Architektur für die Automation der Managementinstrumentierung bausteinbasierter Anwendungen

Dissertation

an der
Fakultät für Mathematik und Informatik
der
Ludwig-Maximilians-Universität München

vorgelegt von

Rainer Hauck

Tag der Einreichung: 4. Juli 2001
Tag der mündlichen Prüfung: 23. Juli 2001

1. Berichterstatter: **Professor Dr. Heinz-Gerd Hegering**, Universität München
2. Berichterstatter: **Professor Dr. Martin Wirsing**, Universität München

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Lehrstuhl für Kommunikationssysteme und Systemprogrammierung des Instituts für Informatik der Ludwig-Maximilians-Universität München.

Besonderer Dank gebührt meinem Doktorvater, Herrn Prof. Dr. Heinz-Gerd Hegering, für seine hervorragende, unermüdliche Betreuung und die zahlreiche Diskussionen, Anmerkungen und Anregungen, die mir auf meinem nicht immer einfachen Weg geholfen haben und wesentlich zur Steigerung der Qualität der Arbeit beigetragen haben. Danken möchte ich insbesondere auch dafür, daß er trotz hoher Belastung immer die Zeit gefunden hat, mehrere Versionen der Arbeit zu lesen und ausgesprochen konstruktiv zu kritisieren.

Ebenso herzlich möchte ich mich bei Herrn Prof. Dr. Martin Wirsing bedanken, dessen zahlreiche konstruktive Anmerkungen und Verbesserungsvorschläge einen wesentlichen Anteil zum Gelingen dieser Arbeit beigetragen haben.

Auch dem gesamten MNM-Team gebührt Dank für die große Diskussionsbereitschaft und Unterstützung sowie die hervorragende Arbeitsatmosphäre. Insbesondere möchte ich mich bedanken für die große Entlastung von vielen Aufgaben im letzten halben Jahr meiner Lehrstuhltätigkeit, die eine Konzentration auf die Fertigstellung dieser Arbeit ermöglicht hat. Besonderer Dank gebührt meinen Kollegen Igor Radisic, Helmut Reiser und Holger Schmidt, die unzählige Vorversionen der Arbeit gelesen und ausgiebig diskutiert haben.

Nicht zuletzt möchte ich mich bei meiner Familie und insbesondere bei meinen Eltern bedanken, die mir diesen Weg ermöglicht haben und mich immer tatkräftig unterstützt haben.

München, im Juli 2001

Zusammenfassung
Bedingt durch den aktuell zu beobachtenden Trend zum Application Service Provisioning (ASP) gewinnt die Überwachung nutzer- bzw. dienstorientierter Dienstgüteparameter zunehmend an Bedeutung. Viele der heutzutage verfügbaren Werkzeuge für das Anwendungsmanagement basieren jedoch auf einer Überwachung des Netzverkehrs oder der Systeme, auf denen die zu überwachenden Anwendungen ausgeführt werden. Eine Ableitung nutzer- bzw. dienstorientierter Parameter aus den so ermittelten Informationen ist in den meisten Fällen nicht möglich. Einzig mit Hilfe von Anwendungsinstrumentierung kann die erforderliche nutzer- bzw. dienstorientierte Information ermittelt werden. Das mit ausgesprochen hohem Aufwand verbundene manuelle Einfügen von Managementanweisungen in den Quelltext der zu überwachenden Anwendung verhindert derzeit aber einen verbreiteten Einsatz dieser Techniken.

Die vorliegende Arbeit schlägt zur Lösung dieses Problemkreises eine Architektur vor, die den durch eine Managementinstrumentierung bedingten Aufwand für den Fall bausteinbasierter Anwendungen wesentlich verringert. Das Einfügen von Managementanweisungen in die zu überwachende Anwendung kann vollständig automatisiert durch die Entwicklungsumgebung vorgenommen werden. Der Anwendungsentwickler muß lediglich die Interaktionen identifizieren, die den Beginn und das Ende zu überwachender Transaktionen darstellen. Dies kann mit sehr geringem Aufwand während des Customizings der Oberflächenbausteine der Anwendung erfolgen. Durch die automatische Instrumentierung wird ein hoher Detailgrad der Messung erreicht, der es gestattet, auftretende Fehler bis zum verursachenden Baustein zurückzuverfolgen.

Ein weiterer wesentlicher Aspekt der vorgeschlagenen Architektur ist die automatische Korrelation von Managementinformation. Auch wenn mehrere Transaktionen parallel ausgeführt werden, ist es möglich, anhand der ausführenden Kontrollflüsse eine vollautomatische und eindeutige Zuordnung jedes Meßwertes zu einer übergeordneten Transaktion vorzunehmen. Auch hierfür ist kein manuelles Eingreifen mehr erforderlich.

Die Arbeit definiert darüber hinaus die Schnittstellen, über die die erforderliche Managementinformation übergeben werden kann. Desweiteren werden Methodiken angegeben, die sowohl für den Bausteinentwickler als auch für den Anwendungsentwickler detailliert vorgeben, welche Schritte bei der Erstellung von Bausteinen bzw. Anwendungen durchzuführen sind.

Eine prototypische Implementierung der vorgeschlagenen Architektur zeigt deren Umsetzbarkeit. Weiterhin konnte mit Hilfe der prototypischen Implementierung gezeigt werden, daß die Beeinflussung der Anwendung durch das Einfügen von Managementanweisungen zu einer nur unwesentlichen Beeinflussung der zu überwachenden Anwendung führt.

