-basierten Fernaufrufen mit hohen Zusatzkosten erfordern. Hierdurch wird der *Overhead* durch die Synchronisationslogik insgesamt minimiert. Auch die durch die dynamische Rekonfiguration verursachte Unterbrechungsdauer bewegt sich bei den durchgeführten Tests im Bereich von μ-Sekunden. Die Nutzung der Adapt.Net-Ausführungsplattform für die sichere Steuerung von eingebetteten Echtzeitexperimenten im Distributed Control Lab hat die Anwendbarkeit der entwickelten Techniken zur Realisierung von Sicherheit und Fehlertoleranz bewiesen. Beim Experiment „Foucaults Pendel“ wurde beim Absturz einer Nutzersteuerung eine Reaktionszeit unter 40 ms gemessen. Komplexe Fallstudien für die dynamische Aktualisierung haben die leichte Integrierbarkeit des entwickelten Verfahrens in existierende Software demonstriert. Die gemessene Aktualisierungsdauer bewegt sich für das Einspielen von fehlerbereinigten Versionen im Bereich von wenigen Sekunden. Auf Grund des begrenzten Platzes wurde im Rahmen dieses Kapitels nur eine Auswahl der vorgenommenen Messungen präsentiert. Einige weitere Messungen können in den Veröffentlichungen des Autors [Rasche und Polze 2003, 2008; Rasche u. a. 2005a, 2003] nachgelesen werden.

8 Verwandte Arbeiten

In diesem Abschnitt werden existierende Forschungsarbeiten auf dem Gebiet der adaptiven Software vorgestellt. Die Arbeiten sollen dabei an Hand von verschiedenen Schwerpunkten diskutiert werden. Zunächst sollen die Motivation und die Zielgruppen der verschiedenen Ansätze unterschieden werden. Von besonderer Bedeutung für diese Arbeit sind die für das Erreichen von Adaptivität eingesetzten Mechanismen sowie die Werkzeuge, die für die Unterstützung von Anwendungsentwicklern erstellt wurden. Zusätzlich soll die Integration der Ansätze in den Softwareentwicklungsprozess diskutiert werden. Gleichzeitig wird der im Rahmen der vorliegenden Arbeit beschriebene Ansatz von den verwandten Arbeiten abgegrenzt. Die verwandten Arbeiten werden dabei in zwei Bereiche unterteilt. Zum einen werden generelle Ansätze für die Entwicklung adaptiver Anwendungen vorgestellt, zum anderen stehen Ansätze für die dynamische Rekonfiguration verteilter Anwendungen im Vordergrund.

8.1 Forschungsprojekte zum Thema „Adaptive Software“

Auf dem Gebiet der Entwicklung adaptiver Anwendungen existiert eine Vielzahl von Forschungsarbeiten, die oft ähnliche Ansätze und Strategien verfolgen. Für die Vorstellung der Forschungsprojekte wurden einige Kriterien bestimmt, um diese zu gruppieren. Auf Grund des begrenzten Platzes werden für die jeweiligen Gruppen nur einige wichtige Vertreter ausgewählt und vorgestellt. Zunächst können die Mechanismen für die Realisierung adaptiver Anwendungen in mehrere Kategorien eingeteilt werden. [Noble u. a. 1997] klassifizieren Adaptionsmechanismen nach der Integration in die Anwendungslogik (siehe Abbildung 8.1). Neben den beiden Extremen der transparenten Adaption (*application-transparent*) und der anwendungsintegrierten (*laissez-faire*) Variante führt [Noble u. a. 1997] im Projekt Odyssey den Begriff der anwendungsbewussten Kollaboration (*application-aware collaboration*) ein.

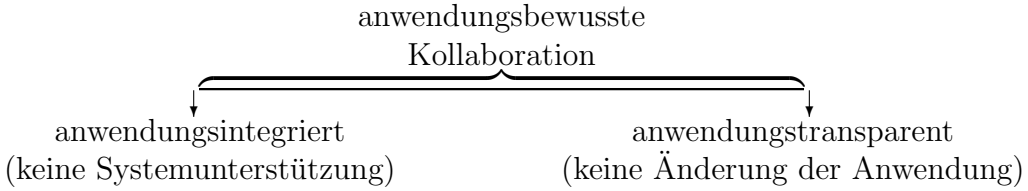


Abbildung 8.1: Einteilung von Adaptionsstrategien nach [Noble u. a. 1997]

Ein Beispiel für die anwendungstransparente Integration von Adaptionstechniken ist Coda [Satyanarayanan u. a. 1990], ein Dateisystem mit Unterstützung eines *Offline*-Modus für mobile Geräte. Coda kann anwendungstransparent die wechselnde Verfügbarkeit von Netzwerkressourcen ausgleichen. Auch mit den *Offline Files and Folders* der Windows-Betriebssysteme ist ein ähnlicher Mechanismus für die transparente Kapselung unterbrochener Netzwerkverbindungen beim Zugriff auf Dateien gegeben. Andere

Beispiele finden sich im Bereich Middleware-basierter Lösungen, die im separaten Abschnitt 8.1.2 vorgestellt werden.

Für die anwendungsintegrierte Adaption wurden in den vorangegangenen Kapiteln schon einige Beispiele in existierenden Produkten vorgestellt. Vertreter dieser Kategorie finden sich z.B. im Bereich von Multimedia-Playern wie z.B. der RealPlayer, bei denen spezielle Protokolle für die Anpassung von Videoströmen eingesetzt werden.

Bei der anwendungsbewussten (*application-aware*) Adaption werden die Anwendungen während der Entwicklungszeit auf ihre Adaptionsfähigkeit vorbereitet. Mit Hilfe spezieller Schnittstellen können Anpassungen in Zusammenarbeit mit Systemkomponenten vorgenommen werden. Anpassungsspezifische Logik kann der Anwendung damit in Form von Bibliotheken bereitgestellt werden - der Anwendungsentwickler muss nicht mehr den gesamten anpassungsspezifischen Code selbst implementieren. Eine Untergruppe anwendungsbewusster Adaptionsansätze basiert auf der Anpassung von Datenströmen; diese werden in Abschnitt 8.1.3 vorgestellt.

Eine weitere Klasse von Forschungsarbeiten untersucht die Schaffung bzw. Erweiterung von Programmiersprachen zur Unterstützung der Entwicklung adaptiver Anwendungen (Abschnitt 8.1.5). Die Verwendung von Werkzeugen der aspektorientierten Programmierung sind Grundlage weiterer Ansätze (Abschnitt 8.1.6), zu denen auch die vorliegende Arbeit gehört. Abschließend werden Ansätze für Adaptivität in Betriebssystemen im Abschnitt 8.1.7 diskutiert.

8.1.1 Die Demeter Methode

Karl Lieberherr prägte im Demeter-Projekt [Lieberherr 1996] den Begriff der adaptiven Softwareentwicklung. Ausgehend von der objektorientierten Softwareentwicklung führt Demeter abstrahierende Konzepte ein, mit denen adaptive Programme beschrieben werden können. *Propagation-Patterns* beschreiben Klassenbeziehungen nur partiell und führen damit Einschränkungen an ein globales Klassenverzeichnis ein. Bei der Implementierung von Klassen werden zunächst nur Methoden spezifiziert, wenn sie eine essentielle Funktionalität des Problems beinhalten. Entsprechende Funktionalität wird dem adaptiven Programm durch Klassen eines Klassenverzeichnisses zur Verfügung gestellt. Beim Erstellen adaptiver Programme wird aus dem Klassenverzeichnis ein Subgraph extrahiert, welcher die Anforderungen der spezifizierten Einschränkungen einhält. Das Besondere bezüglich der Adaption (hier *Adaptiveness*) ist, dass potentiell viele mögliche Lösungen existieren. Ein Exemplar einer solchen „Programmfamilie“ [Parnas 2001] wird schließlich ausgewählt und übersetzt. Es wird versucht Klassenstrukturdetails bei der Entwicklung so weit wie möglich entkoppelt zu beschreiben. In Demeter wurde zunächst nur die statische Adaption untersucht.

Die Adaptivität beim Demeter-Ansatz ist eng an den Softwareentwicklungsprozess gebunden und stellt eine Abstraktion gegenüber der Objektorientierung bereit, wobei diese die Grundlage des Ansatzes bildet. Die Adaptivität in Demeter bezieht sich auf den Softwareentwicklungsprozess selbst, da Klassenstrukturen und Vererbungsbeziehungen leicht an Änderung der Anforderungen an die Software angepasst werden können. Heute ist das Demeter-Projekt eng mit der aspektorientierten Programmierung verbunden. Im Buch „*Adaptive Object-Oriented Software - The Demeter Method*“ [Lieberherr 1996] stellt Karl Lieberherr die Demeter-Methode detailliert vor.

8.1.2 Middleware-basierte Ansätze

Als Middleware wird eine Softwareschicht zwischen Anwendung und Betriebssystem bezeichnet, die eine bestimmte Funktionalität für eine Anwendung bereitstellt und von Betriebssystemkonzepten abstrahiert. Die Funktionalitäten der Middleware befinden sich in einem ständigen Wandel. So können von Anwendungen oft benutzte Funktionen in die Middleware verlagert werden, während Funktionalität der Middleware in das Betriebssystem integriert werden kann. Einige Projekte, die im Folgenden vorgestellt werden sollen, untersuchen die Integration von Anpassungsmechanismen in die **Kommunikations-Middleware**, die Mechanismen für die Nachrichtenübertragung und Synchronisation in einem verteilten System bereitstellt. Typische Vertreter von Kommunikations-Middleware unterstützen Objektfernaufrufe (ORPCs) oder arbeiten nach dem *Publish-/Subscribe*-Verfahren. Middleware-basierte Adaptionsansätze können in zwei Kategorien eingeteilt werden. Unterscheiden lassen sich zum einen Ansätze, bei denen die Anpassung der Middleware selbst, z.B. durch die Verwendung verschiedener Strategien für die Bearbeitung von Nachrichten, im Vordergrund steht (z.B.: DynamicTAO [Kon u. a. 2000]) oder die Middleware Konfigurationsoptionen für die Anwendung bietet, mit der diese Anpassungen an Middleware-spezifischen Funktionen vornehmen kann (z.B.: Auswahl von Nachrichtengrößen, oder Transportkanälen). Zum anderen existieren Ansätze, bei denen die Anwendungen selbst Möglichkeiten zur Auswahl von Alternativen anbieten. Eine Beispiel hierfür ist die Unterstützung der Auswahl alternativer Methoden bei der Verarbeitung von Objektfernaufrufen in Abhängigkeit von Umgebungseigenschaften bei den Quality Objects [Vanegas u. a. 1998; Zinky u. a. 1997].

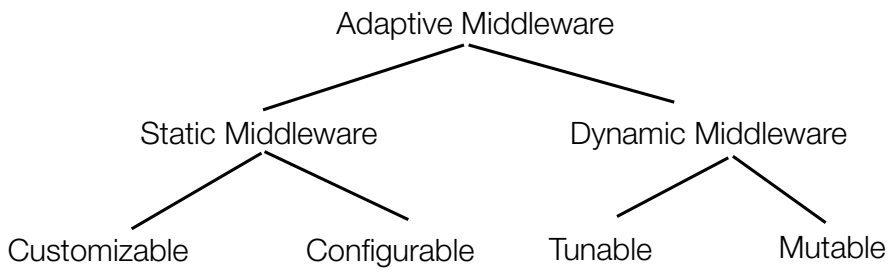


Abbildung 8.2: Klassifikation adaptiver Middleware nach [Sadjadi und McKinley 2003]

Eine Middleware, die Adaptionsmechanismen unterstützt, wird auch als „Adaptive Middleware“ bezeichnet. In Abhängigkeit vom Konfigurationszeitpunkt klassifiziert [Sadjadi und McKinley 2003] „Adaptive Middleware“ zunächst in statische und dynamische (Abbildung 8.2). „Customizable Middleware“ kann z.B. durch Compiler-Flaggen oder Präprozessor-Direktiven während der Übersetzungsphase angepasst werden (Quality Objects). Bei „configurable Middleware“ erfolgt die Anpassung während der Startphase unter Anderem durch Kommandozeilenparameter oder Konfigurationsdateien (z.B. TAO [Schmidt u. a. 1997]). Bei Vertretern der Klasse „tunable Middleware“ können Anpassungen nach der Startphase aber vor der eigentlichen Nutzung vorgenommen werden. Schlussendlich werden Anpassungen während der Anwendungslaufzeit von „mutable Middleware“ unterstützt. Eine Zusammenfassung über aktuelle Techniken von Middleware-Infrastrukturen sind im Buch „Middleware for Communication“ [Mahmoud 2004] beschrieben. Einige wichtige Repräsentanten der verschiedenen Klassen sollen im Folgenden vorgestellt werden.

Quality Objects

Quality Objects (QuO) [Vanegas u. a. 1998; Zinky u. a. 1997] sind eine Erweiterung der CORBA-Middleware, die von der Firma BBN Technologies entwickelt wurde. QuO unterstützen neben der funktionalen Schnittstellenspezifikation (durch IDL) auch die Spezifikation von Dienstgüteeigenschaften (QoS). Mit Hilfe der Sprachfamilie *Quality Description Language* (QDL)[Pal u. a. 2000] ist es möglich, Dienstgüteaspekte einer Anwendung zu beschreiben. Im Rahmen des QuO-Projekts wurde die Rolle des QoS-Entwicklers (QoS-Entwickler) definiert, der für die Entwicklung von QoS-Aspekten verantwortlich ist. Grundlage adaptiven Verhaltens ist die Definition eines QoS-Vertrags, mit denen Bereiche (*regions*) möglicher Einsatzbedingungen klassifiziert werden können. Client- und Server-Anwendungen können für jeden Bereich Anforderungen an die gewünschten Dienstgüteeigenschaften stellen. Definiert werden Bereiche als Prädikate über Systemgegebenheiten (*system-conditions*); dies sind Objekte, welche die Gegebenheiten der Umgebungen einer Anwendung ermitteln bzw. messen können. Sie werden als normale Anwendungsobjekte implementiert oder aus einem Repository entnommen.
Für einen QuO-Client stellt sich die Kommunikation wie ein normaler ORPC-Aufruf dar. Der Aufruf wird aber zunächst an einen Delegate (Proxy) weitergeleitet, welcher durch eine neue *bind*()-Methode (*connect*) bei der Objektinitialisierung erzeugt wurde. Der Delegate wertet über den QuO-Kernel die Systemgegebenheiten aus. Mit Hilfe der definierten Bereiche kann innerhalb des Delegates alternatives Verhalten implementiert werden. Die *Structure Description Language* (SDL) erlaubt QoS-Entwicklern, adaptives Verhalten zu realisieren, indem für jeden Bereich eines mit der *Contract Definition Language (CDL)* definierten Vertrages alternative Abläufe formuliert werden. Als Alternativen stehen dem QoS-Entwickler der Aufruf verschiedener entfernter Objekte (unter anderem auch mehrere Aufrufe an z.B. replizierte Objekte), der Aufruf anderer Methoden oder die lokale Berechnung zur Verfügung. Ein zweiter Mechanismus zur Realisierung adaptiven Verhaltens sind *Callbacks*, die beim Verlassen eines Bereichs des QoS-Vertrags aufgerufen werden können, um alternative Mechanismen im Client zu aktivieren.

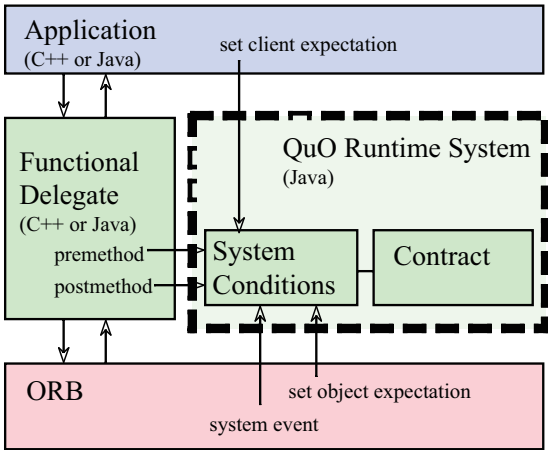


Abbildung 8.3: Quality Objects - Architektur [Zinky u. a. 1997]

Abbildung 8.3 zeigt die Architektur der QuO-Laufzeitumgebung. Durch die beidseitige Integration der Delegate-Objekte ist es möglich, adaptives Verhalten auch Clientseitig zu implementieren. Die QuO-Laufzeitumgebung kann mit Hilfe der QoS-Verträge

alternatives Verhalten implementieren: Ausgehend von den funktionalen Schnittstellenbschreibungen und den QDL-Beschreibungen können QuO-Compiler einen Rahmen für adaptive Anwendungen generieren. Der Ansatz bietet zwar ein gewisse Abstraktion bei der Entwicklung der Adaptionsaspekte - durch die explizite Implementierung der Alternativlogik in den SDL-Dokumenten muss noch eine erhebliche Menge von adaptionsspezifischem Code entwickelt werden. Ein unabhängiger Test der einzelnen Konfigurationen der Anwendungen ist im Gegensatz zur vorliegenden Arbeit bei diesem Ansatz nicht möglich.

DynamicTAO

DynamicTAO [Kon u. a. 2000] ist eine CORBA-ORB-Implementierung, die an der *University of Illinois* unter anderem von Fabio Kon entwickelt wurde. DynamicTAO erweitert TAO (The ACE ORB) [Schmidt u. a. 1997] ein ORB, der die Echtzeiterweiterung von CORBA (Real-Time CORBA [Fay-Wolfe u. a. 2000]) implementiert, um die Fähigkeit der dynamischen Rekonfiguration von ORB-Modulen. Während TAO zur Startzeit konfiguriert wird, können im DynamicTAO-ORB Module während der Laufzeit rekonfiguriert, nachgeladen bzw. entladen werden. Dabei können u.a. neue Scheduling-, Thread-Pool- oder Sicherheitsstrategien unter Verwendung des Strategie-Entwurfsmusters [Gamma u. a. 1995] konfiguriert werden. Die Verwendung des Strategie-Entwurfsmusters stammt ursprünglich aus dem TAO-Projekt, bei dem neue Strategien nur vor der Laufzeit mit Hilfe einer Konfigurationsdatei ausgewählt werden konnten und dann während der Startphase die entsprechenden Module geladen werden.
Der implementierte ORB speichert seine interne Struktur und benutzt diese Information während der Rekonfigurationsphase. Wird eine Strategie innerhalb eines ORBs ersetzt, werden alle aktiven *Requests* mit der alten Strategie verarbeitet, während neue *Requests* mit dem neuen Strategiemodul verarbeitet werden. Der Zustandstransfer zwischen Strategiemodulen wird über anwendungsspezifische Lösungen realisiert.
Für die dynamische Rekonfiguration von multiplen ORBs wurde eine Stapelverarbeitungsfunktion integriert. Rekonfigurationsagenten können Rekonfigurationsanforderungen an den einzelnen ORBs parallel ausführen. Ein Administrator kann dabei eine Topologie für die Migration der Agenten definieren. Mit Hilfe einer Monitoring-Umgebung können Rekonfigurationsprozesse automatisiert ausgelöst werden. DynamicTAO ist ein Beispiel für die Anpassung der Middleware. Anwendungen können mit diesem Ansatz selbst nur bedingt angepasst werden, indem sie von der Rekonfiguration der unterliegenden Infrastruktur profitieren.
Im Rahmen der Implementierung von TAO, DynamicTAO und dem TAO zugrunde liegenden Adaptive Communication Environment (ACE [Schmidt 1998]) wurden eine Reihe von Architektur- und Entwurfsmuster identifiziert, welche in der Buchreihe „Pattern-Oriented Software Architecture“ [Buschmann u. a. 1996; Schmidt u. a. 2000] beschrieben wurden. Diese sind zum Teil auch für diese Arbeit von Bedeutung. Deshalb sollen drei von ihnen im Folgenden vorgestellt werden.
Unter diesen Mustern findet sich das **Component-Configurator**-Entwurfsmuster, welches die Konfiguration von Softwarekomponenten von der Implementierung der Funktionaliät separiert. Komponenten müssen hierfür Methoden zur Initialisierung (*init*), Beendigung (*fini*), Unterbrechung (*suspend*), Fortsetzung (*resume*) sowie Informationsabfragen implementieren. Über diese Schnittstellen kann die Konfiguration im Sinne des Lebenszyklus einer Komponente realisiert werden. Der im Rahmen dieser Arbeit entwickelte Kapselkonfigurator greift dieses Muster auf und erweitert es.

Das **Virtual-Component**-Architekturmuster bietet eine transparente Möglichkeit für das Laden und Entladen von Komponenten. Ziel ist dabei vor allem die Einsparung von Hauptspeicher in eingebetteten Systemen. Eine Anwendung oder auch der ORB einer Middleware muss dabei in Subsysteme zerlegt werden. Während der Laufzeit werden dann nur Komponenten geladen, die wirklich benötigt werden. Durch die Implementierung von Lade- und Entladestrategien wird der eigentlichen Anwendung die Komplexität verborgen.

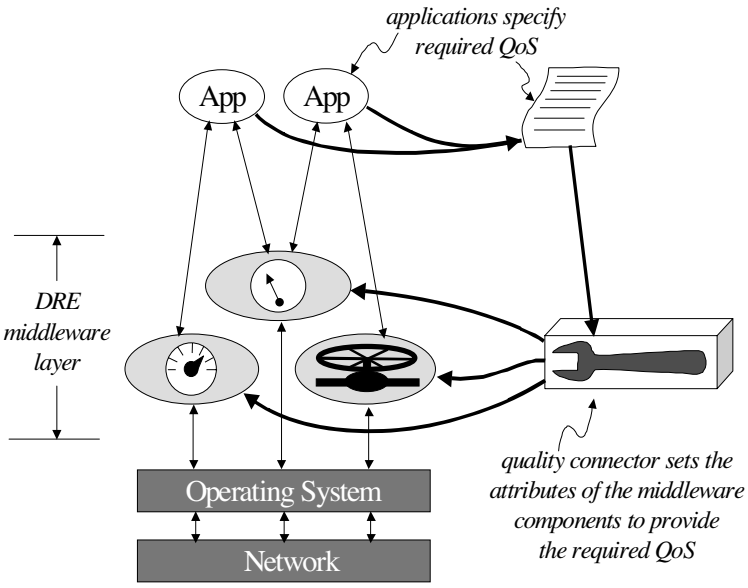


Abbildung 8.4: Quality Connector Architekturmuster [Cross und Schmidt 2003]

Das **Quality-Connector**-Architekturmuster [Cross und Schmidt 2003] bietet eine Lösung für das Problem heterogener Middleware-Schnittstellen zur Konfiguration von Dienstgüteeigenschaften. Die Autoren des Architekturmusters schlagen vor, die Spezifikation von Dienstgüteanforderungen einer Anwendung separat von der Konfigurationslogik in einer zu definierenden Beschreibungssprache durchzuführen. Über eine Verbindungskomponente (*Quality Connector*), welche für jede Middleware-Implementierung zu erstellen ist, werden dann über einheitliche Schnittstellen Konfigurationsschritte durchgeführt. Durch die Wiederverwendbarkeit der Verbindungskomponenten kann die Portabilität von QoS-sensitiven Anwendungen verbessert werden. Abbildung 8.4 verdeutlicht dieses Architekturmuster. Auch dieses Muster kann leicht in die in dieser Arbeit vorgestellte Ausführungsplattform integriert werden, indem eine dedizierte Kapsel einer Anwendungskonfiguration über ihre Parameter die unterliegende Middleware konfiguriert.

DotQoS - Dienstgüte mit .NET Remoting

DotQoS [Ulbrich u. a. 2003] ist eine Erweiterung der .NET-Kommunikations-Middleware .NET Remoting, um Mechanismen zur Dienstgütekontrolle. Dienstgütevereinbarungen werden in DotQoS als Objekte repräsentiert und können in einer .NET Programmiersprache (z.B. C#) definiert werden. Dienstimplementierungen (Komponenten) können mit Hilfe von CLI-Attributen mit Informationen über unterstützte Dienstgüteparameter annotiert werden. Neben der deklarativen Beschreibung von Dienstgütevereinbarungen unterstützt DotQoS Mechanismen zur Realisierung der ver-

einbarten Dienstgüten. Hierfür wird die .NET-Remoting-Infrastruktur genutzt, welche die Integration von Filtern *(interceptor)* in die Nachrichtenverarbeitung auf Client- und Server-Seite erlaubt. Mit Hilfe dieser Filter können vereinbarte Dienstgüteparameter per Reflexion überprüft und umgesetzt werden. Die Umsetzung von Dienstgüten kann unter anderem durch Manipulation der Nachrichten implementiert werden. Das Problem mehrdimensionaler Dienstgüteanforderungen, die sich unter Umständen gegenseitig beeinflussen, kann durch die Angabe der Bearbeitungsreihenfolge von Dienstgütemechanismen konfiguriert werden. Der Austausch der Mechanismen bzw. der Filter ist mit DotQoS nicht während der Laufzeit möglich, wohl aber die Änderung der Dienstgüteanforderungen.

AspectIX

AspectIX [Geier u. a. 1998] ist eine objektbasierte Kommunikationsmiddleware, die an der *Friedrich-Alexander Universität Erlangen-Nürnberg* entwickelt wurde. In AspectIX bestehen Objekte aus ein oder mehreren Fragmenten, die über mehrere Rechner verteilt sein können und die variable Implementierung fixer CORBA-kompatibler Schnittstellen realisieren. Client-Anwendungen können unter Verwendung von Standard-CORBA-Mechanismen auf die Objekte zugreifen. Mobilität und Replikation wird durch die Möglichkeit des transparenten Hinzufügens von Fragmenten unterstützt. Die Implementierung der Schnittstellen der AspectIX-Objekte kann durch die von der Objekthülle gekapselten Fragmente atomar für die Client-Anwendungen verändert werden. Die Rekonfiguration von AspectIX-Anwendungen kann über Aspekte gesteuert werden, die über programmatische Schnittstellen konfiguriert werden können.

OpenORB

OpenORB [Blair u. a. 2002] ist eine reflexive Middleware, die an der University of Lanchester entwickelt wurde. OpenORB unterstützt strombasierte (*stream-oriented*) Multimediaanwendungen. Ähnlich wie in Trap/J (Abschnitt 8.1.6) kann adaptives Verhalten durch Abfangen von Interaktionen einer funktionalen Ebene sowie durch Umleitung und Manipulation in einer Metaebene umgesetzt werden. Mit diesem Verfahren können Interaktionen durch Vor- und Nachbehandlung von übermittelten Daten angepasst werden. Zusätzlich können Verbindungen zwischen Anwendungsobjekten mit OpenORB in einer Metaebene konfiguriert werden. Dabei muss der Anwendungsentwickler explizit, wie z.B. auch in DACIA (Abschnitt 8.1.4), die Verbindung von Komponenten implementieren. Die Metaebene spielt in OpenORB eine wichtige Rolle zur Umsetzung adaptiven Verhaltens. Neben den Metaobjekten *architecture meta-object* und *interception meta-object* zur Verbindungskonfiguration und der Manipulation von Nachrichten können über das *interface meta-object* Schnittstellen erweitert und über das *resources meta-object* Betriebsmittel konfiguriert werden.

Abgrenzung

Die vorgestellten Middleware-basierten Ansätze bieten viel versprechende Lösungen für die Entwicklung adaptiver Anwendungen. Während mit der vorliegenden Arbeit einige der vorgestellten Ansätze wie die mögliche Justierung von Middleware-Parametern über *Quality Connector*-Kapseln oder der Einsatz der Kapselkonfiguratoren aufgegriffen wurde, geht der in dieser Arbeit vorgestellte Ansatz über die reinen Middleware-

basierten Lösungen hinaus. Vor allem die Möglichkeit des unabhängigen Tests einzelner Anwendungskonfigurationen für bestimmte Umgebungssituationen wird von dem in dieser Arbeit vorgestellten Ansatz unterstützt, während die Middleware-basierten Ansätze, wie z.B. OpenORB, die Rekonfiguration explizit implementieren. Die reine Anpassung der Middleware-Konfiguration wie in DynamicTAO verbietet den Einsatz anwendungsspezifischer Anpassungen. Ein Beispiel hierfür ist die Anpassung der graphischen Nutzerschnittstelle zur Visualisierung eingeschränkter Funktionalität, die in der vorliegenden Arbeit beschrieben wurde. Auch die Leistungsbewertung im Rahmen dieser Arbeit hat gezeigt, dass die Adaption auf Basis der Kommunikations-Middleware mit Performanceeinbußen verbunden sein kann, da Methodenaufrufe über eine Middleware mit zusätzlichen Kosten für das Verpacken von Methodenparametern oder der Nachrichtenverarbeitung verbunden sind. Die vorliegende Arbeit unterstützt demgegenüber auch die effiziente Anpassung innerhalb eines Betriebssystemprozesses. Vor allem aber die mögliche Werkzeugunterstützung der vorliegenden Arbeit übertrifft existierende Middleware-basierte Ansätze.

8.1.3 Datenorientierte Adaption

Die folgenden Projekte vereint die Gemeinsamkeit der Realisierung der Adaption durch die Anpassung von Daten, die zwischen Teilen der Anwendungen transferiert werden. Einige Architekturmerkmale wichtiger Projekte sollen im Folgenden vorgestellt werden.

Das BARWAN-Projekt

Ziel des BARWAN-Projektes [Brewer u. a. 1998], welches an der *University of California* in *Berkeley* durchgeführt wurde, war es, die Entwicklung von Software für mobile Netzwerke zu untersuchen. Als Infrastruktur wurde ein *Overlay*-Netzwerk aus vier Netzwerken installiert, welches für die lokale Kommunikation Infrarot, im Nahbereich ein WaveLAN, im städtischen Großraum ein paketbasiertes Radionetzwerk und schließlich Satellitenkommunikation benutzte. Im Projekt sollte entsprechende Software für die Nutzung des installierten Testbett-Netzwerkes implementiert werden und Anwendungen für heterogene Endgeräte entwickelt werden. Im BARWAN-Projekt wurden *Hand-Over*-Techniken zur transparenten Umschaltung zwischen den verschiedenen Netzwerken (horizontal) und verschiedenen Basisstationen (vertikal) untersucht. Zusätzlich wurde ein Mechanismus zur Anpassung von Daten für die jeweiligen Netzwerkeigenschaften entwickelt. Ein zentraler Bestandteil des BARWAN-Projekts ist die Proxy-Architektur zur dynamischen Anpassung von Inhalten an die Hard-/Software-Eigenschaften der verwendeten heterogenen, mobilen Endgeräte sowie variierenden Netzwerkeigenschaften. Dabei werden die Daten bei Bedarf für die entsprechenden Zielgeräte transformiert - ein Prozess der als *distillation* bezeichnet wird. Dabei wird mit Hilfe verlustbehafteter, datentypspezifischer Kompression versucht die Daten in reduzierter Größe zu übertragen. Entsprechende Datentyp-Behandlungskomponenten (*Transcoder*) implementieren die Transformation der Daten und können von Anwendungsentwicklern bereitgestellt werden. Als Adaptionsparameter wurden die Auflösung sowie die Anzahl der darstellbaren Farben der Endgeräte sowie die verfügbare Netzwerkbandbreite betrachtet. Die entsprechenden *Transcoder*-Konfigurationen werden an Hand von groben Geräteklassifikationen (*Midrange PC, High-end Laptop, Palm Pilot*) sowie grobgranularen Bandbreitenabschnitten (*Low, Medium Bandwidth*) festgelegt.

Im Rahmen des BARWAN-Projekts wurde ein Webbrowser für verschiedene Endgeräte implementiert. Dabei wurden Datentransformatoren für *Postscript*, Video- und Audiodaten sowie GIF-Bilder implementiert und evaluiert.

Anwendungsbewusste Adaption mit Odyssey

Odyssey [Noble 2000; Noble u. a. 1997] ist eine Programmierumgebung für Anwendungen mit mobilem Informationszugriff, die an der University of Michigan unter der Leitung von B. Noble und M. Satyananarayanan entwickelt wurde. Neben der Klassifikation von Adaptionsstrategien wurde mit Odyssey der Begriff der anwendungsbewussten Kollaboration eingeführt. Hierbei handelt es sich wie schon zuvor kurz beschrieben um ein Konzept zur Trennung wiederverwendbarer Adaptionslogik, die als Teil des Betriebssystems zur Verfügung gestellt wird, und anwendungsspezifischer Teile, die unter Benutzung der Betriebssystemfunktionalität die eigentliche Anpassung implementieren.
Die Anpassung von Anwendungen an die verfügbaren Ressourcen wird in Odyssey durch eine verlustbehaftete Veränderung der Daten erreicht. In diesem Zusammenhang wurden zwei wichtige Metriken eingeführt, die für den Bereich der adaptiven Softwareentwicklung geprägt haben.

- *Fidelity* als Maß für Datenqualität, wobei den Originaldaten der Wert 1 und der leeren Information eine 0 zugeordnet wird. Der Adaptionsmechanismus von Odyssey basiert darauf, die *Fidelity* des Datenstroms so zu verändern, dass auf veränderte verfügbare Ressourcen optimal reagiert werden kann.

- *Agility* als Maß für die Zeit, die ein System braucht, um auf Änderungen der Umgebung zu reagieren. Dabei muss nur bei Systemen mit ständig wechselnden Bedingungen ein hoher Wert erreicht werden, während bei stabilen Systemen die Schnelligkeit der Reaktion auf Änderungen unwichtiger wird. *Agility* ist eine Systemeigenschaft, die Aufschluss darüber gibt, wie genau auf Änderungen bestimmter Ressourcen eingegangen werden kann.

Odyssey besteht aus einer Reihe von Betriebssystemerweiterungen. Der Zugriff auf die von Odyssey verwalteten Objekte erfolgt über Dateisystemzugriffe. Odyssey unterscheidet zwei unterschiedliche Typen von Objekten: den Vizekönig (*viceroy*) und mehrere Wächter (*wardens*). Der als Singleton realisierte Vizekönig ist ein anwendungsübergreifendes Exemplar für die Überwachung von Ressourcen und ist für die Benachrichtigung von Änderungen der verfügbaren Ressourcen an die Anwendungen verantwortlich. Eine Anwendung kann sich für die Nutzung einer Ressource registrieren (*resource request*) und dabei ein *window of tolerance* (akzeptabler Bereich für die benötigte Ressource) spezifizieren, bei dessen Verlassen die Anwendung benachrichtigt wird. Reicht eine Ressource nicht für die Erbringung einer von der Anwendung gewählten *Fidelity* aus, wird der Anwendung dies mitgeteilt, worauf diese für die verfügbaren Ressourcen angemessene Daten in einer entsprechenden *Fidelity* auswählt.
Die zuvor erwähnten Wächter abstrahieren die datenspezifische Auswahl einer *Fidelity* und deren Operationen auf den Anwendungsdaten. Eine Anwendung kann über den entsprechenden Wächter eine *Fidelity* für einen bestimmten Datentyp auswählen und damit einer entsprechenden Ressourcenverfügbarkeit genügen. Die Schnittstelle zwischen Anwendung und Wächter erfolgt über datentypspezifische Operationen (*type-specific operation*), das heißt es wird nicht versucht, über eine generische Schnittstelle,

z.B. unter Verwendung einer Datenrate, die zu verwendende Datengüte zu konfigurieren, sondern die Anwendung kann über datentypspezifische Funktionen die Konfiguration der *Fidelity* vornehmen. Implementiert wurden die datentypspezifischen Operationen ähnlich den von UNIX bekannten ioctl()'s.
Als Beispielanwendungen wurden ein WebBrowser, eine Video-Abspielsoftware und eine Spracherkennungssoftware vor allem auf die erreichte *Agility* untersucht.