INHALT

1 Einführung **1**

1.1 Motivation 1
1.2 Aufgabenstellung 2
1.3 Aufbau und Ergebnisse der Arbeit 4

2 Begriffsbildung und Umfeld **7**

2.1 Bausteinorientierte Anwendungsentwicklung 7
2.1.1 Existierende Architekturen 8
2.1.1.1 Verknüpfung durch Adapter 8
2.1.1.2 Verknüpfung im Client 15
2.1.2 Begriffsbildung 19
2.2 Anwendungsmanagement 23
2.2.1 Anwendungsüberwachung 24
2.2.2 Transaktionsüberwachung 25
2.3 Dienstorientierung und -management 26

3 Anforderungen an die Überwachung bausteinbasierter Anwendungen **31**

3.1 Allgemeine Anforderungen an die Überwachung von Anwendungsdiensten 31
3.1.1 Modellierung von Anwendungsdiensten 31
3.1.2 Ermittlung der Anforderungen 33
3.2 Spezielle Anforderungen bausteinbasierter Anwendungen 35
3.2.1 Modellierung bausteinbasierter Anwendungsdienste 35
3.2.2 Ermittlung der Anforderungen 37
3.2.2.1 Anforderungen aus der rekursiven Struktur der Anwendung 37

3.2.2.2 Anforderungen aus den beteiligten Rollen 37
3.2.2.3 Anforderungen aus den verschiedenen Varianten von Anwendungslogik 39
3.2.2.4 Anforderungen aus den verschiedenen Arten von Bausteinen 40
3.3 Zusammenfassung 41

4 Status Quo: Überwachung von Anwendungsdiensten 43

4.1 Klassifikation unterschiedlicher Ansätze 43
4.1.1 Klassifikation der Anwendungsüberwachung 43
4.1.1.1 Überwachung des Netzverkehrs 44
4.1.1.2 Überwachung von Systemparametern 45
4.1.1.3 *Client*-seitige Anwendungsüberwachung 46
4.1.1.4 Überwachung der Gesamtanwendung 47
4.1.1.5 Zusammenfassung 49
4.2 Untersuchung aktueller Ansätze zur Anwendungsüberwachung 50
4.2.1 Standards 50
4.2.1.1 Internet Engineering Task Force 50
4.2.1.2 Distributed Management Task Force 56
4.2.1.3 Open Group 61
4.2.1.4 TeleManagement Forum 67
4.2.2 Forschungsansätze 68
4.2.2.1 Biaggiolini und Harms: Automatisierung des Fehlermanagements bausteinbasierter Anwendungen 68
4.2.2.2 Neumair: GAMOCs 69
4.2.2.3 Kaiser: Bestimung der Verfügbarkeit von anwendungsorientierten Diensten 70
4.2.2.4 Kar, Keller und Calo: Automatische Ermittlung von Abhängigkeiten im Anwendungsmanagement 71
4.2.2.5 Frolund et al.: SoLOMon 71
4.2.2.6 Hellerstein et al.: ETE 72
4.2.2.7 Weitere Forschungsansätze 72
4.2.3 Herstellerspezifische Lösungen 73
4.2.3.1 Tivoli 73
4.2.3.2 Candle 78
4.2.3.3 Weitere herstellerspezifische Lösungen 79
4.3 Zusammenfassung: Möglichkeiten und Defizite existierender Ansätze . . 80

5 Überwachung bausteinbasierter Anwendungen 83

5.1 Lösungsansätze . . . 83
5.1.1 Wesentliche Designentscheidungen . . . 84
5.1.2 Untersuchte Ansätze . . . 85
5.2 Erster Ansatz: Komposition von Managementschnittstellen instrumentierter Bausteine . . . 86
5.2.1 Idee . . . 86
5.2.2 Verknüpfung von Managementinformation . . . 87
5.2.3 Erstellung der Managementlogik . . . 91
5.2.4 Bewertung . . . 92
5.3 Zweiter Ansatz: Automation der Managementinstrumentierung bausteinbasierter Anwendungen . . . 95
5.3.1 Idee . . . 95
5.3.2 Identifikation von Meßpunkten . . . 97
5.3.2.1 Überwachung von Benutzertransaktionen . . . 97
5.3.2.2 Überwachung von Subtransaktionen . . . 102
5.3.3 Zuordnung der Teilinformationen . . . 103
5.3.3.1 Zuordnung innerhalb eines Kontrollflusses . . . 104
5.3.3.2 Zuordnung bei Einbeziehung mehrerer Kontrollflüsse . . . 105
5.3.4 Anforderungen an die Implementierung der Bausteinarchitektur . . . 112
5.3.5 Bewertung . . . 113
5.4 Architektur für die Überwachung bausteinbasierter Anwendungen . . . 115
5.4.1 Instrumentierung von Bausteinen . . . 115
5.4.1.1 Integration aktiver Bausteine . . . 116
5.4.1.2 Instrumentierung von Oberflächenbausteinen . . . 116
5.4.1.3 Identifikation interner Fehlerzustände eines Bausteins . . . 117
5.4.2 Architektur für die Ermittlung der Managementinformation . . . 118
5.4.2.1 Architekturübersicht . . . 118
5.4.2.2 Definition von Schnittstellen . . . 121
5.4.3 Architektur für die Automation der Managementinstrumentierung 128
5.4.3.1 Architekturübersicht . . . 128
5.4.3.2 Methodiken . . . 131
5.5 Bewertung . . . 136
5.5.1 Vergleich mit der ARM API . . . 136
5.5.2 Leistungsbewertung . . . 137

5.5.2.1 Start und Stop von Subtransaktionen 138
5.5.2.2 Erzeugung und Beendigung von Kontrollflüssen 140
5.6 Bestimmung der Verfügbarkeit bausteinbasierter Anwendungsdienste . . 141

6 Prototypische Realisierung 143

6.1 Implementierung der Architektur für die Automation der Managementinstrumentierung 143
6.1.1 Erweiterung des Entwicklungswerkzeuges Beanbox 144
6.1.2 Instrumentierung ausgewählter JavaBeans 145
6.1.2.1 Instrumentierung von Eingabebausteinen 146
6.1.2.2 Instrumentierung von Präsentationsbausteinen 147
6.1.2.3 Instrumentierung aktiver Bausteine 147
6.2 Implementierung der Architektur für die Ermittlung der Managementinformation 148
6.2.1 Implementierung des Meßobjekts 148
6.2.2 Instrumentierung von Systemklassen der Java Virtual Machine . . 151
6.2.3 Implementierung eines prototypischen Managementagenten . . . 152
6.2.4 Implementierung einer prototypischen Managementanwendung . 152
6.3 Beispielanwendungen 153
6.3.1 Parallele Ausführung unterschiedlicher BTAs 154
6.3.2 Aktive Beans 158