Regelbasierte QoS-Anpassung

Klara Nahrstedt et al. von der University of Illinois at Urbana-Champaign entwickelten mit dem *„Control-based Middleware Framework for Quality of Service Adaption“* ([Li und Nahrstedt 1999]) eine Middleware-Erweiterung, die durch die Parameteranpassung von Anwendungskomponenten sowie der Steuerung der Ressourcenvergabe adaptive Anwendungen ermöglicht. Dabei wird der Adaptionsprozess in zwei Schritte unterteilt. Das *Task-Control-Model* bestimmt mit Hilfe von Methoden der Regelungstechnik, wie aus den zu überwachenden Eigenschaften des Systems Kontrollsignale generiert werden müssen. In einem zweiten Schritt werden dann unter Verwendung von Fuzzy-Algorithmen[1] anwendungsspezifische Parameteränderungen (*parameter-tuning*) für die beteiligten Komponenten erzeugt. Abbildung 8.1.1 zeigt die grobe Struktur dieses Ansatzes. Es zeigt, wie die zu überwachenden Systemeigenschaften durch den *Observation-Task* gewonnen werden. Der *Adaption-Task* beinhaltet den eigentlichen Regler, aus dessen Ausgangssignalen der *Configurator* anwendungsspezifische Parameteränderungen mit einer Fuzzy-Regelbasis erzeugt. Der *Configurator* rekonfiguriert dann entsprechend die *Target-Task*, die eine Komponente des Gesamtsystems darstellt. Das angewandte Verfahren wird an jeder Komponente der Anwendung durchgeführt. Die Verarbeitung der Anwendungstasks erfolgt zyklisch. Die Anpassung durch eine reglerbasierte Parameteradaption kann auch in dem in dieser Arbeit vorgestellten Ansatz zur Parameterfeinjustierung verwendet werden (siehe Abschnitt 5.2.1).

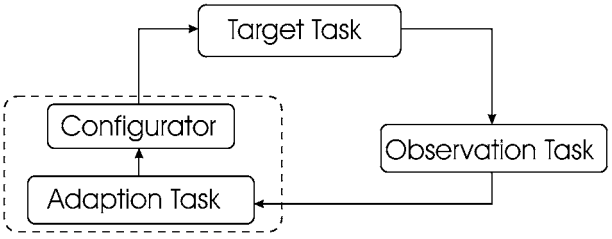


Tabelle 8.1.1: Task Control Modell nach [Li und Nahrstedt 1999]

Eine Besonderheit dieses Ansatzes ist, dass auch die Fairness zwischen mehreren Anwendungen, die das vorgestellte Framework benutzen, unter anderem durch theoretische Überlegungen betrachtet wird.
Als Beispielanwendung wird eine verteilte *Tracking*-Applikation vorgestellt. Als *Tracking* wird im Allgemeinen die Verfolgung eines sich bewegenden Objektes in einem Videostrom verstanden. Als kritischer Parameter wird die *Tracking*-Präzision identifiziert. Je geringer diese ist, desto wahrscheinlicher ist es, das zu verfolgende Objekt zu verlieren. Die Präzision muss also auf einem bestimmten Niveau gehalten werden. Beeinflusst wird die *Tracking*-Präzision durch den verwendeten *Tracking*-Algorithmus sowie die Qualität des zur Verfügung stehenden Videostroms. Ist die

[1] Fuzzytheorie - Theorie von unscharfen Mengen

verfügbare Netzwerkbandbreite für die Übertragung des Videostroms zu gering, muss dieser komprimiert, der Bildausschnitt verkleinert oder die Farbtiefe herabgesetzt werden (beeinflussbare Parameter). Im Gegensatz dazu kann ein CPU-lastigerer *Tracking*-Algorithmus verwendet werden.

Abgrenzung

Auch die datenorientierte Adaption bietet leistungsfähige Ansätze für die Realisierung adaptiver Anwendungen. Vor allem haben diese Techniken schon heute Einzug in einige Produkte gefunden. Neben den schon erwähnten Multimedia-Playern filtern auch einige Mobilfunkanbieter die Dateninhalte beim Web-Surfen über langsame Verbindungen. So kann z.B. transparent die Auflösung von Bildern auf Webseiten reduziert oder die Kompression erhöht werden. Trotzdem fehlen bei der datenorientierten Adaption einige Mechanismen, welche die Adaptivität noch leistungsfähiger realisieren können. So unterstützt die vorliegende Arbeit auch den Einsatz alternativer Algorithmen für die Realisierung einer bestimmten Funktionalität durch die dynamische Aktualisierung von Kapseln und auch die Migration von Kapseln während der Laufzeit, welche mit datenorientierten Ansätzen nicht möglich ist. Auch die Änderung von Verbindungsstrukturen wie z.B. dem nachträglichen Einfügen zusätzlicher Filter ist mit der datenorientierten Adaption nur begrenzt möglich.

8.1.4 Softwarearchitekturbasierte Adaption

[Oreizy u. a. 1999] skizzieren einen Ansatz für Softwareanpassungen auf der Architekturebene. Anwendungen bestehen aus Komponenten, die durch aktive Konnektoren miteinander verbunden sind. Eine wichtige Forderung ist die Unabhängigkeit der Komponenten, die ihre Kommunikationspartner nicht kennen sollen. Anpassungen der Softwarearchitektur werden als *adaptation in-the-large* bezeichnet. Als mögliche Kandidaten für die Umsetzung dynamischer Softwarearchitekturen werden Weaves und C2 vorgestellt, die durch asynchrone Kommunikationsmechanismen zwischen Komponenten eine lose Kopplung realisieren. Um die Konsistenz einer Anwendung während einer Adaption zu gewährleisten, wird der *Architecture Evolution Manager* vorgestellt. Dieser kann anwendungsspezifische Randbedingungen überprüfen und die atomare Ausführung einer Rekonfiguration garantieren. Konstraints können dabei das Vorhandensein eines bestimmten Komponententyps oder Anforderungen an Verbindungen zwischen Komponenten sein.
[Oreizy u. a. 1999] identifiziert das Problem von Komponenten- bzw. Objektmodellen und Programmiersprachen, die keine Unterstützung für die Erhaltung der Konsistenz während der Anpassung laufender Software bieten. Der Ansatz beruft sich auf die lose Kopplung der Komponenten, die durch asynchrone Kommunikationsverfahren umgesetzt werden können. Neben den nachfolgend vorgestellten Plattformen *Weaves* und C2 eignen sich auch *Tupelspaces* [Gelernter 1985] bzw. *Object Spaces* [**?**] für die Umsetzung der beschriebenen Technologie.

Weaves

Weaves [Gorlick und Razouk 1991] sind Netzwerke parallel laufender *Tool Fragmente*, welche durch den Austausch von Objekten miteinander kommunizieren. Jedes *Tool Fragment* beinhaltet einen oder mehrere Threads. Ein *Weaver* ist in der Lage, durch

Auflösung von Ein-/Ausgangs-Beziehungen der einzelnen Fragmente, ausführbare Anwendungen zu erzeugen. *Instrumente* sind spezielle Fragmente, die in den Kommunikationsfluss anderer Fragmente eingebracht werden können, um deren Leistungsfähigkeit zu evaluieren. In Weaves existieren Regeln, die eine lose Kopplung von Komponenten ermöglichen. Dazu gehört, dass Komponenten kein Wissen über Quell- und Zielkomponenten ausgetauschter Objekte haben, keine Annahmen über die Semantik von Konnektoren existieren und eine Komponente kein Wissen über den Ausfall einer Verbindung hat.

C2

C2 ist ein nachrichtenorientierter Architukturstil für die Konstruktion flexibler, erweiterbarer Softwarearchitekturen [Medvidovic u. a. 1997]. Komponenten werden über Konnektoren verbunden - die Verarbeitung von Nachrichten wird durch Framework-Funktionen unterstützt. Entwickler können auf eine Reihe vordefinierter Komponenten und Konnektoren zurückgreifen (abstrakte Klassen) und diese verfeinern. C2-Frameworks stehen für Java und C++ zur Verfügung. Neben der Integration zahlreicher Standardkomponenten (Graphische Subsysteme - *OTS Graphics Toolkit*) wurde eine Infrastruktur zum Web-basierten Deployment von Komponenten entwickelt. Mit Hilfe des Werkzeugs *ArchShell* [Oreizy 1996] können Laufzeitmodifikationen an C2-Architekturen durchgeführt werden. *ArchShell* erlaubt die Konfiguration von *pipe- und filter*-Architekturen. Dabei können während der Laufzeit neue Komponenten integriert und verbunden werden. Durch die Verwendung asynchroner Kommunikation können Laufzeitmodifikationen leicht durchgeführt werden.

DACIA

Ein weiteres *Framework* zur Entwicklung adaptiver verteilter Anwendungen ist DACIA (*Dynamic Adjustment of Component InterActions*) [Litiu und Prakash 2000]. Die Adaption einer Anwendung erfolgt in DACIA durch Komponentenmigration, dem Bewegen von Komponenten zu einem anderen Rechner, sowie durch die dynamische Rekonfiguration von Verbindungen zwischen Anwendungskomponenten. Anwendungsspezifische Konfigurationsentscheidungen werden von einer Monitorkomponente getroffen, welche Teil einer so genannten *Engine*, dem Kernstück von DACIA ist. Der Monitor muss vom Anwendungsprogrammierer implementiert werden, was den größten Aufwand für die Anpassung der Anwendungen in dessen Hände legt. Im Monitor können Funktionen zum Verbinden, Trennen und Erzeugen von Komponenten der *Engine* aufgerufen werden. Der Entwickler erstellt für die Änderung der Anwendungsinteraktionen Rekonfigurationsskripte, welche die Änderungen an der Anwendungsarchitektur enthalten. Mit Hilfe dieser Skripte können z.B. De-/Kompressionsmodule in existierende Kommunikationspfade integriert werden. Ein Beispiel für ein DACIA-Rekonfigurationsskript wird im 3. Kapitel vorgestellt, wo zusätzlich die skriptbasierte Beschreibung von Softwareänderungen einer deklarativen Konfigurationsbeschreibung gegenübergestellt wird.

BARK

Ein weiterer Vertreter für die architekturbasierte Anpassung ist BARK (*Bean Automatic Reconfiguration Framework*). In [Rutherford u. a. 2002] wird die Rekonfiguration

von komponentenbasierten Anwendungen im *JavaBean Component Model* beschrieben. Zunächst wurde BARK als Deployment-Infrastruktur für *Java Enterprise Beans (EJB)* implementiert, in welchen die *Offline*-Installation von EJBs über das Internet unterstützt wird.
Eine Erweiterung von BARK ([Rutherford u. a. 2002] ermöglicht die dynamische Rekonfiguration der EJB-Anwendungen. Die Interaktion zwischen EJB-Komponenten wird über das *Java Naming and Directory Interface* (JNDI) realisiert, welche JNDI-Namen Referenzen auf EJBs zuordnet. Client-Komponenten benutzen einen eindeutigen Namen zur Suche von Komponenten bestimmter Funktionalität. Soll eine Komponente durch eine andere Komponente ersetzt werden (eine neue Version soll geladen werden), kann das Umschalten über den JNDI-Namen erfolgen, indem bei der nächsten Namensauflösung eine andere Referenz herausgegeben wird.
BARK definiert eine XML-basierte Skriptsprache, mit der Rekonfigurationen mit transaktionaler Semantik beschrieben werden können. Es können EJBs geladen werden, Verbindungen zwischen EJBs hergestellt und Abschnitte bestätigt (*commit*) werden. Für die Verbindung benutzt BARK eigene JNDI-Namen (z.B.: /BARK/w323423), definiert aber Zuordnungen auf die zuvor verwendeten JDNI-Namen, um Client-Komponenten zu integrieren, welche nicht von BARK verwaltet werden. Diese lösen die entsprechende Komponente über ihren zuvor bekannten JNDI-Namen auf.
Die Zusatzkosten für die notwendigen Namensauflösungen während der Anwendungslaufzeit sind gegenüber dem in dieser Arbeit vorgestellten Ansatz sehr hoch, da sie im Prinzip vor jeder Interaktion durchgeführt werden müssen, um somit die Rekonfiguration ggf. auslösen zu können.

Abgrenzung

Im Prinzip greift der in dieser Arbeit vorgestellte Ansatz für die Entwicklung adaptiver Anwendungen wesentliche Züge der architekturbasierten Adaption nach [Oreizy u. a. 1999] auf. Die Anpassung der Architektur komponentenbasierter Anwendungen stellt eine mächtige Möglichkeit dar, adaptive Anwendungen zu realisieren. Während [Oreizy u. a. 1999] die Vision der Architekturanpassung für asynchrone Kommunikationsmechanismen beschreibt, fokussiert die vorliegende Arbeit auf die Integration der architekturbasierten Adaption in moderne Komponentenplattformen und der effizienten Ausführung und Rekonfiguration von Anwendungen auf der Basis synchroner ORPC-basierter Kommunikation und ist damit in den weit verbreiteten Komponentenplattformen, der Java- und .NET-Plattform, einsetzbar.

8.1.5 Sprachbasierte Ansätze zur Adaption

In diesem Abschnitt sollen Forschungsarbeiten vorgestellt werden, die Spracherweiterungen für die Implementierung adaptionsspezifischer Belange verwenden. Die vorliegende Arbeit setzt domänenspezifische Sprachen für die Beschreibung der Konfiguration verteilter komponentenbasierter Anwendungen sowie der Definition von Profilen und der Beschreibung des Monitoring ein. Die Natur dieser Beschreibungsdokumente ist deklarativ und eine Analyse verwandter Ansätze erfolgte bereits bei der Beschreibung der XML-basierten Dokumente. Auch für die Markierung von Konfigurationspunkten in den Anwendungsklassen wurde ein neues Verfahren im Rahmen dieser Arbeit vorgestellt. Anstatt hierfür eine Spracherweiterung zu definieren, wurde das Konzept der Metadatenannotationen verwendet, um Attribute von Klassen als Parameter

bzw. Kommunikationsendpunkte zu deklarieren. Die im Rahmen dieses Abschnitts vorgestellten Ansätze führen neue Sprachelemente ein, um adaptionsspezifische Logik imperativ entwickeln zu können.

Adaptive Java

Im RAPIDware-Projekt [McKinley u. a. 2005] wurden der Entwurf und die Nutzung von adaptiven komponentenbasierten Middleware-Plattformen für den Schutz kritischer Infrastrukturen in dynamischen, heterogenen Einsatzumgebungen untersucht. Unter der Leitung von K. McKinley und K. Stirewalt sind unter anderem die Sprachерweiterung Adaptive Java und Trap/J (Abschnitt 8.1.6) entstanden.
Adaptive Java [Kasten u. a. 2002] erweitert die Reflexionsschnittstellen der Java-Umgebung durch die Einführung neuer Sprachelemente. Basiselemente in Adaptive Java sind *components* bzw. *adaptable classes*, denen drei verschiedene Schnittstellenarten zugeordnet werden können:

- *invocations* : für die Ausführung normaler Operationen (*computation*)
- *refractions* : für die Überwachung von internem Zustand (*intercession*)
- *transmutations* : für die Manipulation internen Zustands (*introspection*)

Vorhandene Java-Klassen können zunächst durch *absorption* in *base-level components* überführt werden (neues Schlüsselwort *absorbs*). *Base-level components* können veränderliche Methoden, *invocations*, zugewiesen werden, die das Verhalten der Originalklasse exportieren. Diese können während der Laufzeit von einer Metaebene hinzugefügt bzw. entfernt werden. In einem zweiten Schritt (*metafication*) können *refractions* und *transmutations* in der Metaebene erstellt werden, die auf den *base-level components* operieren. Anwendungskomponenten laufen auf der Basisebene und werden *base components* genannt.
Eine ausführliche Fallstudie für Adaptive Java existiert in Form der *MetaSockets* [Sadjadi u. a. 2006], einer Erweiterung von *Multicast-Sockets*. Mit Hilfe von Adaptive Java kann eine zur Laufzeit anpassbare Kette von Filterkomponenten in den Kommunikationsfluss zwischen Client- und Server-Anwendungen integriert werden.
Adaptive Java ist ein Java-Java-Compiler. Während der Übersetzungsphase werden entsprechende Java-Klassen erzeugt, welche die definierten Erweiterungen enthalten. Funktionale Aufrufe an Adaptive Java *components* werden über das Schlüsselwort *invoke* implementiert. Alle Aufrufe werden über eine Hash-Tabelle verarbeitet. Durch diese Indirektion können *invocations* während der Laufzeit ausgetauscht werden. Adaptives Verhalten muss in Adaptive Java imperativ definiert werden. Dabei müssen zunächst funktionale Klassen um Schnittstellen zur Manipulation konfigurationsspezifischer Daten sowie zur Beobachtung internen Zustands explizit erweitert werden. Während der Laufzeit müssen entsprechende Entscheidungsträger die Adaption mit Hilfe der geschaffenen Schnittstellen umsetzen.
Adaptive Java erlaubt die Erstellung konfigurierbarer Java-Klassen direkt auf der Sprachebene von Java, während in der vorliegenden Arbeit die Konfigurationspunkte der Klassen über Metadatenannotationen der funktionalen Klassen vorgenommen werden. Der Einsatz von Annotationen ist dabei einfacher zu verwenden als das Erlernen der neuen, von Adaptive Java eingeführten Schlüsselworte. Auch in Adaptive Java werden Teile der konfigurationsspezifischen Logik für den Zugriff auf Interna der

funktionalen Anwendung generiert. Im Gegensatz zur vorliegenden Arbeit muss allerdings die dynamische Rekonfiguration der Anwendung und auch die Beobachtung von Umgebungsparametern und das Treffen von Adaptionsentscheidungen explizit in der Metaebene implementiert werden.

Program Control Language - PCL

Die Program Control Language (PCL) [Ensink u. a. 2003] ist ein Framework zur Leistungsoptimierung. PCL benutzt statische Task-Graphen, um die Laufzeitbeziehungen von Anwendungstasks zu beschreiben. Kontrollflüsse, Kommunikation und Synchronisation werden dabei abstrakt durch Graphen beschrieben. Anpassungen werden im PCL-Framework durch das Hinzufügen (*AddTask*), Ersetzen (*ReplaceTask*) und Entfernen (*RemoveTask*) von Tasks sowie der Änderung von Parametern (*ChangeParam*) erreicht. Als ein Beispiel für Parameteranpassungen wird die Iterationstiefe bei der Berechnung eines Fraktals angegeben.
Eine adaptive Anwendung besteht aus einem PCL-Steuerungsprogramm, welches *Metriken* einer Zielanwendung auswertet und entsprechende zuvor genannte Verfahren zur Adaption der Anwendung auslöst. Metriken können mit Hilfe von PCL-Konstrukten definiert werden. Sie beinhalten Anweisungen für den Compiler zum Instrumentieren des Zielcodes, um dort entsprechende Performancemaße ableiten zu können. Weiterhin können Ereignisse definiert werden, welche bei Veränderung definierter Werte ausgelöst werden. In PCL implementierte Ereignisbehandlungsroutinen können mit Hilfe von Adaptionsanweisungen die Anwendung durch die zuvor beschriebenen Schritte anpassen.
Um die Korrektheit der Adaption überprüfen zu können, schlagen die Autoren Adaptionspolitiken vor. Zunächst können bestimmte Bereiche eines Task-Graphen als Regionen zusammengefasst werden, für die dann Bedingungen an die Ausführung von Adationsoperationen definiert werden können. Als ein Beispiel wird die Blockierung des Eintretens neuer Aktivität im Task-Graphen in eine Region gefordert (*DeferAndBlock(region)*).
Mit PCL können Anpassungen an verteilten Anwendungen vorgenommen werden, um z.B. auf die Verfügbarkeit neuer Prozessoren reagieren zu können. Das PCL-Framework legt dabei die Grundlage für eine Reihe notwendiger Aktivitäten wie das Monitoring, die Beobachtung von Zieldienstgüteparametern (Metriken) und bietet konzeptionell (Adaptionspolitiken wurden nicht implementiert) Unterstützung bei den Anpassungsoperationen. Große Teile der Adaptionslogik müssen aber vom Anwendungsprogrammierer entwickelt werden. Während die vorliegende Arbeit adaptive Anwendungen auf der Basis von Softwarekomponenten realisiert, die implizit die Verwaltung von Kontrollflüssen beinhaltet, arbeitet PCL direkt auf der Ebene von Tasks.