7 Ausblick 161

7.1 Zusammenfassung und wesentliche Ergebnisse der Arbeit 161
7.2 Zukünftige Forschungsfragestellungen 164

Abkürzungsverzeichnis 167

Abbildungsverzeichnis 171

Literaturverzeichnis 175

Index 185

Kapitel 1

Einführung

1.1 Motivation

In jüngster Zeit ist auch im IT-Bereich ein sich stetig verstärkender Trend zur Dienstorientierung zu verzeichnen. Aufgaben, die außerhalb der Kernkompetenzen eines Unternehmens liegen, werden nicht mehr selbst erbracht, sondern als Dienst von Dienstleistern eingekauft und zur Verfügung gestellt. Begriffe wie *Outsourcing* oder *Service Provisioning* rücken zunehmend in den Vordergrund.

Bedingt durch die ständig zunehmende Vernetzung, die erheblich gestiegenen und weiterhin steigenden Übertragungskapazitäten sowie die weiterhin ansteigende Komplexität der benötigten Anwendungen [BHP+ 00], läßt sich dieser Trend in den letzten Jahren insbesondere im Bereich der in einem Unternehmen eingesetzten Anwendungen beobachten. Statt des Betriebs von Anwendungen durch eigene IT-Abteilungen werden nun auch Anwendungdienste von Dienstleistern bezogen. Diese sogenannten *Application Service Provider (ASP)* bieten Zugriff auf den Anwendungsdienst und gewährleisten die Verfügbarkeit des Dienstes mit einer bestimmten Güte.

Die genaue Spezifikation der anzubietenden Dienste sowie der Dienstgüten gewinnt somit zunehmend an Bedeutung. Im Rahmen von Dienstbeschreibungen und -vereinbarungen muß genau festgehalten werden, welche Funktionalität und Dienstgüte vom Dienstnehmer gefordert werden kann. Verletzungen dieser Vereinbarung führen zu Konsequenzen, die – je nach Vereinbarung – von Rabatten für den Kunden über die Zahlung von Konventionalstrafen durch den ASP bis zur Auflösung des Vertrags und somit zum Verlust des Kunden an andere Anbieter reichen können. Um die Erfüllung bzw. Nicht-Erfüllung dieser Vereinbarungen nachweisen zu können, ist eine Überwachung des Dienstes erforderlich. Diese findet üblicherweise durch Überwachung der den Dienst erbringenden Anwendung statt.

Unter dem Begriff Anwendungsüberwachung sollen in dieser Arbeit in Anlehnung an [HAN 99] die Aufgaben des Fehler- und Leistungsmanagements verstanden werden. Die Managementinformation wird hierbei unmittelbar von der zu überwachenden Anwendung zur Verfügung gestellt und nicht, wie z.B. beim Server bzw. Desktop Management mittelbar durch die darunterliegenden Systeme.

Betrachtet man die heute für die Überwachung von Anwendungen eingesetzten Verfahren, so stellt man fest, daß diese entweder nicht in der Lage sind, die erforderlichen Informationen zu liefern, oder aber einen hohen Realisierungsaufwand erfordern, der nur in wenigen Ausnahmefällen gerechtfertigt ist. Dies liegt an der großen Heterogenität der Anwendungen, also den großen Unterschieden, die zwischen den unterschiedlichen Anwendungen bestehen und somit generische Ansätze erheblich erschweren.

Diese Feststellung wirkt sich vor dem Hintergrund zunehmender Bausteinorientierung in der Entwicklung von Anwendungen noch gravierender aus. Dienstnehmer verlangen immer weniger nach vorkonfektionierten Massendiensten, sondern fordern spezielle, nach ihren Wünschen gefertigte Dienste (sog. Individualdienste). Hierbei ist die *time-to-market*, also die Zeit, die vergeht, bis ein Dienst tatsächlich angeboten werden kann, ein entscheidender Wettbewerbsfaktor [AdWi 96]. Die Dienstleister nutzen daher vorgefertigte Bausteine, die nach Kundenwünschen konfiguriert und zu einer den gewünschten Dienst erbringenden Anwendung zusammengefügt werden können. Dies führt zu einer erheblichen Beschleunigung des Entwicklungsprozesses bei gleichzeitiger Erhöhung der Qualität der entstandenen Software durch Einsatz bereits erprobter Bausteine. Weiterhin wird es somit möglich, weniger gut ausgebildete Programmierer zur Erstellung dieser Anwendungen einzusetzen, da z.B. grafische Entwicklungsumgebungen einen Großteil der anfallenden Arbeiten abnehmen können.

1.2 Aufgabenstellung

Die derzeit zur Überwachung von Anwendungen bzw. Anwendungsdiensten zum Einsatz kommenden (Instrumentierungs-) Verfahren sind nicht in der Lage, den aus der oben dargestellten Situation resultierenden Anforderungen gerecht zu werden. Entweder verhindert der erhebliche Realisierungsaufwand eine weite Verbreitung oder aber die Verfahren sind überhaupt nicht für den Bedarf bausteinorientierter Anwendungsentwicklung ausgelegt, da sie z.B. Zugriff zum *Source Code* der Anwendung erfordern.

Ziel der vorliegenden Arbeit ist die Erstellung einer Architektur, die den Aufwand für die Instrumentierung einer Anwendung mit Managementfunktionalität soweit verringert, daß er kein Hindernis für die Umsetzung einer effektiven Anwendungsüberwachung mehr darstellt. Idee ist es, hierbei die oben beschriebene Bausteinstruktur zukünftiger Anwendungen für Managementzwecke geeignet auszunutzen.