ADAPT

Das ADAPT Framework der Purdue University [Voss und Eigemann 2001] führt Optimierungen von Anwendungen während ihrer Laufzeit durch. Dabei werden durch einen Compiler während der Laufzeit ständig neue Programmversionen generiert, welche dann ausgeführt werden und definierte Leistungswerte mit der bisher besten Lösung verglichen werden. Die Optimierungslogik kann dabei auf einem vom Zielsystem unabhängigen Rechner durchgeführt werden. Der Anwendungsentwickler spezifiziert die Optimierungslogik in einer eigenen Sprache, der ADAPT Language (AL). Er kann

dabei auch auf gemessene Leistungswerte der Laufzeitumgebung zurückgreifen und die nächsten Testversionen bestimmen. Aus dieser C-ähnlichen Sprache wird der eigentliche Optimierungscode generiert. Innerhalb der Sprache kann auf Schlüsselwörter zurückgegriffen werden, die während des Optimierungsvorgangs ausgewertet werden (z.B. time: Ausführungszeit, l2_cache: Größe des L2-Caches der CPU). Mit Hilfe des Schlüsselwortes *collect* kann die Evaluierung eines neuen Codeabschnittes (Programmversion) ausgelöst werden, um in einem nächsten Schritt die gemessenen Leistungswerte zu berücksichtigen. Das ADAPT-Framework erlaubt somit die Anpassung von Anwendungslogik an die Gegebenheiten der unterliegenden Hardware (z.B. Cache-Größe).

8.1.6 Adaption mit aspektorientierter Programmierung

Während der Entstehungsphase dieser Arbeit sind eine Reihe von Ansätzen entwickelt worden, welche die aspektorientierte Programmierung für die Integration adaptiven Verhaltens benutzen. Im Wesentlichen handelt es sich dabei um Aspektweber, welche die Aktivierung von Aspekten sowie die dynamische Aktualisierung von Aspektimplementierungen während der Laufzeit erlauben. In diesen Ansätzen steht allerdings meistens die Implementierung des Aspektwebers im Vordergrund. Die Realisierung adaptiven Verhaltens wird oft nur am Rande behandelt. Dabei wird argumentiert, dass die Entwicklung von adaptiven Anwendungen durch dynamisches Aspektweben ermöglicht wird, weshalb einige Vertreter dieser Ansätze im Rahmen dieses Abschnitts vorgestellt werden sollen.

Trap/J

Trap/J [Sadjadi u. a. 2004] ist ein Generatorframework für Adaptive Java zur Erstellung adaptiver Anwendungen ohne Quellcodeverfügbarkeit. Mit Hilfe des Aspektwebers AspectJ wird für jede Komponente eine Proxy-Klasse erstellt, innerhalb derer Methodenaufrufe auf alternative Methoden in einer Metaebene umgelenkt werden. Alternative Methoden werden über installierbare Delegates konfiguriert, die von einem nachladbaren Objekt einer Delegate-Ebene (*delegate level*) implementiert werden.

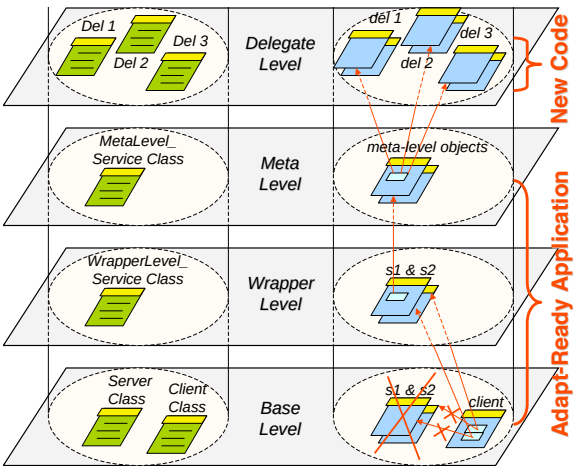


Abbildung 8.5: Trap/J Architektur [Sadjadi u. a. 2004]

Abbildung 8.5 zeigt die Ebenen des Laufzeitmodells von Trap/J. Referenzen in der Basisebene (*base level*) werden zunächst auf die *Wrapper*-Ebene (*wapper level*) um-

gelenkt. Innerhalb der *Wrapper*-Ebene können alternative Methoden in der Delegate-Ebene aufgerufen werden, die in der Metaebene während der Laufzeit konfiguriert werden können. Die Generierung von Anwendungen ist zweistufig. Während der Übersetzungsphase werden Anwendungskomponenten *adapt-ready*, also bereit für die Adaption gemacht. Dabei werden *Wrapper*- und Metaebenen-Komponenten erzeugt. Während der Laufzeit kann adaptives Verhalten durch dynamisches Nachladen von Objekten in der Delegate-Ebene und die Auswahl alternativer Methoden realisiert werden.

In der Metaebene werden Aufrufe in die Delegate-Ebene mit Hilfe der Java-Reflexionsmethoden implementiert. Dabei werden zunächst entsprechende Methoden in der Delegate-Ebene mit Hilfe struktureller Reflexion (*class.getDeclaredMethod*) gefunden, die durch *Method.invoke* aufgerufen werden. Obwohl eine performante Implementierung kein Entwurfsziel von Trap/J war, liegt hier ein wesentlicher Nachteil von Trap/J gegenüber der in dieser Arbeit vorgestellten Implementierung. Betrachtet man nur die Zusatzkosten für Methodenaufrufe durch die Reflexions-Methode *Method.invoke*, dann benötigt dieser gegenüber einem virtuellen Methodenaufruf signifikant mehr Zeit [Forax u. a. 2005]. Diese Zusatzkosten müssen bei jedem Methodenaufruf einberechnet werden.

Ein weiterer entscheidender Unterschied gegenüber der vorliegenden Arbeit ist die Implementierung von adaptivem Verhalten in einer separaten Delegate-Ebene. Hierbei wird die Anwendungslogik wiederum über mehrere Komponenten verteilt und dies führt zur Erhöhung der Komplexität des Softwareentwicklungsprozesses. Demgegenüber können mit dem in dieser Arbeit vorgestellten Ansatz Komponenten mit angepasstem Verhalten als Ganzes entwickelt werden.

Iguana/J

Einen weiteren Ansatz zur Integration adaptiver Logik während der Laufzeit wurde im Iguana/J-Projekt [Redmond und Cahill 2002] implementiert. Während Anwendungen in einer Basis-Ebene ausgeführt werden, können Methoden der Originalklassen in einer Metaebene überschrieben werden. Alternative Logik wird als Ableitung der Originalklasse implementiert und durch die Definition eines Metaobjektprotokolls (MOP) an die Anwendungsobjekte gebunden. MOPs können statisch als textuelle Beschreibung vorliegen, aber auch programmatisch während der Anwendungslaufzeit definiert werden. Die Integration der Metaobjekte kann über sieben verschiedene Verknüpfungspunkte erfolgen, darunter die Objekterzeugung, Methodenaufrufe und Feldzugriffe. MOPs können komponiert werden, wobei auch mehrere Metaobjekte an eine Aktion gebunden werden können, die verkettet ausgeführt werden. Die Aufrufkette wird über den Aufruf von *proceed()* in den Metaobjekten weitergeleitet.

Iguana/J nutzt das *JIT Compiler Interface* der Sun Java Virtual Maschine, um entsprechende Änderungen (Aufrufe in die Metaebene) in der Basisanwendung zu installieren. Ein Nachteil bei diesem Verfahren ist die Abhängigkeit der implementierten Mechanismen von der eingesetzten virtuellen Maschine.

Der Iguana/J-Ansatz ähnelt sehr der aspektorientierten Programmierung und hat die gleiche Mächtigkeit. Wie schon in einigen anderen vorgestellten Arbeiten wird auch in Iguana/J alternativer Anwendungscode in separaten Klassen implementiert. Adaptivität muss durch die Änderung von Methodenaufrufen (vgl. *Interceptor* Muster) in der Basisanwendung erfolgen.

PROSE

PROSE (*PROgrammable extenSions of sErvices*) [Popovici u. a. 2002] ist ein dynamischer Aspektweber, der die Verwebung von Aspekten zur Lade- und Laufzeit von Anwendungen unterstützt. Innerhalb einer Aspektimplementierung können existierende Methoden erweitert (*before*, *after*, *advice*), Methoden redefiniert, der Zugriff auf Felder manipuliert sowie Ausnahmebehandlungen angepasst werden.

Die PROSE-Architektur enthält zwei wesentliche Bestandteile. Die *AOP-Engine* verwaltet Aspektimplementierungen sowie Verwebungspunkte. Der *Execution-Monitor* webt Aspekte in die funktionale Anwendungslogik. In PROSE wurden drei verschiedene Strategien für die Verwebung von Aspektcode und Anwendungslogik untersucht. Zunächst wurden Verwebungspunkte mit Hilfe der *Java Virtual Machine Debugger* Schnittstelle integriert. In einer zweiten Implementierung wurde der JIT-Compiler erweitert, um Verwebungspunkte im generierten (nativen) Code zu integrieren. Schlussendlich benutzt die aktuellste Implementierung eine erweiterte virtuelle Maschine. In die Jikes RVM [Alpern u. a. 2005] wurden Methoden zur Manipulation des Bytecodes von Java-Klassen während der Laufzeit integriert, mit denen der Bytecode einer Methode überschrieben werden kann. Mit Hilfe dieser Methoden können neue Verwebungspunkte während der Laufzeit integriert werden.

Adaptives Verhalten kann mit PROSE ähnlich wie in der im Folgenden beschriebenen *Dynamic Service Adaptation* durch die Aktivierung von Aspekten während der Laufzeit einer Anwendung erfolgen. Wie auch in dem in dieser Arbeit eingesetzten Aspektweber Rapier-Loom.Net [Schult und Polze 2002] werden in PROSE Aspekte in Java und nicht in einer separaten Sprache wie in AspectJ [Kiczales u. a. 2001] definiert. Wie auch im Iguana/J-Projekt ist bei PROSE die Erweiterung der virtuellen Maschine problematisch für die Portabilität der entwickelten adaptionsspezifischen Logik.

Dynamic Service Adaptation

Ein weiterer Ansatz, adaptives Verhalten mit Hilfe aspektorientierter Programmierung zu erreichen, wird in [Hirschfeld und Kawamura 2006] beschrieben. Die virtuelle Maschine von Smalltalk/Squeak stellt Mechanismen zur Laufzeit-Reflexion sowie später Bindung zur Verfügung. Diese Mechanismen werden genutzt, um den dynamischen Aspektweber *AspectS* zu implementieren. Neues/adaptives Verhalten kann über Aspekte in existierende Anwendungen integriert werden, ohne deren Quellcode zu verändern. Neues Verhalten wird so z.B. im Aspekt vor der Ausführung einer Originalmethode implementiert. Der *AspectS*-Aufsatz *PerspectiveS* koordiniert die kontextabhängige Aktivierung von angepasster Logik, also die bedingte Ausführung von Aspektcode in Abhängigkeit vom Ausführungskontext.

Ein Nachteil des vorgestellten Ansatzes ist die eingeschränkte Performance durch die Verwendung einer dynamischen Sprache. Der Ansatz bietet dennoch die Möglichkeit, sehr elegant existierende Anwendungen auch während der Laufzeit zu erweitern, ohne dass die Änderungen vor der Laufzeit bekannt sein müssen. Bei komplexen Anpassungen muss allerdings beachtet werden, dass durch die Realisierung der Adaptionslogik als Aspekt zusätzlicher Aufwand bei der Softwareentwicklung besteht, da die vorhandene Logik der aktiven Komponenten nicht geändert werden kann, sondern nur Ein- und Ausgabeparameter von Methodenaufrufen manipuliert werden können. Da sich dieser Ansatz aber sehr gut für die Integration von Adaptionslogik in bestimmten

Fällen eignet, wurde auch im Rahmen dieser Arbeit ein Mechanismus zur dynamischen Aspektaktivierung und -rekonfiguration in die Ausführungsplattform integriert.

8.1.7 Adaption in Betriebssystemen

Auch die Integration von Adaptionsmechanismen in Betriebssysteme wurde von einigen Forschern untersucht. Vor allem die Konfiguration von Betriebssystemen im Bereich der eingebetteten Systeme spielt eine wichtige Rolle bei der Anpassung an eingeschränkte Ressourcen und vielfältige Hardware-Plattformen.

PURE und eCOS

PURE [Beuche u. a. 1999] und eCOS [Cygnus 1999] unterstützen die Auswahl und Konfiguration von Modulen mit Hilfe graphischer Werkzeuge. Dabei können vor der Übersetzungsphase Scheduling-Strategien, zu integrierende Treiber und weitere betriebssystemspezifische Details konfiguriert werden. PURE benutzt für die Konfiguration auch Werkzeuge der aspektorientierten Programmierung, während eCOS Präprozessor-Direktiven für die quellcodebasierte Konfiguration verwendet. PURE und eCOS unterstützen keine Rekonfiguration von Modulen während der Laufzeit.

K42 Betriebssystem

K42 [Silva u. a. 2006] ist ein Mikrokernbetriebssystem, welches von IBM entwickelt wird. Es ist quelloffen und Linux ABI (*Application Binary Interface*) und POSIX-API konform. Betriebssystemkomponenten können in K42 während der Laufzeit rekonfiguriert werden. Zum einen können durch *Interposition* neue Funktionen durch die Integration eines *Wrappers* installiert werden. Zum anderen können per *Hot-swapping* Komponentenimplementierungen ausgetauscht werden. Für die Rekonfiguration wird eine einzelne Komponente in den Zustand *quiescent* überführt. Hierfür wird ein Generationenzähler der Betriebssystemthreads benutzt. Bei der Erzeugung neuer Threads wird dieser Zähler in den existierenden Threads angepasst. Der *quiescent* Zustand wird erreicht, indem auf die Beendigung aller Threads gewartet wird, die einen Generationenzähler besitzen, der kleiner ist als der höchste beim Auslösen der Rekonfiguration. Dieser Algorithmus setzt die schnelle Beendigung von Betriebssystemthreads voraus, da die Rekonfiguration sonst nicht ohne große Unterbrechungen durchgeführt werden kann. Diese werden in K42 auf Grund seiner Architektur als gegeben vorausgesetzt, stellen aber für einige existierende Programmierschnittstellen Einschränkungen dar.

8.2 Dynamische Rekonfiguration

Nach der Vorstellung verwandter Arbeiten auf dem Gebiet der Entwicklung adaptiver Software soll im Folgenden ein Teilaspekt adaptiver Anwendungen, die dynamische Rekonfiguration, die einen Schwerpunkt der vorliegenden Arbeit darstellt, betrachtet werden. Dabei wird an Hand der historischen Entwicklung der dynamischen Rekonfigurationsalgorithmen der im Rahmen dieser Arbeit entwickelte Ansatz mit den verwandten Ansätzen verglichen.

8.2.1 Der knotenpassivierende Ansatz

Die Forschung auf dem Gebiet der dynamischen Rekonfiguration wurde wesentlich von Kramer und Magee im Rahmen der Conic-Plattform beeinflusst. Ihre grundlegenden Ansätze wurden von zahlreichen nachfolgenden Arbeiten aufgegriffen und verbessert. Der initiale Algorithmus von Kramer und Magee soll im Folgenden vorgestellt werden. Beim knotenpassivierenden Ansatz [Kramer und Magee 1990] wird ein rekonfigurierbarer Zustand einer Anwendung durch das Passivieren von Anwendungsbestandteilen, die allgemein als Komponenten bezeichnet werden, erreicht. In Abwesenheit abhängiger Transaktionen gilt eine Komponente als passiv, wenn sie in keine von ihr initiierte Transaktion involviert ist und keine neuen Transaktionen initiiert. Eine Komponente erreicht den passiven Zustand in endlicher Zeit, da sie nur auf das Ende aller ihrer aktiven Transaktionen warten muss, und dies geschieht laut Voraussetzung in endlicher Zeit. Eine Komponente befindet sich im Zustand untätig (*quiescent*), wenn alle Komponenten mit Verbindungen zu dieser auch im Zustand passiv sind. Die Änderungen während einer Rekonfiguration bestimmen die Komponenten, die während der Rekonfiguration untätig sein müssen, um die Konsistenz der Anwendung gewährleisten zu können.
Im Falle abhängiger Transaktionen muss der beschriebene Algorithmus erweitert werden. Im generalisierten passiven Zustand (*generalised passive state*) ist eine Komponente nicht in nachfolgende Transaktionen verwickelt, die von ihr initialisiert wurden, und sie initialisiert von selbst keine neuen Transaktionen. Die Komponente verarbeitet aber weiterhin Transaktionen und initiiert ggf. abhängige Transaktionen, um eingehende Anfragen verarbeiten zu können. Im erweiterten passiven Zustand (*extended passive state*) befinden sich Komponenten im generalisierten passiven Zustand, wenn auch alle Komponenten, die abhängige Transaktionen an ihr initiieren können, in diesem Zustand sind.
Ein Nachteil dieses Ansatzes ist, dass der Entwickler komplexe Rekonfigurationslogik in die Komponentenimplementierung integrieren muss, um das beschriebene Verhalten beim Passivieren von Komponenten zu erreichen. Vor allem muss beim Passivieren serverseitig entschieden werden, ob eine Transaktion verarbeitet werden darf bzw. neue Transaktionen initiiert werden können oder sie blockiert werden müssen. In vielen Laufzeitsystemen, die CLI und die Java-Plattform eingeschlossen, ist eine solche Entscheidung nicht möglich, da hier Kontextinformationen des Rufers benötigt werden würden.