Die Managementinstrumentierung eines einzelnen Bausteins kann mit relativ wenig Aufwand durch seinen Entwickler erfolgen. Die Aggregation und Korrelation der Managementinformationen der einzelnen Bausteine einer Anwendung kann dann weitestgehend automatisch in einer Management-Komponente erfolgen, der die Zusammenhänge innerhalb der Anwendung bekannt sind. Hierzu bedarf es einer Erweiterung der eingesetzten Entwicklungsumgebung, so daß diese beim Zusammenfügen der Bausteine automatisch Management-Code in die erstellte Anwendung einfügt bzw. die Abhängigkeiten zwischen den einzelnen Bausteinen festhält.

Die resultierende Architektur verringert den Aufwand – sowohl für den Anwendungsentwickler als auch für den Bausteinentwickler – erheblich und schafft somit die Voraussetzungen für eine verstärkte Nutzung von Instrumentierungstechniken für die Anwendungsüberwachung. Insbesondere wird nur für einige wenige Bausteine eine spezielle Instrumentierung gefordert. Der Großteil der Bausteine läßt sich ohne jegliche Veränderung in die Überwachung integrieren.

Die hierbei vornehmlich zu lösenden Fragestellungen sind im folgenden aufgeführt:

- Welche Anforderungen sind an die Überwachung von Anwendungen zu stellen? Welche Managementinformationen und -funktionalität ist für die Überwachung von Anwendungsdiensten erforderlich?
- Welche besonderen Managementanforderungen ergeben sich aus dem Szenario der bausteinorientierten Anwendungsentwicklung, die über die Anforderungen „normaler“ Anwendungen hinausgehen?
- Welche Informationen müssen überwacht werden? Wie muß eine Managementschnittstelle für einzelne Bausteine bzw. für die gesamte Anwendung beschaffen sein?
- Welche Ansätze existieren bereits für die Anwendungsüberwachung und insbesondere für die Überwachung bausteinbasierter Anwendungen? Welche Nachteile existierender Lösungen lassen sich angeben?
- Wie kann die Information einzelner Bausteine zu Information über die Gesamtanwendung aggregiert und korreliert werden?
- Nach welcher Methodik sollten Anwendungs- und Bausteinentwickler bei der Erstellung von Anwendungen und Bausteinen verfahren, um Managementfunktionalität in die entstehenden Bausteine bzw. Anwendungen zu integrieren?
- Wie ist die Entwicklungsumgebung zu erweitern, um eine weitgehend automatische Managementinstrumentierung sowie die automatische Aggregation und Korrelation von Managementinformation zu gestatten?

1.3 Aufbau und Ergebnisse der Arbeit

In dieser Arbeit wird aufgezeigt, wie der Aufwand für die Managementinstrumentierung bausteinbasierter Anwendungen erheblich reduziert werden kann. Die hierbei angewandte Vorgehensweise wird in Abb. 1.1 graphisch dargestellt (die wesentlichen Ergebnisse der Arbeit sind durch graue Hinterlegung gekennzeichnet) und im folgenden kurz beschrieben:

Kapitel 2 gibt einen Überblick über typische Vertreter von Bausteinarchitekturen. Aufgrund erheblicher Unterschiede der unterschiedlichen Architekturen wird daraufhin ein für den weiteren Verlauf der Arbeit eindeutiger begrifflicher Rahmen festgelegt und darüber hinaus durch weitere Begriffsdefinitionen aus den Bereichen des Anwendungs- und Dienstmanagements ergänzt.

Ausgehend von der Beobachtung zweier Trends, nämlich dem zum *Application Service Provisioning (ASP)* und dem zur Bausteinorientierung, werden in Kapitel 3 die Anforderungen, die an eine Managementlösung für die Überwachung bausteinbasierter Anwendungen gestellt werden müssen, ermittelt. Die Szenarien des ASP und der bausteinorientierten Anwendungsentwicklung werden in einer allgemeinen Form modelliert und ausführlich analysiert. Wesentliche Anforderungen, die hierbei ermittelt werden, sind die Forderung nach Überwachung nutzerorientierter Parameter bei gleichzeitiger Möglichkeit der detaillierten Ermittlung von Fehlerquellen sowie die Möglichkeit, bei Identifikation eines Problems eines einzelnen Bausteins, die Auswirkungen auf die Verfügbarkeit der unterschiedlichen angebotenen Dienste ermitteln zu können.

Im Falle bausteinorientierter Anwendungen stellt die Überwachung der einzelnen Bausteine den geeigneten Detaillierungsgrad dar. Insbesondere die Überwachung von Benutzertransaktionen sowie der von den einzelnen Bausteinen erbrachten Subtransaktionen ist von großer Bedeutung. Die zentrale Forderung, die sich aus der Analyse der Szenarien ergibt, ist jedoch die Forderung nach weitgehend werkzeuggestützter und automatisierter Managementinstrumentierung der Anwendungen.

Die ermittelten Anforderungen werden in Kapitel 4 den aktuell existierenden Ansätzen gegenübergestellt. Hierzu wird eine Klassifikation der unterschiedlichen Ansätze nach Art der Informationsgewinnung angegeben, in die die verschiedenen Lösungen von Standardisierungsorganisationen und Herstellern sowie aktuelle Forschungsansätze eingeordnet werden können. So wird es möglich, unabhängig von einzelnen Produkten die eigentlichen Konzepte zu untersuchen und zu bewerten. Es zeigt sich, daß alle derzeit verfügbaren Ansätze entweder nicht die Information liefern können, die für eine sinnvolle Anwendungsüberwachung erforderlich ist, oder aber sich der Aufwand für ihre Realisierung als zu hoch erweist.