8.2.2 Der knotenblockierende Ansatz

Moazami-Goudarzi [Moazami-Goudarzi 1999] entwickelte einen Ansatz, der darauf basiert, dass sich die Komponenten nur während einer Transaktion in einem inkonsistenten Zustand befinden. Eine Komponente ist dann im Zustand untätig, wenn sie in keine Transaktion verwickelt ist. Unter der Annahme, dass Transaktionen nicht verschachtelt werden, also eine Komponente zu einer Zeit immer nur in eine Transaktion involviert ist, kann eine Anwendung mit diesem Ansatz in einen rekonfigurierbaren Zustand überführt werden. Dabei werden alle involvierten Komponenten blockiert, indem eine Blockierungsnachricht übermittelt wird und nach Beendigung der aktiven Transaktion der Zustand blockiert übermittelt wird. Es kann potentiell vorkommen, dass Komponenten kurzzeitig entblockt werden müssen, um abhängende Komponenten blockieren zu können.

Die Zusatzkosten durch Blockierungsnachrichten sind bei diesem Algorithmus hoch, da Komponenten potentiell mehrere Male entblockt werden müssen. Die Konfigurationsschnittstellen für die Komponenten müssen durch den Anwendungsentwickler implementiert werden. Anwendungen mit multiplen Threads werden von diesem Ansatz nicht unterstützt.

8.2.3 Der verbindungsblockierende Ansatz

Wermelinger [Wermelinger 1997] verbesserte den Algorithmus von Kramer/Magee bzgl. der minimalen Unterbrechungszeit. Zunächst führt er Ports als Kommunikationspunkte von Komponenten in das Anwendungsmodell ein. Er unterscheidet *initiator*- und *recipient*-Ports, welche die Schnittstelle einer Komponente definieren. Transaktionen werden über *initiator*-Ports initiiert und über *recipient*-Ports verarbeitet. Abhängigkeiten von Transaktionen werden über Portabhängigkeiten beschrieben. Eine Verbindung wird durch einen *initiator*-Port und einen *recipient*-Port definiert.
Im Gegensatz zum Ansatz von Kramer und Magge, bei dem für die Berechnung der zu passivierenden Knoten nur die Verbindungsstruktur betrachtet wird, unterscheidet Wermelinger individuelle Transaktionen über diese Verbindungen. Während Verbindungen statische Strukturen zwischen Komponenten beschreiben, existieren Transaktionen zur Laufzeit. Es können mehrere Verbindungen mit einem Port verknüpft werden, aber immer nur eine Transaktion pro Port-Paar. Zur Vermeidung von Verklemmungen wird eine azyklische Verbindungsstruktur vorausgesetzt.
Wermelingers zentrale Idee ist, dass nur Verbindungen blockiert werden, die bei der Rekonfiguration entfernt werden. Eine Verbindung kann blockiert werden, indem in den initiierenden Knoten auf die Beendigung aktiver Transaktionen über diese Verbindung gewartet wird und keine neuen initiiert werden. Im Falle abhängiger Transaktionen muss die Blockierung von Verbindungen geordnet werden. Die Verbindung V_1, über die eine Transaktion t_1 läuft, die von einer Transaktion t_2 abhängt, die über eine Verbindung V_2 läuft, darf nicht vor dieser blockiert werden. Um das Erreichen des rekonfigurierbaren Zustands zu sichern, muss der Anwendungsgraph azyklisch sein.
Wermelingers Ansatz erfordert die dynamische Aufzeichnung von Transaktionen zur Laufzeit. Dabei müssen innerhalb einer Komponente Abhängigkeiten zwischen Transaktion über Port-Beziehungen durch den Programmierer verwaltet werden. Bei vorhandener Abhängigkeit wird einem In-Port ein entsprechender Out-Port je Transaktion zugeordnet.

8.2.4 Dynamische Rekonfiguration in CORBA I

[Bidan u. a. 1998] beschreiben einen „dynamischen Rekonfigurationsdienst“ (*Dynamic Reconfiguration Service*) für CORBA. Die Konsistenz der Anwendung wird durch die Sicherstellung der Integrität von Fernaufrufen (ORPCs) realisiert. Das heißt, eine Rekonfigurationsaktion darf keine initiierten Fernaufrufe unbearbeitet lassen. Bidan et al. erweitern die *LifeCycle Common Object Services* um die Fähigkeit zur dynamischen Rekonfiguration. Auch dieser Ansatz basiert auf dem Passivieren von Verbindungen. Die Autoren argumentieren, dass abhängige ORPCs entsprechend ihrer Reihenfolge blockiert werden müssten; ob und wie dies implementiert wurde, ist jedoch nicht veröffentlicht worden. Da auch bei diesem Ansatz die Reihenfolge der Blockierungen berücksichtigt werden muss, können keine zyklischen Aufrufe zwischen den CORBA-Objekten unterstützt werden.

8.2.5 Dynamische Rekonfiguration in CORBA II

Almeida und Wegdam (*University of Twente*) diskutieren eine andere Erweiterung der CORBA-Middleware [Almeida u. a. 2001; Wegdam 2003]. Die Implementierung unterstützt die dynamische Rekonfiguration nach einem ähnlichen Verfahren wie Kramer/Magee. Durch die Weitergabe impliziter Parameter entlang der Kommunikationspfade können abhängige Aufrufe leicht verarbeitet werden. Die Middleware-Infrastrukur fügt jedem Aufruf die Identifikation der rufenden Komponente hinzu. Mit Hilfe dieser Informationen kann ein rekonfigurierbarer Zustand erreicht werden.
Aufrufe werden dabei in zwei Kategorien eingeordnet. Aufrufe der *laissez-passer*-Kategorie (franz. passieren lassen) werden verarbeitet, während die anderen Aufrufe in eine Warteschlange einsortiert werden. Dieses Verfahren wird als *selective queuing* bezeichnet. Die Blockierung eines Aufrufs lässt sich dabei auf Client bzw. auf Serverseite implementieren.
Der Laufzeit-*Overhead* für dieses Verfahren ist höher, da sich die Kontextinformationen mit der Aufruftiefe vergrößern, aber es können Anwendungen mit re-entranten Komponenten und zyklischen Aufrufbeziehungen zwischen den Komponenten unterstützt werden. Auch der Overhead für einzelne Methodenaufrufe während der Rekonfiguration ist erhöht, da immer der gesamte Kontext traversiert werden muss. Die Komplexität der Zusatzkosten von Methodenaufrufen in Abhängigkeit von der Aufruftiefe wurde mit dem im Rahmen dieser Arbeit entwickelten Verfahren ReDAC auf O(1) gegenüber O(n) bei Almeida und Wegdam verbessert.
Almeida und Wegdam fordern zusätzlich, dass keine Synchronisation zwischen den Anwendungsthreads über geschachtelte Aufrufe einer re-entranten Komponente hinweg besteht, also vor einem ausgehenden Ruf alle Synchronisationsobjekte freigegeben werden. Diese Einschränkung ist durch den in dieser Arbeit vorgestellten Rekonfigurationsalgorithmus nicht gegeben. Zusätzlich hat der im Rahmen dieser Arbeit vorgestellte Algorithmus den Vorteil, dass bei der Terminierung eines Threads durch eine unbehandelte strukturierte Ausnahme keine für die Rekonfiguration benötigten Informationen verloren gehen, wie dies bei Almeida und Wegdam der Fall wäre, da die Informationen mit dem Aufrufkontext mitgeführt werden. Im Algorithmus dieser Arbeit können diese Threads aus den jeweiligen Tabellen der Kapseln leicht ausgetragen werden. Ein weiterer Nachteil der von Almeida und Wegdam präsentierten Implementierung ist, dass alle Aufrufe zwischen den Objekten über die CORBA Middleware realisiert werden müssen, währenddessen die in dieser Arbeit vorgestellte Implementierung auch effiziente lokale Methodenaufrufe unterstützt.

8.2.6 Rekonfiguration in POLYLITH

Eine grundlegende Arbeit über dynamische Rekonfiguration verteilter Anwendungen wurde von Christine R. Hofmeister et al. verfasst. In [Hofmeister und Purtilo 1993; Hofmeister 1994] wird ein Framework zur Entwicklung dynamisch rekonfigurierbarer Anwendungen beschrieben. Basierend auf der POLYLITH-Software-Bus-Middleware wird unter Verwendung von Algorithmen von Kramer/Magee eine Bibliothek zur Rekonfiguration verteilter Prozesse in heterogenen Systemen implementiert. Mögliche Rekonfigurationsformen umfassen dabei den Austausch von Modulimplementierungen (*Updates*), die Änderung der Struktur (Verbindung der Prozesse) sowie die Änderung der Geometrie (Zuordnung von Prozessen zu Ausführungsorten - Migration). Für den Transfer von Zustandsinformationen zwischen Rechnergrenzen wird eine abstrakte Zu-

standsrepräsentation in Form von abstrakten Datentypen (ADT) eingeführt. Diese basiert auf einer Arbeit von [Herlihy und Liskov 1982]. Sämtliche Rekonfigurationsschritte müssen in einer Art Rekonfigurationsskript explizit formuliert werden. Dazu gehören das Entfernen/Hinzufügen von Verbindungen, das Erzeugen/Entfernen von Modulen, der Zustandstransfer sowie weitere Rekonfigurationsdetails. Der potentielle Zustandstransfer muss während der Modulimplementierung durch die Unterstützung entsprechender Schnittstellen integriert werden. Nachteile dieses Ansatzes sind die statische Integration der rekonfigurationsspezifischen Logik, welche vom Anwendungsentwickler realisiert werden muss. Die busbasierte Kommunikation stellt den Entwickler vor die Herausforderungen, dieses Paradigma zunächst erlernen zu müssen. Der direkte Zugriff auf plattformspezifischen Zustand (Heap und Stack) erschwert die Portabilität der entwickelten Ansätze, da eine genaue Systemkenntnis notwendig ist.
Nach der Vorstellung verwandter Arbeiten auf dem Gebiet der dynamischen Rekonfiguration im Allgemeinen sollen im Folgenden zwei Ansätze für die Ausführung rekonfigurierbarer Anwendungen im Umfeld eingebetteter Echtzeitsysteme vorgestellt werden. In diesem Umfeld werden zusätzliche Anforderungen an die dynamische Rekonfiguration gestellt. Neben dem Erhalt der Konsistenz ist die Einhaltung von Zeitschranken (*deadlines*) oberstes Gebot.

8.2.7 Rekonfiguration in OSA+

OSA+ [Schneider u. a. 2004] ist eine dienstorientierte Middleware, die an der Universität Karlsruhe entwickelt wurde. In OSA+ kommunizieren Dienste über *order*- und *result*-Nachrichten. OSA+ unterstützt die vorhersagbare Ausführung von Diensten und deren dynamische Rekonfiguration unter Einhaltung vorgegebener Zeitschranken. Eine dynamische Rekonfiguration wird von einem dedizierten Rekonfigurationsdienst über eine Nachricht ausgelöst, die wie alle anderen Nachrichten in OSA+ eine Priorität besitzt, mit der die Wichtigkeit einer Rekonfiguration festgelegt werden kann. Bei einer dynamischen Rekonfiguration, bei der ein Dienst durch einen anderen ersetzt wird, kann eine maximale Unterbrechungszeit spezifiziert werden. Für den Zustandstransfer wurden verschiedene Strategien implementiert, die den *Trade-Off* zwischen Rekonfigurations- und Unterbrechungsdauer verschieden realisieren. Beim *Full-Blocking Approach* wird der alte Dienst während des Zustandstransfers angehalten (keine Nachrichten von der Middleware an ihn gesendet) und anschließend der neue Dienst, durch eine spezielle Nachricht (*switch*) an die Middleware, aktiviert. Die Rekonfigurationszeit ist in diesem Fall optimal. Beim *Non-Blocking Approach* erfolgt der Zustandstranfer inkrementell. Während der Rekonfigurationsphase kann der alte Dienst weiter Anforderungen verarbeiten. Ändert sich hierdurch der transferierte Zustand, werden Teile erneut übertragen. Durch die Angabe einer maximalen Blockierungszeit wird die Zeit für den Zustanstransfer begrenzt, es kann allerdings vorkommen, dass die Rekonfigurationszeit sehr lange dauert.

8.2.8 Rekonfiguration mit Port-based Objects

Port-based Objects (PBO)s [Stewart u. a. 1993] nutzen ein Kommunikationparadigma, das eine einfache Austauschbarkeit von Softwaremodulen ermöglicht. Jedes PBO besitzt einen eigenständigen zyklischen Kontrollfluss. Die Kommunikation zwischen PBOs erfolgt über Zustandsvariablen (*State Variables*), welche sich in lokalen und globalen Tabellen in einem *Shared-Memory*-Bereich befinden. PBOs arbeiten auf lokalen

Kopien ihrer Zustandsvariablen. In jedem Zyklus werden Daten am Eingabe-Port gelesen und die entsprechenden Variablen des Ausgangs-Ports beschrieben. Eine in einer Konfigurationsdatei definierte Frequenz gibt die Aktualisierungshäufigkeit zwischen lokalem und globalem Zustand vor. Mit Hilfe eines Blocktransfers, welcher innerhalb einer kritischen Sektion ausgeführt wird, wird die Zustandssynchronisation realisiert.
Die Architektur der PBOs erlaubt den Austausch von Komponenten, da durch die Benutzung des Port-Konzepts eine lose Kopplung zwischen den einzelnen Elementen erreicht wird. Zwischen der zyklischen Verarbeitung kann die Rekonfiguration einer Anwendung durch Änderungen an der Verknüpfung zwischen den Ports der PBOs erfolgen. Im Rahmen der Masterarbeit von Helge Issel [Issel 2006] wurde der Einsatz von PBOs für das „Hau den Lukas“-Experiment untersucht. Dabei wurden Methoden zur automatischen Generierung von PBOs evaluiert und Adaptionstechniken integriert, die über die Überwachung von Port-Variablen realisiert wurde. Zusätzlich wurde eine XML-basierte Beschreibungssprache für PBO-Konfigurationen entwickelt und eine entsprechende Laufzeitumgebung implementiert.
Die Kommunikation von OSA+ und den PBOs basiert jeweils auf einem aktiven Mediator, der Informationen zwischen den Anwendungsmodulen transferiert. Die vorliegende Arbeit beruht auf der Verwendung direkter ORPC-basierter Kommunikation, wie sie in modernen Komponentenplattformen Verwendung findet.

8.3 Zusammenfassung

Im Rahmen dieses Kapitels wurden verwandte Arbeiten auf dem Gebiet der adaptiven Anwendungen sowie der dynamischen Rekonfiguration beschrieben und die im Rahmen dieser Arbeit entwickelten Techniken abgegrenzt. Forschungsarbeiten auf dem Gebiet der adaptiven Software wurden kategorisiert und wichtige Vertreter der einzelnen Gruppen vorgestellt. Erste Lösungsansätze für die Entwicklung adaptiver Anwendungen fanden sich im Bereich der Middleware-basierten Lösungen (Abschnitt 8.1.2) und der datenorientierten Adaption (Abschnitt 8.1.3), bei denen die möglichen Adaptionsstrategien eingeschränkt sind. Erst mit der architekturbasierten Adaption (Abschnitt 8.1.4) ist es möglich, Änderungen auch auf Basis der Anwendungskomponenten und deren Verbindungsstruktur vornehmen zu können und damit die Komponententechnologie einzusetzen. Die vorliegende Arbeit greift diesen Ansatz auf und schafft durch die Integration der architekturbasierten Adaption in moderne Komponentenplattformen eine Basis für ein neues Programmiermodell für adaptive Anwendungen. Durch den Einsatz der aspektorientierten Programmierung können signifikante Anteile der adaptionsspezifischen Logik generiert werden. Auch andere Ansätze verwenden die Konzepte der aspektorientierten Programmierung (Abschnitt 8.1.6), fordern allerdings die Implementierung der Adaptionslogik in einer separaten Metaebene, was durch den Einsatz von Reflexion für den Aufruf von Methoden (Trap/J) Performance-Nachteile mit sich bringt bzw. durch die Implementierung adaptiven Verhaltens innerhalb von Aspekten die Möglichkeiten der Anpassung auf die Manipulation von Methodenparametern begrenzt. Auch der in der Arbeit entwickelte Rekonfigurationsalgorithmus unterstützt gegenüber den verwandten Arbeiten verteilte Anwendungen mit multiplen Threads und re-entranten Komponenten. Ein ähnlicher Ansatz von Almeida und Wegdam verursacht durch die Übergabe impliziter Parameter im Aufrufkontext sowie durch den Einsatz der CORBA-Middleware für jeden Methodenaufruf höhere Zusatzkosten bei der Ausführung.

9 Zusammenfassung und Ausblick

Die vorliegende Arbeit stellt einen ganzheitlichen Ansatz für die Entwicklung und Ausführung adaptiver komponentenbasierter Anwendungen vor. Mit Hilfe bekannter Konzepte aus der Softwarearchitektur, in der die Struktur von Software durch Komponenten und Konnektoren dargestellt wird, konnte der Einsatz der komponentenorientierten Softwareentwicklung zur Anpassung von Anwendungen an wechselnde Umgebungsparameter und Ressourcenverfügbarkeiten durch die Konzeption und Entwicklung entsprechender Techniken für moderne Komponentenplattformen ermöglicht werden. Durch die Ausdehnung der Komponentenabstraktion auf die Anwendungslaufzeit ist die Aktivierung von Alternativen - einer Grundvorausetzung die für Adaption - auf Basis der Komponenten und der Softwarestruktur möglich. Erst die Komponentenorientierung schafft durch ihre Konzepte eine Ausgangsbasis für alternative Bausteine, die durch unterschiedliche nichtfunktionale Eigenschaften die Adaption ermöglichen. Durch die in dieser Arbeit entwickelte Ausführungsplattform auf Basis der .NET- und Java-Komponentenplattform können adaptive Anwendungen mittels dynamischer Rekonfiguration der Anwendungsstruktur und der Komponenten realisiert werden.