Daher wird in Kapitel 5 eine Lösung vorgestellt, die die oben beschriebenen Defizite nicht aufweist und die gestellten Anforderungen erfüllen kann. Zunächst werden die bei der Auswahl einer geeigneten Lösung zu treffenden Designentscheidungen dargelegt. Ein erster Ansatz, die *Komposition von Managementschnittstellen instrumentierter Bausteine* wird vorgestellt. Hierbei wird eine Instrumentierung der grundlegenden Bausteine vorausgesetzt. Aufgrund der Abhängigkeiten zwischen Bausteinen lassen sich die Managementschnittstellen zusammengesetzter Bausteine hieraus ohne weitere Instrumentierung bestimmen. Dieser Ansatz erwies sich allerdings im Verlauf der Untersuchungen als nur bedingt geeignet, die gestellten Anforderungen zu erfüllen. Aus diesem Grund wird ein zweiter Ansatz, die *automatische Instrumentierung bausteinbasierter Anwendungen* untersucht: Dieser verzichtet vollständig auf die Instrumentierung einzelner Bausteine. Stattdessen werden Meßpunkte mit Hilfe einer erweiterten Entwicklungsumgebung größtenteils automatisch in die zu erstellende Anwendung eingefügt. Auch dieser Ansatz kann letztlich aber nicht alle Anforderungen erfüllen. Die im Anschluß daran vorgestellte Architektur basiert somit auf einer Kombination beider Ansätze und vereint deren Vorteile zu einer Lösung, die den aufgestellten Anforderungen umfassend gerecht werden kann. Die Vorstellung der Architektur gliedert sich in zwei Teile: Zunächst wird die *Architektur für die Ermittlung der Managementinformation* dargestellt. Diese spezifiziert Managementschnittstellen, die an die im Rahmen der Arbeiten zur *Application Response Measurement API (ARM)* [C807] entstandene Schnittstelle angelehnt sind. Es wird also nicht versucht, existierende Ansätze zu ersetzen, sondern geeignet zu erweitern und zu verbessern. Weiterhin wird hier angegeben, wie bestimmte Systembibliotheken zu erweitern sind, um die einfache Überwachung zu gewährleisten. Der zweite Teil der vorgestellten Architektur, die *Architektur zur Automation der Managementinstrumentierung* legt fest, wie eine Entwicklungsumgebung erweitert werden muß, um einen Großteil der Managementinstrumentierung vollständig automatisiert durchführen zu können. Weiterhin werden sowohl für den Anwendungsentwickler als auch für den Bausteinentwickler einfache Methodiken vorgegeben, nach denen bei der Erstellung von Anwendungen bzw. Bausteinen zu verfahren ist, um mit geringem Aufwand managementinstrumentierte Anwendungen zu erhalten.

Um die Anwendbarkeit der vorgeschlagenen Lösung zu demonstrieren wird in Kapitel 6 eine prototypische Implementierung des Ansatzes vorgestellt. Es handelt sich um eine Erweiterung einer Entwicklungsumgebung für *Java-Beans*, mit deren Hilfe instrumentierte Anwendungen mit geringem Aufwand für Anwendungs- und Bausteinentwickler erstellt werden können.