Zusammengefasst sind die wichtigsten Beiträge der Arbeit ein verbesserter Rekonfigurationsalgorithmus, der zyklische Verbindungsstrukturen und Anwendungen mit multiplen Threads unterstützt sowie ein neuartiger Ansatz zur dynamischen Aktualisierung von Softwarekomponenten zur Anwendungslaufzeit und die umfangreiche Werkzeugunterstützung unter Einsatz der aspektorientierten Programmierung, die zusammen ein effizientes Entwickeln und Ausführen adaptiver Software ermöglichen. Auch die Verwendung der entwickelten Techniken zur sicheren Ausführung von Steuerungsprogrammen im *Distributed Control Lab* ist an dieser Stelle hervorzuheben.

Nach der Beschreibung der verwendeten Terminologie, einem Abriss der geschichtlichen Entwicklung adaptiver Software und der Vorstellung grundlegender Basistechnologien in Kapitel 2 wird im zentralen 3. Kapitel der Arbeit ein Modell einer Ausführungsplattform für adaptive Anwendungen vorgestellt, welches die notwendigen Konzepte aktueller Laufzeitumgebungen von Komponentenplattformen wie der Java- oder der .NET-Plattform formalisiert. Im Rahmen des 3. Kapitels führt der Autor die wichtigen Konzepte der „Komponente zur Laufzeit“ in Form von Kapseln über die Identifizierung von Wurzelobjekten sowie die Anwendungskonfiguration als Komposition von parametrierten Kapseln, Konnektoren und einer Rechnerzuordnung als wichtigen Beitrag der Arbeit ein.

Die Realisierung adaptiven Verhaltens auf der Basis von Anwendungskonfigurationen hat den großen Vorteil, dass die einzelnen Anwendungskonfigurationen unabhängig voneinander für bestimmte Einsatzszenarien entwickelt und die nichtfunktionalen Eigenschaften der jeweiligen Anwendungskonfigurationen evaluiert werden können. Während der Anwendungslaufzeit kann durch die Beobachtung der Umgebungsparameter und der verfügbaren Ressourcen auf Veränderungen durch den Austausch der aktiven Anwendungskonfigurationen im Rahmen einer dynamischen Rekonfiguration reagiert werden. Existierende Rekonfigurationsalgorithmen unterstützen dabei oft keine An-

wendungen mit mehreren Threads bzw. verursachen hohe Zusatzkosten während der normalen Laufzeitphase. Im Rahmen dieser Arbeit wurde der neue Rekonfigurationsalgorithmus ReDAC entwickelt, der durch den Einsatz Thread-spezifischer Zähler in jeder Kapsel zum einen Anwendungen mit mehreren Threads unterstützt und zum anderen durch die Nutzung logischer Thread-IDs, die Threads systemweit eindeutig identifizieren, für verteilte Anwendungen funktioniert und dabei existierende Verfahren bzgl. der Komplexität der Zusatzkosten in Abhängigkeit der Aufruftiefe für die notwendige Synchronisation während der normalen Ausführung von O(n) nach O(1) verbessert. Durch die abstrakte Präsentation der entwickelten Techniken ist ihr Einsatz in vielen aktuellen Komponentenplattformen möglich. Wichtige Eigenschaften wie die Verklemmungsfreiheit, der gegenseitige Ausschluss und der Fortschritt des entworfenen Algorithmus evaluiert der Autor durch den Einsatz formaler Methoden und durch ausführliche Tests nach.

Im zweiten Teil des 3. Kapitels wurde die im Rahmen des Adapt.Net-Projekts entwickelte Ausführungsplattform für adaptive Anwendungen auf Basis der *Common Language Infrastructure* (CLI) vorgestellt, welche die zuvor abstrakt eingeführten Konzepte durch eine leistungsfähige Implementierung untermauert. Neben der Implementierung der zuvor formal diskutierten Rekonfigurationstechniken zeigt die vorliegende Arbeit, wie mit Hilfe der aspektorientierten Programmierung (re-)konfigurationsspezifische Belange separiert werden können. Die hierdurch ermöglichte effiziente Entwicklung adaptiver Anwendungen durch wiederverwendbare Aspekte stellt einen zweiten wichtigen Beitrag der Arbeit dar, der einen komplexen Einsatzfall für die aspektorientierte Programmierung realisiert. Durch die Nutzung von Basistechnologien bei der Implementierung der Ausführungsplattformen sind die entwickelten Techniken leicht auf andere Komponentenplattformen wie die Java-Plattform übertragbar, was die Arbeit durch die Möglichkeit der Integration heterogener Kapseln in Form von Java-Objekten zeigt. Bei der Implementierung der Ausführungsplattform für adaptive Anwendungen wurde eine XML-basierte domänenspezifische Sprache für die Beschreibung von Konfigurationen entwickelt, auf deren Basis Anwendungen gestartet und Rekonfigurationsoperationen durch den Vergleich zweier Konfigurationsbeschreibungen ermittelt werden können. Eine Besonderheit der generierten Implementierung der Konfigurationsschnittstellen der Kapseln ist dabei, dass die Verbindungen zwischen den Kapseln einer Anwendung und deren Parametrierung automatisch durch einen Konfigurationsmanager auf Basis der XML-Dokumente durchgeführt werden können. Für den Anwendungsentwickler entfallen dadurch die Konfiguration der Kommunikations-Middleware und die Verwaltung von Verbindungen. Für das notwendige Markieren von Attributen, welche die Kommunikationsendpunkte und Parameter repräsentieren, wurde ein neues Programmiermodell auf Basis von Metadatenannotationen vorgestellt. Eine weitere Besonderheit der implementierten Ausführungsplattform ist die Möglichkeit der direkten Kommunikation von lokalen Kapseln über einfache virtuelle Methodenaufrufe, während existierende Ansätze (z.B. [Wegdam 2003]) für die Kommunikation zwischen austauschbaren Einheiten immer eine Kommunikations-Middleware verwenden, die hohe Zusatzkosten während der Ausführung für das Verpacken von Methodenparametern benötigt. Auch die Verteilung und Installation der beteiligten Softwarekomponenten auf entfernten Rechnern konnte durch die entwickelte Ausführungsplattform unterstützt werden. Der Anwendungsentwickler kann die Zuordnung von Kapseln auf Rechner in visueller Form mit einem entwickelten Werkzeug beschreiben. Das Deployment der Komponenten erfolgt automatisch durch die Adapt.Net-Infrastruktur.

Die dynamische Aktualisierung von Kapseln (beschrieben in Kapitel 4) stellt eine komplexe Rekonfigurationsoption dar, die u.a. den Einsatz alternativer Algorithmen und die Anpassung graphischer Nutzerschnittstellen ermöglicht. Besonders hervorzuheben ist die Entwicklung eines Graphtraversierungsalgorithmus für den Objektgraphen einer Kapsel basierend auf der Nutzung der Reflexionsschnittstellen, mit der die dynamische Aktualisierung von Anwendungen ohne die Manipulation der unterliegenden Laufzeitumgebung implementiert werden konnte. Zusätzlich ermöglicht das entwickelte Verfahren die dynamische Aktivierung und die Rekonfiguration von Aspekten, was sich durch die Nutzung der Aspektkonfiguration mit Metadatenannotationen im verwendeten Aspektweber Rapier-Loom.Net sehr gut für die Entwicklung alternativer Anwendungskonfigurationen eignet.

Besonders zu betonen ist, dass die entwickelte *Dynamic Software Update Platform* auch entkoppelt für das Einspielen fehlerbereinigter Versionen in verteilte Anwendungen eingesetzt werden kann. Durch den Einsatz dynamischer Proxys, können Objekte transparent für alle Nutzer dieser Objekte aktualisiert werden, da in einer Anwendung nur Referenzen auf die Proxys existieren. Die explizite Kenntnis interner Verbindungsstrukturen wie in Adapt.Net ist dann nicht notwendig. Mit einer Fallstudie wurde im 133.000 Quellcodezeilen umfassenden Paint.Net gezeigt, dass das entwickelte Verfahren leicht in existierende Software integrierbar ist, ohne dabei komplexe Änderungen am Quellcode vornehmen zu müssen. Die an Hand von Fallstudien bestimmten Leistungsmaße sind dabei vielversprechend.

Im Rahmen des 6. Kapitels wird zunächst die Realisierung adaptiven Verhaltens durch die Vorstellung einer Monitoring-Infrastruktur und der profilbasierten Zuordnung von Anwendungskonfigurationen beschrieben. Des Weiteren wurde ein Werkzeug für die Entwicklung adaptiver Anwendungen mit Hilfe der vorgestellten Techniken beschrieben, wobei vor allem die Generierung adaptionsspezifischer Logik und die Erstellung von auslieferbaren Paketen der Anwendungen im Vordergrund stehen. Die Vorstellung von Mustern und Strategien für die Entwicklung alternativer Anwendungskonfiguration schließen dieses Kapitel ab und demonstrieren Einsatzmöglichkeiten für die in Kapitel 3 und 4 beschriebenen Techniken.

Ein weiterer wichtiger Beitrag der Arbeit ist der Einsatz der entwickelten Techniken für die sichere Ausführung von Steuerungsanwendungen in einem verteilten Web-Labor - dem *Distributed Control Lab* (DCL). Das DCL ermöglicht die Ausführung von Steuerungsexperimenten über das Internet und wurde in zahlreichen Lehrveranstaltungen am Hasso-Plattner-Institut und an der Blekinge Techniska Högskola in Ronneby, Schweden eingesetzt. Um die Beschädigung der Experiment-Hardware durch fehlerhafte und bösartige Steuerungen zu verhindern, wurde die Steuerung als adaptive komponentenbasierte Anwendung realisiert. Durch die Beobachtung dynamischer Umgebungsparameter und der dynamischen Rekonfiguration im Fehlerfall konnten die entwickelten Techniken erfolgreich für den Schutz der Experiment-Hardware eingesetzt werden. Neben der Vorstellung eines Fehlermodells und der Diskussion der Behandlung des Kapselzustands bei der dynamischen Rekonfiguration wurde im Rahmen des Real-Time.Net-Projekts eine Laufzeitumgebung für die vorhersagbare Ausführung von .NET-Programmen vorgestellt.

Trotz der umfangreichen Behandlung vieler Faktoren bei der Entwicklung und Ausführung adaptiver komponentenbasierter Anwendungen müssen einige Fragestellungen in zukünftigen Arbeiten untersucht werden. Hierzu gehört die Ressourcenverwaltung bei der Ausführung multipler Anwendungen auf einem System, die z.B. durch einen zentra-

len Ressourcenmanager realisiert werden kann, der zusätzlich die Konfiguration von Nutzerpräferenzen erlaubt. Genauer untersucht werden muss auch die Implementierung der vorgestellten Techniken auf mobilen Geräten mit eingeschränkten Ressourcen, die im Rahmen dieser Arbeit konzeptionell durch die Identifikation von Basistechnologien beschrieben wurde.
Bei der dynamischen Aktualisierung wurden mit den *UpdateHandlern* und den Aktualisierungskonstruktoren mächtige Werkzeuge für den selektiven Zustandstransfer vorgestellt. An dieser Stelle sind aber durchaus komfortablere Lösungen, die ohne die Nutzung der Reflexionsfunktionalität auskommen, vorstellbar, da sie direkt vom Anwendungsentwickler verwendet werden müssen. Vor allem bei starken Differenzen der Objektgraphen alternativer Algorithmen können neue Techniken untersucht werden. Erweiterungen der Laufzeitumgebungen mit Möglichkeiten der Manipulation des Ausführungs-Stack würden die dynamische Aktualisierung auch bei aktiven Methoden ermöglichen, da auf diese Weise auch während laufender Methodenaufrufe Referenzen auf Wurzelobjekte auf dem Stack aktualisiert werden könnten. Diese Problematik muss vor allem für langlaufende Methodenaufrufe untersucht werden, da die Unterbrechungszeit des vorgestellten Rekonfigurationsalgorithmus von der maximalen Dauer der Methodenaufrufe bestimmt wird. Im Rahmen der Arbeit hat sich allerdings gezeigt, dass durch eine geschickte Wahl der Wurzelobjekte die durch die Laufzeit der Methodenaufrufe verursachten Unterbrechungszeiten bei vielen Anwendungen unproblematisch sind.
Auch die Komposition von Anwendungen aus Komponenten bedarf der weiteren Erforschung, damit die vollständige automatische Erstellung von Anwendungskonfigurationen ermöglicht wird. Vor allem die vielen Freiheitsgrade bei der Berechnung der Eigenschaften der Komponenten und der resultierenden Anwendungseigenschaften auf verschiedenen Rechnern sind eine komplexe Problematik. Der im Rahmen dieser Arbeit vorgestellte Ansatz kann hier leicht durch zusätzliche Planungsmodule ergänzt werden, die über den Konfigurationsmanager neue während der Laufzeit berechnete Konfigurationen nachladen können, auch um auf unvorhergesehene Änderungen der Umgebung reagieren zu können. Die in der Arbeit eingesetzte Verknüpfung von Umgebungseigenschaften und Anwendungskonfigurationen mit Hilfe von Profilen wurde hauptsächlich zur Demonstration der entwickelten Rekonfigurationsstrategien verwendet. Bei einer Vielzahl von Umgebungsparametern sind Profile problematisch, da potentiell viele Kombinationen von Parameterbereichen festgehalten werden müssen. Außerdem können zwischen den einzelnen Bereichen Konflikte entstehen, die vom Anwendungsentwickler manuell überblickt werden müssen.
In der Zukunft sind im breiten Gebiet der adaptiven Software weiterführende Fragestellungen zu erforschen. Vor allem die Adaption komplexer Anwendungen im Rahmen einer Selbst-Organisation mit dezentraler Steuerung muss untersucht werden. Der Einsatz der in dieser Arbeit vorgestellten Rekonfigurationstechniken ist dabei denkbar. Im Bereich der mobilen Systeme muss die Entwicklung komponentenbasierter Ausführungsplattformen und entsprechender Middleware, die den eingeschränkten Ressourcen Rechnung trägt, betrachtet werden. Dynamische Aktualisierungen stellen komplexe Änderungen zur Laufzeit dar, die, um den Ausfall einer Software zu vermeiden, zuvor getestet bzw. verifiziert werden müssen. Durch die kürzlich verfügbaren Bytecode-basierten Aspektweber ist die weitere Verbesserung der Werkzeuge zur Entwicklung adaptiver Anwendungen vorstellbar.

Literaturverzeichnis

[Abowd u. a. 1999] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a Better Understanding of Context and Context-Awareness. In: Gellersen, H.-W. (Hrsg.): *1st International Symposium on Handheld and Ubiquitous Computing (HUC'99)*. Karlsruhe, Germany : Springer-Verlag, September 1999, S. 304–307. – LNCS 1707. – ISBN 3-540-66550-1

[Alia u. a. 2006] Alia, Mourad ; Eliassen, Frank ; Hallsteinsen, Svein ; Stav, Erlend: MADAM: towards a flexible planning-based middleware. In: *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*. New York, NY, USA : ACM, 2006, S. 96–96. – ISBN 1-59593-403-0

[Almeida u. a. 2001] Almeida, Joao Paulo A. ; Sinderen, Marten V. ; Nieuwenhuis, Lambert: Transparent Dynamic Reconfiguration for CORBA. In: *DOA '01: Proceedings of the Third International Symposium on Distributed Objects and Applications*. Washington, DC, USA : IEEE Computer Society, 2001, S. 197. – ISBN 0-7695-1300-X

[Alpern u. a. 2005] Alpern, B. ; Augart, S. ; Blackburn, S. M. ; Butrico, M. ; Cocchi, A. ; Cheng, P. ; Dolby, J. ; Fink, S. ; Grove, D. ; Hind, M. ; McKinley, K. S. ; Mergen, M. ; Moss, J. E. B. ; Ngo, T. ; Sarkar, V.: The Jikes research virtual machine project: building an open-source research community. In: *IBM Systems Journal* 44 (2005), Nr. 2, S. 399–417. – ISSN 0018-8670

[Avizienis 1995] Avizienis, A.: The Methodology of N-Version Programming. In: Lyu, M. R. (Hrsg.): *Software Fault Tolerance*. John Wiley & Sons, 1995, S. 23–46. – ISBN 978-0471950684

[Barthel 2008] Barthel, Stefan: *Performance Prediction of Centralized Protection and Control Applications*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, 2008. – in Zusammenarbeit mit ABB Corporate Research Switzerland

[Ben-Ari 1982] Ben-Ari, Mordechai: *Principles of Concurrent Programming*. Prentice Hall Professional Technical Reference, 1982. – ISBN 0137010788

[Beuche u. a. 1999] Beuche, Danilo ; Guerrouat, Abdelaziz ; Papajewski, Holger ; Schröder-Preikschat, Wolfgang ; Spinczyk, Olaf ; Spinczyk, Ute: The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In: *ISORC '99: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 1999, S. 45–53. – ISBN 0-7695-0207-5

[Bidan u. a. 1998] Bidan, Christophe ; Issarny, Valerie ; Saridakis, Titos ; Zarras, Apostolos: A Dynamic Reconfiguration Service for CORBA. In: *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*. Washington, DC, USA : IEEE Computer Society, 1998, S. 35. – ISBN 0-8186-8451-8

[Blair u. a. 2002] Blair, Gordon S. ; Coulson, Geoff ; Blair, Lynne ; Duran-Limon, Hector A. ; Grace, Paul ; Moreira, Rui S. ; Parlavantzas, Nikos: Reflection, self-awareness and self-healing in OpenORB. In: Garlan, David (Hrsg.) ; Kramer, Jeff (Hrsg.) ; Wolf, Alexander L. (Hrsg.): *WOSS '02: Proceedings of the first workshop on Self-healing systems*, ACM, 2002, S. 9–14. – ISBN 1-58113-609-9

[Bloom und Day 1993] Bloom, Toby ; Day, Mark: Reconfiguration and module replacement in Argus: theory and practice. In: *Software Engineering Journal* (1993), März, S. 102–108

[Bollella u. a. 2000] Bollella, Greg ; Brosgol, Ben ; Dibble, Peter ; Furr, Steve ; Gosling, James ; Hardin, David ; Turnbull, Mark ; Belliardi, Rudy: *The Real-Time Specification for Java*. http://www.rtj.org/. 2000

[Bollow 2008] Bollow, Norbert: *DotGNU Project*. www.gnu.org/software/dotgnu. 2008

[Borchert 2005] Borchert, Paul: *Verlässliche Programmausführung in der Asparagusumgebung*, Universität Potsdam, Diplomarbeit, Oktober 2005

[Borland 2007] Borland: *Plattform Interoperabilität für Organisationen*. 2007. – URL `http://www.borland.com/de/products/janeva/`

[Braden 1997] Braden, R.: *Resource ReSerVation Protocol (RSVP) Version 1 Functional Specification*. 1997. – URL `citeseer.ist.psu.edu/braden96resource.html`

[Brewer u. a. 1998] Brewer, E. ; Katz, R. H. ; Chawathe, Y. ; Gribble, S. D. ; Hodes, T. ; Nguyen, G. ; Stemm, M. ; Henderson, T. ; Amir, E. ; Balakrishnan, H. ; Fox, A. ; Padmanabhan, V. N. ; Seshan, S.: A Network Architecture for Heterogeneous Mobile Computing. In: *IEEE Personal Communications Magazine* 5 (1998), Oktober, Nr. 5, S. 8–24

[Buschmann u. a. 1996] Buschmann, F. ; Meunier, R. ; Rohnert, H. ; Sommerlad, P. ; Stal, M.: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996. – ISBN 0471958697

[Chen und Kotz 2000] Chen, Guanling ; Kotz, David: A Survey of Context-Aware Mobile Computing Research / Dept. of Computer Science, Dartmouth College. URL `http://www.cs.dartmouth.edu/reports/TR2000-381/`, November 2000 (TR2000-381). – Forschungsbericht

[Corbato und Vyssotsky 1965] Corbato, F. ; Vyssotsky, V.: Introduction and overview of the MULTICS system. In: *Proceedings of AFIPS 1965 Fall Joint Computer Conference* Bd. 27, June 1965, S. 185–197

[Cross und Schmidt 2003] Cross, Joseph K. ; Schmidt, Douglas C.: Applying the quality connector pattern to optimise distributed real-time and embedded applications. (2003), S. 209–235. ISBN 1-85233-506-8

[Cygnus 1999] Cygnus: *eCos: Embedded Cygnus Operating System - White Paper*. 1999. – URL `http://www.cygnus.com/ecos`

[Darwin 1859] Darwin, Charles: *On The Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. London : John Murray, 1859

[DeRemer und Kron 1976] DeRemer, Frank ; Kron, Hans H.: Programming-in-the-Large versus Programming-in-the-Small. In: Schneider, Hans J. (Hrsg.) ; Nagl, Manfred (Hrsg.): *Fachtagung ueber Programmiersprachen* Bd. 1, Springer, 1976, S. 80–89. – ISBN 3-540-07619-0

[Dini u. a. 2004] Dini, Petre ; Gentzsch, Wolfgang ; Potts, Mark ; Clemm, Alexander ; Yousif, Mazin ; Polze, Andreas: Internet, GRID, Self-Adaptability and Beyond: Are We Ready? In: *DEXA '04: Proceedings of the Database and Expert Systems Applications, 15th International Workshop on (DEXA'04)*. Washington, DC, USA : IEEE Computer Society, 2004, S. 782–788. – ISBN 0-7695-2195-9

[Dmitriev 2001] Dmitriev, Mikhail: Safe Class and Data Evolution in Large and Long-Lived Java[tm] Applications. Mountain View, CA, USA : Sun Microsystems, Inc., 2001. – Forschungsbericht

[Ecma International 2006a] Ecma International: *Standard ECMA-334 C♯ Language Specification 4th edition*. http://www.ecma-international.org/publications/standards/Ecma-334.htm. Juni 2006

[Ecma International 2006b] Ecma International: *Standard ECMA-335, Common Language Infrastructure (CLI) 4th edition*. http://www.ecma-international.org/publications/standards/Ecma-335.htm. Juni 2006

[Ensink u. a. 2003] Ensink, Brian ; Stanley, Joel ; Adve, Vikram: Program control language: a programming language for adaptive distributed applications. In: *J. Parallel Distrib. Comput.* 63 (2003), Nr. 11, S. 1082–1104. – ISSN 0743-7315

[Fabry 1976] Fabry, R. S.: How to design a system in which modules can be changed on the fly. In: *ICSE '76: Proceedings of the 2nd international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1976, S. 470–476

[Fay-Wolfe u. a. 2000] Fay-Wolfe, Victor ; DiPippo, Lisa C. ; Copper, Gregory ; Johnston, Russell ; Kortmann, Peter ; Thuraisingham, Bhavani: Real-Time CORBA. In: *IEEE Trans. Parallel Distrib. Syst.* 11 (2000), Nr. 10, S. 1073–1089. – ISSN 1045-9219

[Forax u. a. 2005] Forax, Rémi ; Duris, Etienne ; Roussel, Gilles: Reflection-based implementation of Java extensions: the double-dispatch use-case. In: *Journal of Object Technology, vol. 4, no. 10, Special Issue: OOPS Track at SAC 2005 Santa-Fe* (2005), december, S. 49–69. – URL `http://www.jot.fm/issues/issues 2005 12/article3`

[Gamma u. a. 1995] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster*. Addison-Wesley, 1995. – ISBN 3-8273-1862-9

[Garlan u. a. 1997] Garlan, David ; Monroe, Robert T. ; Wile, David: Acme: an architecture description interchange language. In: Johnson, J. H. (Hrsg.): *CASCON*, IBM, 1997, S. 7

[Gebhart 2006] Gebhart, Dr. A.: *SAP Dynamic Capacity Management*. 2006. – URL `http://www.dcl.hpi.uni-potsdam.de/teaching/IndusSem/slides/gebhart.pdf`

[Geier u. a. 1998] Geier, Martin ; Steckermeier, Martin ; Becker, Ulrich ; Hauck, Franz J. ; Meier, Erich ; Rastofer, Uwe: Support for Mobility and Replication in the AspectIX Architecture. In: *ECOOP '98: Workshop ion on Object-Oriented Technology*. London, UK : Springer-Verlag, 1998, S. 325–326. – ISBN 3-540-65460-7

[Geihs u. a. 2006] Geihs, Kurt ; Khan, Mohammad U. ; Reichle, Roland ; Solberg, Arnor ; Hallsteinsen, Svein: Modeling of Component-Based Self-Adapting Context-Aware Applications for Mobile Devices. In: *Proceedings of IFIP Working Conference on Software Engineering Techniques*, Oktober 2006

[Gelernter 1985] Gelernter, David: Generative communication in Linda. In: *ACM Trans. Program. Lang. Syst.* 7 (1985), Nr. 1, S. 80–112. – ISSN 0164-0925

[German u. a. 1995] German, Reinhard ; Kelling, Christian ; Zimmermann, Armin ; Hommel, Günter: TimeNET: a toolkit for evaluating non-Markovian stochastic Petri nets. In: *Performance Evaluation* 24 (1995), Nr. 1-2, S. 69–87. – ISSN 0166-5316

[Goldberg und Robson 1983] Goldberg, Adele ; Robson, David: *Smalltalk-80: the language and its implementation*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1983. – ISBN 0-201-11371-6

[Goodenough und Sha 1988] Goodenough, J. B. ; Sha, L.: The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. In: *Ada Lett.* VIII (1988), Nr. 7, S. 20–31. – ISSN 1094-3641

[Gorlick und Razouk 1991] Gorlick, Michael M. ; Razouk, Rami R.: Using weaves for software construction and analysis. In: *ICSE '91: Proceedings of the 13th international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1991, S. 23–34. – ISBN 0-89791-391-4

[Heineman 1998] Heineman, George T.: Adaptation and software architecture. In: *ISAW '98: Proceedings of the third international workshop on Software architecture*. New York, NY, USA : ACM Press, 1998, S. 61–64. – ISBN 1-58113-081-3

[Heinrich 1993] Heinrich, Michael: Ressourcenorientiertes Konfigurieren. In: *Künstliche Intelligenz Band 7* Volume 1 (1993)

[Herlihy und Liskov 1982] Herlihy, Maurice P. ; Liskov, Barbara: A Value Transmission Method for Abstract Data Types. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), Nr. 4, S. 527–551. – ISSN 0164-0925

[Hirschfeld und Kawamura 2006] Hirschfeld, Robert ; Kawamura, Katsuya: Dynamic service adaptation: Experiences with Auto-adaptive and Reconfigurable Systems. In: *Software - Practice and Experience (SPE)* 36 (2006), Nr. 11.12, S. 1115–1131. – ISSN 0038-0644

[Hofmeister und Purtilo 1993] Hofmeister, Christine ; Purtilo, James M.: Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement. In: *Proceedings of the 13th International Conference on Distributed Computing Systems*. Pittsburgh, Pennsylvania, USA : IEEE Computer Society Press, 1993, S. 101–110

[Hofmeister 1994] Hofmeister, Christine R.: *Dynamic Reconfiguration of Distributed Applications*, University of Maryland, Department of Computer Science, Dissertation, January 1994. – Technical Report CS-TR-3210

[IIOP.NET Homepage 2007] IIOP.NET Homepage: *.NET, CORBA and J2EE Interoperation*. http://iiop-net.sourceforge.net. 2007

[Issel 2006] Issel, Helge: *Dynamische Rekonfiguration in Eingebetteten Systemen*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, 2006

[Jacob u. a. 2004] Jacob, Bart ; Lanyon-Hogg, Richard ; Nadgir, Devaprasad K. ; Yassin, Amr F.: A Practical Guide to the IBM Autonomic Computing Toolkit. In: *IBM Redbook* (2004), S. 41–50. ISBN 073849805X

[Jann u. a. 2003] Jann, J. ; Browning, L. M. ; Burugula, R. S.: Dynamic reconfiguration: Basic building blocks for autonomic computing on IBM pSeries servers. In: *IBM Syst. J.* 42 (2003), Nr. 1, S. 29–37. – ISSN 0018-8670

[Kasten u. a. 2002] Kasten, Eric P. ; McKinley, Philip K. ; Sadjadi, S. M. ; Stirewalt, Kurt: Separating Introspection and Intercession to Support Metamorphic Distributed Systems. In: *ICDCSW '02: Proceedings of the 22nd International Conference on Distributed Computing Systems*. Washington, DC, USA : IEEE Computer Society, 2002, S. 465–472. – ISBN 0-7695-1588-6

[Kephart und Chess 2003] Kephart, Jeffrey O. ; Chess, David M.: The Vision of Autonomic Computing. In: *Computer* 36 (2003), Nr. 1, S. 41–50. – ISSN 0018-9162

[KG 2007] KG, SAP Deutschland AG & C.: *Adaptive Computing mit der Plattform SAP NetWeaver*. 2007. – URL `http://www.sap.com/germany/media/50074626.pdf`

[Kiczales u. a. 2001] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ. In: *Lecture Notes in Computer Science* 2072 (2001), S. 327–355

[Kiczales u. a. 1997] Kiczales, Gregor ; Lamping, John ; Menhdhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean-Marc ; Irwin, John: Aspect-Oriented Programming. In: Mehmet Akşit and Satoshi Matsuoka (Hrsg.): *Proceedings European Conference on Object-Oriented Programming* Bd. 1241. Berlin, Heidelberg, and New York : Springer-Verlag, 1997, S. 220–242

[Kon 2000] Kon, Fabio: *Automatic Configuration of Component-Based Distributed Systems*, Department of Computer Science, University of Illinois at Urbana-Champaign, Dissertation, May 2000

[Kon u. a. 2000] Kon, Fabio ; Román, Manuel ; Liu, Ping ; Mao, Jina ; Yamane, Tomonori ; Magalha, Claudio ; Campbell, Roy H.: Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In: *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 2000, S. 121–143. – ISBN 3-540-67352-0

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

[Kramer und Magee 2007] Kramer, Jeff ; Magee, Jeff: Self-Managed Systems: an Architectural Challenge. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007, S. 259–268. – ISBN 0-7695-2829-5

[Kramer und Sloman 1988] Kramer, Jeff ; Sloman, Morris: *Verteilte Systeme und Rechnernetze*. Hanser Fachbuchverlag, 1988. – ISBN 978-3446153462

[Laprie 1998] Laprie, Jean C.: *Dependability. Basic Concepts and Terminology*. Springer Verlag, 1998. – ISBN 978-3211822968

[Leidner 2006] Leidner, Andreas: *Online-Updating von hochverfügbaren Webanwendungen*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, 2006

[Li und Nahrstedt 1999] Li, B. ; Nahrstedt, K.: A Control-Based Middleware Framework for Quality of Service Adaptations. In: *IEEE J. Select. Areas Commun.* (1999). – URL `citeseer.nj.nec.com/li99controlbased.html`

[Lieberherr 1996] Lieberherr, Karl J.: *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. – ISBN 0-534-94602-X

[Limprecht 1997] Limprecht, R.: Microsoft Transaction Server. In: *COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference*. Washington, DC, USA : IEEE Computer Society, 1997, S. 14. – ISBN 0-8186-7804-6

[Litiu und Prakash 2000] Litiu, Radu ; Prakash, Atul: DACIA: A Mobile Component Framework for Building Adaptive Distributed Applications. In: *Principles of Distributed Computing (PODC) 2000 Middleware Symposium* (2000). – URL `citeseer.nj.nec.com/410888.html`

[Liu und Layland 1973] Liu, C.L. ; Layland, J.W: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In: *Journal of the ACM* 20 (1973), January, Nr. 1, S. 46–61

[Liu 2000] Liu, Jane W. S. W.: *Real-Time Systems*. Upper Saddle River, NJ, USA : Prentice Hall PTR, 2000. – ISBN 0130996513

[dotPDN LLC 2008] LLC dotPDN: *Paint.NET Source Code*. http://www.dotpdn.com/downloads/pdnsrc.html. 2008

[von Löwis und Möller 2006] Löwis, Martin von ; Möller, Jan: A Microsoft .NET Front-End for GCC. In: *NET Technologies'2006*, 2006, S. 17–20. – ISBN 80-86943-11-9

[von Löwis und Rasche 2006] Löwis, Martin von ; Rasche, Andreas: Towards a Real-Time Implementation of the ECMA Common Language Infrastructure. In: *Proceedings of Ninth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2006), 24-26 April 2006, Gyeongju, Korea*, IEEE Computer Society, 2006, S. 125–132. – ISBN 0-7695-2561-X

[Luckham u. a. 1995] Luckham, David C. ; Kenney, John L. ; Augustin, Larry M. ; Vera, James ; Bryan, Doug ; Mann, Walter: Specification and Analysis of System Architecture Using Rapide. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 4, S. 336–355

[LumiSoft 2008] LumiSoft: *LumiSoft Mail Server*. http://www.lumisoft.ee/lswww/ENG/Products/Mail_Server/mail_index_eng.aspx?type=info. 2008

[Maes 1987] Maes, Pattie: Concepts and experiments in computational reflection. In: *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*. New York, NY, USA : ACM Press, 1987, S. 147–155. – ISBN 0-89791-247-0

[Magee u. a. 1995] Magee, Jeff ; Dulay, Naranker ; Eisenbach, Susan ; Kramer, Jeff: Specifying Distributed Software Architectures. In: *Proceedings of the 5th European Software Engineering Conference*. London, UK : Springer-Verlag, 1995, S. 137–153. – ISBN 3-540-60406-5

[Magee u. a. 1989] Magee, Jeff ; Kramer, Jeff ; Sloman, Morris: Constructing Distributed Systems in Conic. In: *IEEE Transactions on Software Engineering* 15 (1989), Nr. 6

[Mahmoud 2004] Mahmoud, Qusay H. (Hrsg.): *Middleware for Communications*. John Wiley & Sons, Juli 2004. – ISBN 0-470-86206-8

[McKinley u. a. 2005] McKinley, Philip. K. ; Stirewalt, R. E. K. ; Cheng, Betty H. C. ; Dillon, Laura K. ; Kulkarni, Sandeep: RAPIDware: Component-Based Development of Adaptive and Dependable Middleware. East Lansing, Michigan 48824 : Software Engineering and Network Systems Laboratory Department of Computer Science and Engineering Michigan State University, 2005. – Forschungsbericht. – URL `www.cse.msu.edu/rapidware`

[Medvidovic u. a. 1997] Medvidovic, Nenad ; Oreizy, Peyman ; Taylor, Richard N.: Reuse of off-the-shelf components in C2-style architectures. In: *SSR '97: Proceedings of the 1997 symposium on Software reusability*. New York, NY, USA : ACM Press, 1997, S. 190–198. – ISBN 0-89791-945-9

[Microsoft Corporation 2008] Microsoft Corporation: *Shared Source Common Language Infrastructure 2.0 Release*. msdn.microsoft.com/net/sscli. 2008

[Milanovic und Malek 2004] Milanovic, Nikola ; Malek, Miroslaw: Current Solutions for Web Service Composition. In: *IEEE Internet Computing* 8 (2004), Nr. 6, S. 51–59. – ISSN 1089-7801

[Moazami-Goudarzi 1999] Moazami-Goudarzi, K.: *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*, Imperial College London, Dissertation, March 1999. – URL `http://www-dse.doc.ic.ac.uk/~km2/phd/thesis.ps.gz`

[Müller und Widmer 2005] Müller, Stephan ; Widmer, Sven: *Semesterarbeit - Version VFS: Erweiterung des virtuellen Dateisystems unter Linux um eine Versionsverwaltung*. Dezember 2005. – Universität Potsdam

[Nicolai 2005] Nicolai, Johannes: Sichere Ausführung nicht vertrauenswürdiger Programme - Evaluation verschieder Ansätze an vier Beispielen / Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam. 2005 (Technische Berichte Nr. 9). – Forschungsbericht. – S. 132. – ISBN 3-937786-73-2

[Noble 2000] Noble, Brain D.: *System Support for Mobile, Adaptive Applications*. February 2000. – URL `citeseer.csail.mit.edu/noble00system.html`

[Noble u. a. 1997] Noble, Brian D. ; Satyanarayanan, M. ; Narayanan, Dushyanth ; Tilton, James E. ; Flinn, Jason ; Walker, Kevin R.: Agile Application-Aware Adaptation for Mobility. In: *Sixteen ACM Symposium on Operating Systems Principles*. Saint Malo, France : ACM Press, 1997, S. 276–287. – URL `citeseer.nj.nec.com/noble97agile.html`