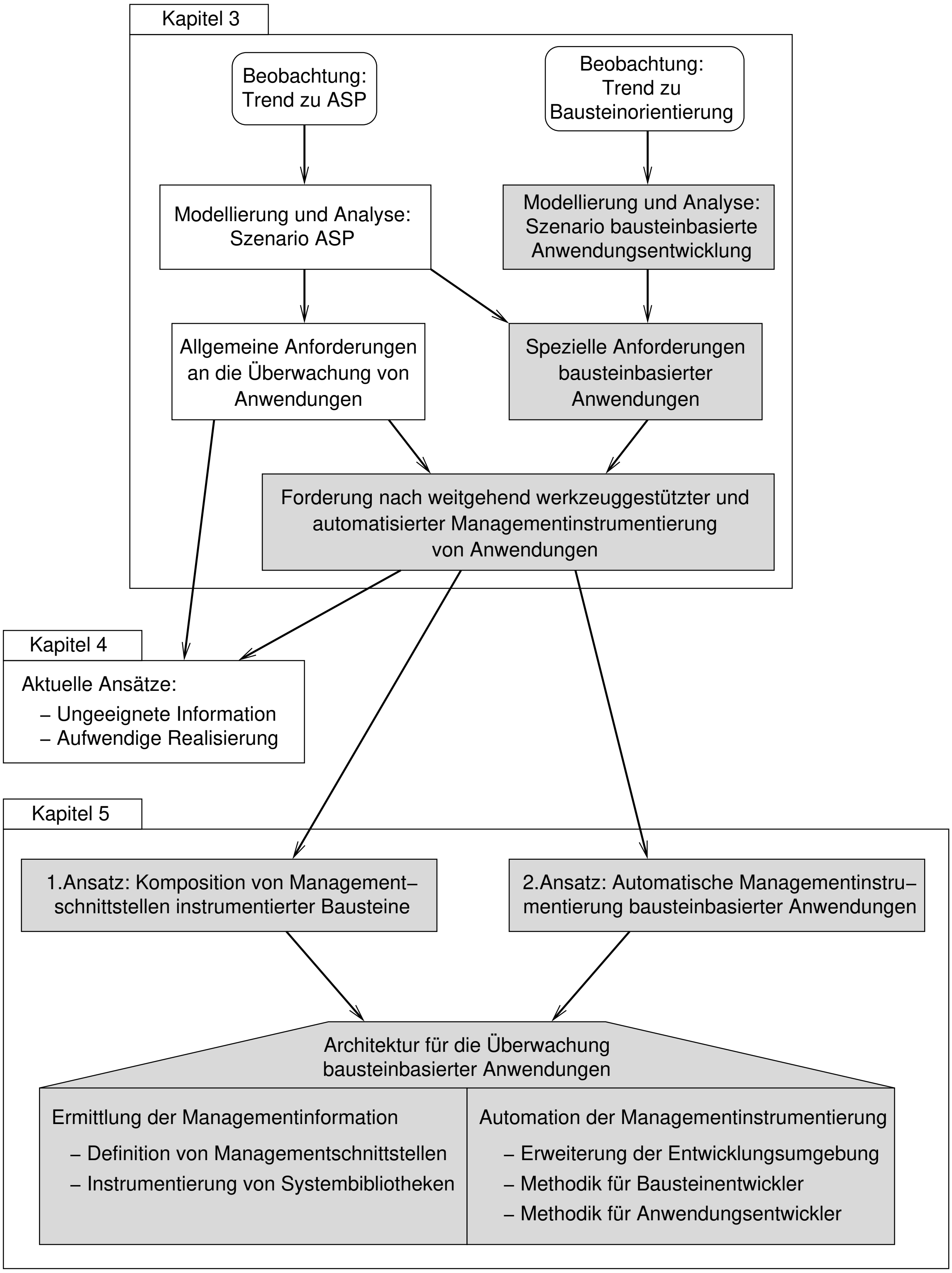

Abbildung 1.1: Graphische Darstellung der Vorgehensweise

Kapitel 2

Begriffsbildung und Umfeld

Das folgende Kapitel definiert die wesentlichen, für den weiteren Verlauf der Arbeit benötigten Begriffe aus den Bereichen der bausteinorientierten Anwendungsentwicklung, dem Anwendungsmanagement sowie der Dienstorientierung. Dies ist erforderlich, da in diesen Bereichen vielfach noch keine allgemein akzeptierten Begriffe existieren und sich somit teilweise erheblich unterschiedliche oder sogar widersprüchliche Interpretationen einzelner Begriffe ergeben. Die hier getroffenen Definitionen legen die Bedeutung der Begriffe für den weiteren Verlauf der Arbeit eindeutig fest.

2.1 Bausteinorientierte Anwendungsentwicklung

In den letzten Jahren ist eine Vielzahl von unterschiedlichen Architekturen für die bausteinorientierte Anwendungsentwicklung entstanden. Auch wenn die Architekturen erhebliche Unterschiede hinsichtlich ihrer Realisierung aufweisen, verfolgen sie dennoch weitgehend ähnliche Konzepte. So kennen alle Architekturen Bausteine als Einheit der Wiederverwendung und Verteilung, aus denen Anwendungen zusammengesetzt werden können. Diese verfügen über wohldefinierte Schnittstellen, über die die Funktionalität des Bausteins aufgerufen werden kann. Im Rahmen des *Customizing* können die einzelnen Bausteine für die Verwendung in einer konkreten Anwendung angepaßt werden.

Im folgenden Abschnitt werden zunächst die derzeit überwiegend eingesetzten bzw. aktuell diskutierten Bausteinarchitekturen *Java Beans*, *Enterprise JavaBeans (EJB)*, *ActiveX* sowie das *CORBA Component Model (CCM)* vorgestellt [Grif 98]. Hierbei wird besonderer Wert auf die im weiteren Verlauf der Arbeit bedeutsamen Unterschiede bezüglich der Verknüpfung einzelner Bausteine zu lauffähigen Anwendungen gelegt. Ebenso wird dargelegt, wie die bereits erwähnten Begriffe des Bausteins, der Schnittstelle sowie des *Customizings* im Rahmen der jeweiligen Architektur verstanden werden. Sogenannte *Compound Document Models* werden nicht berücksichtigt, da diese lediglich eine Integration unterschiedlicher Anwendungen in eine gemeinsame Oberfläche und keine echte bausteinorientierte Anwendungsentwicklung gestatten. Im Anschluß daran werden Begriffsdefinitionen vorgenommen, die eine einheitliche Verwendung der Begriffe aus dem Bereich der bausteinorientierten Anwendungsentwicklung im Rahmen der vorliegenden Arbeit sicherstellen.

2.1.1 Existierende Architekturen

Bei der Untersuchung existierender Architekturen zeigt sich, daß diese sich hinsichtlich der Erstellung von Anwendungen in zwei Klassen unterteilen lassen: Architekturen, bei denen die Verknüpfung zwischen Bausteinen mit Hilfe spezieller Adapter erfolgt, sowie Architekturen, die die Verknüpfung der Bausteine zu lauffähigen Anwendungen innerhalb der *Clients* erfordern. Im weiteren Verlauf der Arbeit wird sich zeigen, daß für die erste der beiden Klassen eine nahezu vollständige Automation der Managementinstrumentierung erreicht werden kann. Auch für die zweite Klasse läßt sich mit Hilfe des vorgestellten Ansatzes eine wesentliche Verringerung des Instrumentierungsaufwandes erzielen.