[Novell 2008] Novell: *The Mono Project*. www.mono-project.com. 2008

[Object Management Group 2004] Object Management Group: *Common object request broker architecture: core specification, version 3.0.3 OMG specification formal/04-03-12*. march 2004

[Object Management Group 2006] Object Management Group, Inc.: *Deployment and Configuration of Component-based Distributed Applications Specification Version 4.0*. April 2006. – OMG Available Specification formal/06-04-02

[Oreizy 1996] Oreizy, Peyman: Issues in the Runtime Modification of Software Architectures / University of California, Irvine. Irvine, CA, USA, August 1996 (UCI-ICS-96-35). – Forschungsbericht

[Oreizy u. a. 1999] Oreizy, Peyman ; Gorlick, Michael M. ; Taylor, Richard N. ; Heimbigner, Dennis ; Johnson, Gregory ; Medvidovic, Nenad ; Quilici, Alex ; Rosenblum, David S. ; Wolf, Alexander L.: An Architecture-Based Approach to Self-Adaptive Software. In: *IEEE Intelligent Systems* 14 (1999), S. 54–62. – URL `http://www.ics.uci.edu/ peymano/papers/`

[Pal u. a. 2000] Pal, Partha ; Loyall, Joseph ; Schantz, Richard ; Zinky, John ; Shapiro, Rich ; Megquier, James: Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. In: *ISORC '00: Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 2000, S. 310. – ISBN 0-7695-0607-0

[Palma u. a. June 2001] Palma, N. D. ; Laumay, P. ; Bellissard, L.: Ensuring Dynamic Reconfiguration Consistency. In: *6th International Workshop on Component-Oriented Programming (WCOP 2001), ECOOP related Workshop*. Budapest,Hungary, June 2001, S. 18–24

[Parnas 2001] Parnas, David L.: On the design and development of program families. In: *Software fundamentals: collected papers by David L. Parnas* (2001), S. 193–213. ISBN 0-201-70369-6

[Polze und Rasche 2005] Polze, Andreas ; Rasche, Andreas: In: Liggesmeyer, Peter (Hrsg.) ; Rombach, Dieter (Hrsg.): *Software Engineering eingebetteter Systeme: Grundlagen - Methodik - Anwendungen*. Spektrum akademischer Verlag Elsevier, 2005, Kap. Programmierung eingebetteter Software, S. 179–203. – ISBN 3-8274-1533-0

[Popovici u. a. 2002] Popovici, Andrei ; Gross, Thomas ; Alonso, Gustavo: Dynamic weaving for aspect-oriented programming. In: *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. New York, NY, USA : ACM, 2002, S. 141–147. – ISBN 1-58113-469-X

[Puhlmann 2005] Puhlmann, Marco: *Semesterarbeit (nicht eingereicht) - Dynamische Rekonfiguration und Adaption heterogener verteilter Anwendungen*. 2005. – Universität Potsdam

[Rajkumar u. a. 1997] Rajkumar, R. ; Lee, C. ; Lehoczky, J. ; Siewiorek, D.: A Resource Allocation Model for QoS Management. In: *IEEE Real-Time Systems Symposium*, Dezember 1997, S. 298–307

[Rasche 2006] Rasche, Andreas: Werkzeugunterstützung für die Entwicklung selbstadaptiver Anwendungen. In: *Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS) - Fachgespräch der GI/ITG-Fachgruppe Kommunikation und Verteilte Systeme Universität Kassel* (2006)

[Rasche und Polze 2002] Rasche, Andreas ; Polze, Andreas: Configurable Services for mobile Users. In: *Proceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*. San Diego, CA, January 2002, S. 163–171

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

[Rasche und Polze 2005] Rasche, Andreas ; Polze, Andreas: Dynamic Reconfiguration of Component-based Real-time Software. In: *Proceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*, IEEE Computer Society, 2005, S. 347–354. – ISBN 0-7695-2347-1

[Rasche und Polze 2008] Rasche, Andreas ; Polze, Andreas: ReDAC - Dynamic Reconfiguration of distributed component-based applications with cyclic dependencies. In: *to appear in ISORC '08: Proceedings of the 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. Orlando, Florida, USA : IEEE Computer Society, May 2008

[Rasche u. a. 2005a] Rasche, Andreas ; Puhlmann, Marco ; Polze, Andreas: Heterogeneous Adaptive Component-Based Applications with Adaptive.Net. In: *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 418–425. – ISBN 0-7695-2356-0

[Rasche u. a. 2005b] Rasche, Andreas ; Rabe, Bernhard ; Löwis, Martin von ; Möller, Jan ; Polze, Andreas: Real-Time Robotics and Process Control Experiments in the Distributed Control Lab. In: *IEE Software, Special Edition on Microsoft Research Embedded Systems RFP*, IEE Proceedings Software, Oktober 2005, S. 229– 235. – ISSN 1462-5970

[Rasche u. a. 2004] Rasche, Andreas ; Rabe, Bernhard ; Tröger, Peter ; Polze, Andreas: Distributed Control Lab. In: *Proceedings of The 1st International Workshop on e-learning and Virtual and Remote Laboratories (VIRTUAL-LAB'2004)*. Setúbal, Portugal : INSTICC Press, August 2004, S. 150–160. – ISBN 972-8865-14-7

[Rasche und Schult 2007] Rasche, Andreas ; Schult, Wolfgang: Dynamic Updates of Graphical Components in the .NET Framework. In: *Proceedings of Workshop on Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS) im Rahmen der GI/ITG-Tagung Kommunikation in Verteilten Systemen,*, VDE Verlag, 2007, S. 219 – 230. – ISBN 978-3-8007-2980-7

[Rasche u. a. 2005c] Rasche, Andreas ; Schult, Wolfgang ; Polze, Andreas: Self-adaptive multithreaded applications: a case for dynamic aspect weaving. In: *ARM '05: Proceedings of the 4th workshop on Reflective and adaptive middleware systems*. New York, NY, USA : ACM Press, 2005. – ISBN 1-59593-270-4

[Rasche u. a. 2003] Rasche, Andreas ; Tröger, Peter ; Dirska, Michael ; Polze, Andreas: Foucault's Pendulum in the Distributed Control Lab. In: *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*. Capri Island, Italy, October 2003, S. 299–306

[Raymond 1995] Raymond, Kerry: *Reference model of Open Distributed Processing (RM-ODP):Introduction RM-ODP Tutorial*. Center for Information Technology,University of Queesland, Australia, 1995. – URL `citeseer.ist.psu.edu/raymond95reference.html`

[Redmond und Cahill 2002] Redmond, Barry ; Cahill, Vinny: Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In: *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*. London, UK : Springer-Verlag, 2002, S. 205–230. – ISBN 3-540-43759-2

[RFC793 1981] RFC793: Transmission Control Protocol. (1981), September. – URL `http://www.ietf.org/rfc/rfc793.txt`. – DARPA Internet Program Protocol Specification

[Richling 2006] Richling, Jan: *Komponierbarkeit eingebetteter Echtzeitsysteme*. Cuvillier, E, 2006. – ISBN 978-3865377784

[Richter 2006] Richter, Stefan: *Software Development for Embedded Systems Based on ECMA 335*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, September 2006

[Richter u. a. 2007] Richter, Stefan ; Rasche, Andreas ; Polze, Andreas: Hardware-Near Programming in the Common Language Infrastructure. In: *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 2007, S. 329–336. – ISBN 0-7695-2765-5

[Rutherford u. a. 2002] Rutherford, M.J. ; Anderson, K. ; Carzaniga, A. ; Heimbigner, D. M. ; Wolf, A. L.: Reconfiguration in the Enterprise Java Beans Component Model. In: J. Bishop (Hrsg.): *Proc. of the* 1^{st} *IFIP/ACM Working Conference on Component Deployment, Berlin, Germany*. Springer, 2002 (LNCS). – To appear

[Ryman 2001] Ryman, Arthur: Simple object access protocol (SOAP) and Web services. In: *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2001, S. 689. – ISBN 0-7695-1050-7

[Sadjadi und McKinley 2003] Sadjadi, S. M. ; McKinley, P. K.: A Survey of Adaptive Middleware / Department of Computer Science, Michigan State University. East Lansing, Michigan, December 2003 (MSU-CSE-03-35). – Forschungsbericht. – S. 38

[Sadjadi u. a. 2006] Sadjadi, S. M. ; McKinley, P. K. ; Kasten, E. P. ; Zhou, Z.: MetaSockets: design and operation of runtime reconfigurable communication services: Experiences with Auto-adaptive and Reconfigurable Systems. In: *Software - Practice and Experience (SPE)* 36 (2006), Nr. 11-12, S. 1157–1178. – ISSN 0038-0644

[Sadjadi u. a. 2004] Sadjadi, S. M. ; McKinley, Philip K. ; Cheng, Betty H. ; Stirewalt, R.E. K.: 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

[Sastry und Bodson 1989] Sastry, Shankar ; Bodson, Marc: *Adaptive control: stability, convergence, and robustness*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1989. – ISBN 0-13-004326-5

[Satyanarayanan u. a. 1990] Satyanarayanan, M. ; Kistler, James J. ; Kumar, Puneet ; Okasaki, Maria E. ; Siegel, Ellen H. ; Steere, David C.: Coda: A Highly Available File System for a Distributed Workstation Environment. In: *IEEE Transactions on Computers* 39 (1990), Nr. 4, S. 447–459

[Schilit und Theimer 1994] Schilit, Bill ; Theimer, M.: Disseminating Active Map Information to Mobile Hosts. In: *IEEE Network* 8 (1994), Nr. 5, S. 22–32

[Schmeck 2004] Schmeck, Hartmut: Organic Computing-Vision and Challenge for System Design. In: *Proceedings of the Parallel Computing in Electrical Engineering,*

International Conference on (PARELEC 2004). Los Alamitos, CA, USA : IEEE Computer Society, 2004, S. 3–3. – ISBN 978-0-7695-2080-3

[Schmeck 2005] Schmeck, Hartmut: Organic Computing - A New Vision for Distributed Embedded Systems. In: *ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*. Washington, DC, USA : IEEE Computer Society, 2005, S. 201–203. – ISBN 0-7695-2356-0

[Schmidt u. a. 1997] Schmidt, D. ; Gokhale, A. ; Harrison, T. ; Levine, D. ; Cleeland, C.: *TAO: A High-performance Endsystem Architecture for Real-time CORBA*. 1997

[Schmidt u. a. 2000] Schmidt, D. ; Stal, M. ; Rohnert, H. ; Buschmann, F.: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, September 2000. – ISBN 0471606952

[Schmidt 1998] Schmidt, Douglas C.: *An Architectural Overview of the ACE Framework*. 1998

[Schneider u. a. 2004] Schneider, Etienne ; Picioroagă, Florentin ; Brinkschulte, Uwe: Dynamic reconfiguration through OSA+, a real-time middleware. In: *DSM '04: Proceedings of the 1st international doctoral symposium on Middleware*. New York, NY, USA : ACM Press, 2004, S. 319–323. – ISBN 1-58113-948-9

[Schöbel 2005] Schöbel, Michael: *Effiziente Implementierung des TupelSpace-Ansatzes für .NET*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, Dezember 2005

[Schult und Polze 2002] Schult, Wolfgang ; Polze, Andreas: Dynamic Aspect-Weaving with .NET. In: *Workshop zur Beherrschung nicht-funktionaler Eigenschaften in Betriebssystemen und Verteilten Systemen*. TU Berlin, Germany, November 7-8 2002

[Schulzrinne u. a. 2003] Schulzrinne, H. ; Casner, S. ; Frederick, R. ; Jacobson, V.: RFC 3550: RTP: A Transport Protocol for Real-Time Applications / IETF. URL `www.ietf.org/rfc/rfc3550.txt`, 2003. – Forschungsbericht

[Seitz 2003] Seitz, Matthias: *Speicherprogrammierbare Steuerungen*. Fachbuchverlag Leipzig, 2003. – ISBN 3-446-22174-3

[Selic 1998] Selic, Bran: 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*. London, UK : Springer-Verlag, 1998, S. 250–260. – ISBN 3-540-65075-X

[Senf 2006] Senf, Matthias: *Entwurf und Implementierung einer dynamischen Serviceplattform für eingebettete Systeme - Am Beispiel eines mobilen Messdatenauswertungssystems für Ruderboote*, Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam, Masterarbeit, February 2006. – in Zusammenarbeit mit dem Institut für Forschung und Entwicklung von Sportgeräten (FES) in Berlin

[Sha u. a. 1998] Sha, L. ; Goodenough, J.B. ; Pollak, B.: Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems. In: *The J. Defense Software Eng. vol. 11* (1998), april, S. 716

[Sha u. a. 1989] Sha, L. ; Rajkumar, R. ; Lehoczky, J. ; Ramamritham, K.: Mode Change Protocols for Priority-Driven Preemptive Scheduling. Amherst, MA, USA : University of Massachusetts, 1989. – Forschungsbericht. – S. 243–264

[Sha 1998] Sha, Lui: Dependable System Upgrade. In: *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*. Washington, DC, USA : IEEE Computer Society, 1998, S. 440–8. – ISBN 0-8186-9212-X

[Sha 2001] Sha, Lui: Using Simplicity to Control Complexity. In: *IEEE Software* 18(4) (2001), S. 20–28

[Shaw und Garlan 1996] Shaw, M. ; Garlan, D.: *Software Architecture*. Prentice Hall, New York, 1996

[Silva u. a. 2006] Silva, Dilma D. ; Krieger, Orran ; Wisniewski, Robert W. ; Waterland, Amos ; Tam, David ; Baumann, Andrew: K42: an infrastructure for operating system research. In: *SIGOPS Oper. Syst. Rev.* 40 (2006), Nr. 2, S. 34–42. – ISSN 0163-5980

[Smith 1982] Smith, B.C.: *Procedural Reflection in Programming Languages*, Mass. Inst. of Technology, Dissertation, January 1982

[Stewart u. a. 1993] Stewart, David B. ; Volpe, Richard A. ; Khosla, Pradeep: Design of Dynamically Reconfigurable Real-Time Software using Port-Based Objects / Robotics Institute, Carnegie Mellon University. Pittsburgh, PA, July 1993 (CMU-RI-TR-93-11). – Forschungsbericht

[Systems und at Hasso-Plattner-Institute 2008] Systems, OSM O. ; Hasso-Plattner-Institute, Middleware G. at: *OSM Movies*. http://www.dcl.hpi.uni-potsdam.de/media/. 2008

[Szyperski 1999] Szyperski, Clemens: *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, 1999. – ISBN 0-201-17888-5

[Thai 1999] Thai, Thuan L. ; Oram, Andy (Hrsg.): *Learning DCOM*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1999. – Designed By-Nancy Priest. – ISBN 1565925815

[The Loom.NET Project 2005] The Loom.NET Project: *Loom.Net Homepage*. http://www.rapier-loom.net. 2005

[Tiedt 2007] Tiedt, Sebastian: *Adaptive Dienstintegration in mobilen Ad-hoc Netzwerken*, Universität Potsdam, Diplomarbeit, 2007

[Tröger und Polze 2003] Tröger, Peter ; Polze, Andreas: Object and Process Migration in .NET. In: *Procceedings of IEEE Workshop on Object-Oriented Realtime Dependable Systems*. Guadalajara, January 2003, S. 139

[Tröger u. a. 2008] Tröger, Peter ; Rasche, Andreas ; Feinbube, Frank ; Wierschke, Robert: SOA Meets Robots - A Service-Based Software Infrastructure For Remote Laboratories. Potsdam : Proceedings of 2nd International Workshop on e-learning and Virtual and Remote Laboratories 2008, February 2008

[Ulbrich u. a. 2003] Ulbrich, Andreas ; Weis, Torben ; Geihs, Kurt ; Becker, Christian: DotQoS - A QoS Extension for .NET Remoting. In: *International Workshop on Quality of Service (IWQoS)* 2707 (2003), S. 363–380. – URL `http://www.vs.uni-kassel.de/publications/2003/UWGB03`. – Lecture Notes in Computer Science (LNCS), Monterey, CA, 2003. Springer.

[Vaidyanathan und Trahan 2004] Vaidyanathan, Ramachandran ; Trahan, Jerry L.: *Dynamic Reconfiguration: Architectures and Algorithms (Series in Computer Science (Kluwer Academic/Plenum Publishers).)*. Plenum Publishing Co., 2004. – ISBN 0306481898

[Vanegas u. a. 1998] Vanegas, Rodrigo ; Zinky, John A. ; Loyall, Joseph P. ; Karr, David ; Schantz, Richard E. ; Bakken, David E.: 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)*. The Lake District, England, September 1998, S. 15–18

[Voss und Eigemann 2001] Voss, Michael J. ; Eigemann, Rudolf: High-level adaptive program optimization with ADAPT. In: *ACM SIGPLAN Notices* 36 (2001), Nr. 7, S. 93–102

[Want u. a. 1992] Want, Roy ; Hopper, Andy ; Falcao, Veronica ; Gibbons, Jonathan: The active badge location system. In: *ACM Trans. Inf. Syst.* 10 (1992), Nr. 1, S. 91–102. – ISSN 1046-8188

[Wegdam 2003] Wegdam, Maarten: Dynamic reconfiguration and load distribution in component middleware. (2003). – URL `http://purl.org/utwente/41469`

[Weiser 1991] Weiser, M.: The Computer for the 21st Century. In: *Scientific American* 43 (1991), September, Nr. 3, S. 66–75

[Wermelinger 1997] Wermelinger, Michel: A Hierarchic Architecture Model for Dynamic Reconfiguration. In: *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*. 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA : IEEE, 1997, S. 243–254. – URL `citeseer.nj.nec.com/wermelinger97hierarchic.html`

[Widmer 2007] Widmer, Sven: *Entwicklung eines Betriebssystems mit ECMA-335*, Universität Potsdam, Diplomarbeit, 2007

[Woodside und Menascé 2006] Woodside, C. M. ; Menascé, Daniel A.: Guest Editors' Introduction: Application-Level QoS. In: *IEEE Internet Computing* 10 (2006), Nr. 3, S. 13–15

[Zinky und Bakken 1995] Zinky, J. ; Bakken, D.: *Overview of Quality of Service for Distributed Objects*. 1995

[Zinky u. a. 1997] Zinky, John A. ; Bakken, David E. ; Schantz, Richard E.: Architectural support for quality of service for CORBA objects. In: *Theory and Practice of Object Systems* 3 (1997), Nr. 1