2.1.1.1 Verknüpfung durch Adapter

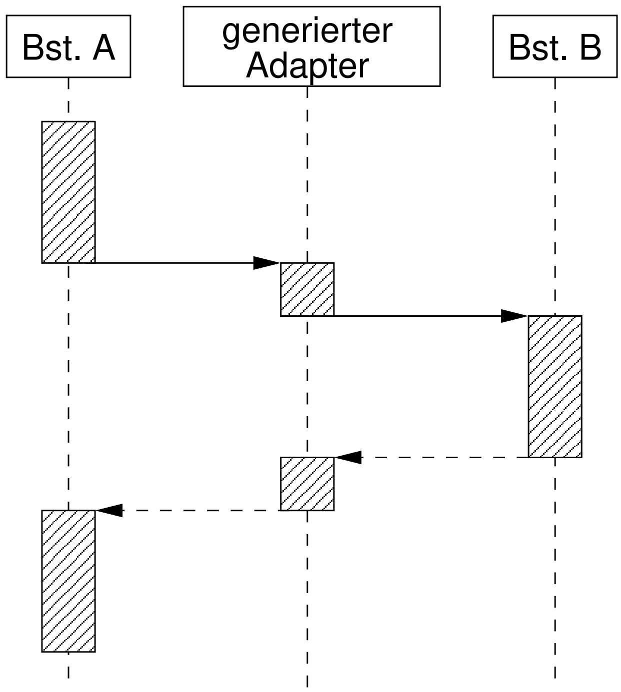

Abbildung 2.1: Verknüpfung von Bausteinen über generierte Adapter

Schreibt eine Bausteinarchitektur nicht nur die Struktur der Aufrufschnittstelle eines Bausteins vor, sondern darüber hinaus auch die Struktur der Ausgangsschnittstelle (z.B. Versenden von *Events*) des Bausteins, so ist es möglich, die Verknüpfung zweier Bausteine mit Hilfe sogenannter Adapter vorzunehmen. Diese können teilweise vollständig von einer Entwicklungsumgebung generiert werden. Dies setzt allerdings voraus, daß sich die Bausteine an strenge Konventionen bezüglich der Namensgebung halten oder Metainformationen bereitstellen, die die von ihnen angebotenen Schnittstellen beschreiben. Eine Entwicklungsumgebung kann so die angebotenen Schnittstellen erkennen und einem Entwickler zur Verfügung stellen. Dieser kann dann (beispielsweise unter Verwendung einer grafischen Oberfläche) die gewünschte Verknüpfung zweier Bausteine definieren; die

Entwicklungsumgebung generiert daraufhin einen die Verknüpfung realisierenden Adapter; es ist häufig keine manuelle Erstellung von *Code* mehr erforderlich.

Die Verknüpfung zweier Bausteine wird in diesem Fall also nicht innerhalb eines Bausteins festgelegt, sondern mit Hilfe von generiertem *Code* außerhalb der Bausteine. Ruft in diesem Fall ein Baustein einen anderen Baustein auf, so ergibt sich ein Ablauf, wie er in dem in Abbildung 2.1 dargestellten Sequenzdiagramm skizziert wird[1]. Baustein A ruft zunächst eine Methode des generierten Adapters auf, der seinerseits wiederum die gewünschte Methode des Bausteins B aufruft. Die Rückkehr aus Baustein B erfolgt ebenfalls zunächst zum generierten Adapter und erst von dort zurück zu Baustein A.

In diesem Fall wird also jeweils vor und nach dem Aufruf eines Bausteins für kurze Zeit ein Stück *Code* ausgeführt, der von der Entwicklungsumgebung generiert wurde. Dies wird im weiteren Verlauf der Arbeit ausgenutzt, um durch automatisches Einfügen von Meßpunkten in den generierten *Code* die Überwachung der Anwendung zu gewährleisten. Die im folgenden vorgestellten Architekturen *JavaBeans* sowie *ActiveX* verfügen über die hierzu erforderliche Möglichkeit, die Verknüpfung zweier Bausteine (zumindest teilweise) automatisch zu generieren.

JavaBeans

Die Firma *Sun* erweiterte Ende 1996 die Programmiersprache *Java* um die *JavaBeans* Spezifikation, die aktuell in Version 1.01 [JavaBeans 1.01] verfügbar ist. Ziel war es hierbei, einen Standard für die Entwicklung von Softwarebausteinen zu schaffen, die mit Hilfe von Entwicklungswerkzeugen zu lauffähigen Anwendungen zusammengesetzt werden können.

- Bausteinbegriff

  Unter einer *JavaBean* wird ein wiederverwendbarer Softwarebaustein verstanden, der mit Hilfe von Entwicklungswerkzeugen visuell manipuliert werden kann. Das Spektrum möglicher Bausteine reicht dabei von sehr einfachen Oberflächenbausteinen, wie beispielsweise *Buttons* oder Eingabefeldern, bis hin zu vollständigen Anwendungen.

  Auch wenn die visuelle Manipulation von Bausteinen ein zentraler Bestandteil des *JavaBeans*-Konzeptes ist, ist es nicht erforderlich, daß alle Bausteine auch zur Laufzeit eine grafische Repräsentation besitzen. Viele *JavaBeans* verfügen über eine grafische Repräsentation, die aber nur zum Zeitpunkt der Anwendungserstellung innerhalb der Entwicklungsumgebung verwendet wird, zur Laufzeit einer daraus erstellten

[1] Die UML gestattet es nicht darzustellen, welches Objekt zu einem bestimmten Zeitpunkt in einem bestimmten Thread ausgeführt wird. Da dies aber im weiteren Verlauf der Arbeit von Bedeutung ist, wird in den folgenden Sequenzdiagrammen (abweichend von der Standard UML Notation) auf die Darstellung des Steuerungsfokus verzichtet und stattdessen schraffierte Rechtecke eingeführt, um den ausführenden Thread zu modellieren. Dies wird in Abschnitt 2.1.2 nochmals ausführlich erläutert

Anwendung hingegen nicht in Erscheinung tritt. Somit ist es möglich, sowohl Oberflächen- als auch *Server*-Bausteine als *JavaBeans* zu implementieren.

- Schnittstellen

  Die Schnittstelle zum Zugriff auf eine *JavaBean* umfaßt die von ihr angebotenen öffentlichen (*Java*)-Methoden. Diese können natürlich beliebige Aufrufparameter beinhalten. Die öffentlichen Attribute einer *Bean* können ebenfalls über entsprechende Methoden ausgelesen und manipuliert werden. Eine Bean übermittelt Änderungen ihres Zustandes mit Hilfe von *Java Events*. Die Benennung der Methoden, Attribute und *Events* ist an eine strenge Namenskonvention gebunden, um die automatische Integration in generische Entwicklungsumgebungen zu gestatten. Die von einer *Bean* erzeugten *Events* können aktuelle Parameterwerte enthalten.

- Customization

  Die Anpassung einzelner *Beans* an die konkreten Anforderungen einer zu erstellenden Anwendung erfolgt im Rahmen der sogenannten *Customization*. Dies kann über Veränderung der öffentlichen Attribute einer Bean erfolgen: Für jedes öffentliche Attribut müssen Methoden zum Auslesen und Verändern seines Wertes vorliegen. Diese können – wiederum aufgrund von Namenskonventionen – vom Entwicklungswerkzeug automatisch erkannt werden. Ein Entwicklungswerkzeug kann somit eine Eingabemaske für jeden Baustein generieren, mit deren Hilfe es einem Anwendungsentwickler möglich ist, die gewünschten Konfigurationen vorzunehmen. Alternativ ist es möglich, einen speziellen *Customizer* gemeinsam mit der *Bean* auszuliefern, der vom Entwicklungswerkzeug gestartet werden kann und aufwendigere Anpassungen gestattet.

- Verknüpfung von Bausteinen

  Die Verknüpfung einzelner *Beans* zu einer lauffähigen Anwendung erfolgt mit Hilfe des *Java Event*-Mechanismus. Wie bereits erwähnt kann eine *Bean* jede Änderung ihres Zustandes der Umwelt mit Hilfe eines *Java Events* mitteilen. Erstellt man nun eine sogenannte Adapterklasse, also eine Klasse, die sich als *Event-Listener* der *Bean* registriert und eine beliebige Methode einer anderen *Bean* aufruft, ist die Verbindung der beiden *Beans* bereits hergestellt. Die Verwendung von Adapterklassen ist erforderlich, um den Typ des ausgelösten *Events* an die aufzurufende Methode anpassen zu können und die in den *Events* übergebene Information geeignet auf Parameter der aufgerufenen Methode abbilden zu können.

  Die Erstellung der Adapterklassen erfolgt typischerweise mit Hilfe spezieller Entwicklungsumgebungen. Hierbei handelt es sich entweder um grafische Entwicklungswerkzeuge oder um spezielle Skriptsprachen. In Abbildung 2.2 ist die Verknüpfung zweier *JavaBeans* mit Hilfe des Werkzeugs *Beanbox* dargestellt.

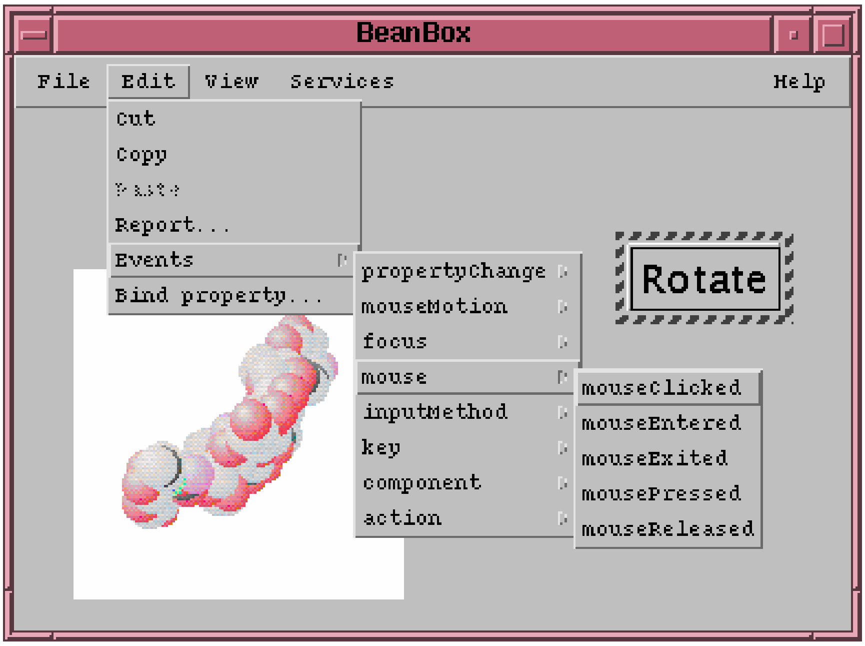

Abbildung 2.2: Verknüpfung von *JavaBeans* mit Hilfe der *Beanbox*

Der Anwendungsentwickler wählt aus den angebotenen *Events* eines Bausteins (in diesem Fall eines *Buttons* mit der Aufschrift *Rotate*) ein geeignetes *Event* aus (im Beispiel `mouseClicked`, also das Anklicken des *Buttons* mit der Maus) und verbindet dieses mit der aufzurufenden Methode des Zielbausteins (im Beispiel die Methode `rotateOnX` eines Bausteins zur dreidimensionalen Darstellung von Molekülen). Die Entwicklungsumgebung generiert daraufhin eine Adapterklasse wie die in Abbildung 2.3 dargestellte (einen sogenannten *Hookup*). Mit Hilfe der `setTarget`-Methode der Adapterklasse kann die *Beanbox* das private Attribut `target` auf den aufzurufenden Baustein setzen. Die Adapterklasse wird als *Event-Listener* des *Rotate-Buttons* registriert, so daß bei Anklicken des *Buttons* die Methode `mouseClicked` der Adapterklasse aufgerufen wird (aus Gründen der Übersichtlichkeit wurden die in diesem Beispiel nicht relevanten Methoden des `MouseListener` *Interface*, also `MouseEntered`, `MouseReleased`, etc., nicht dargestellt). Die Adapterklasse wiederum ruft die Methode `rotateOnX` des Zielbausteins, in diesem Fall also des Bausteins zur Visualisierung von Molekülen auf. Betätigt ein Benutzer nun den *Rotate Button* wird das aktuell dargestellte Molekül in der x-Ebene rotiert.

Darüber hinaus ist es natürlich ebenfalls möglich, auf die Verwendung einer derartigen Entwicklungsumgebung zu verzichten und *Java Beans* durch manuelle Implementierung von Adapterklassen zu lauffähigen Anwendungen zusammenzusetzen.

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.molecule.Molecule;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class ___Hookup_16d4439d80 implements java.awt.event.
       MouseListener, java.io.Serializable {

   public void setTarget(sunw.demo.molecule.Molecule t){
       target = t;
   }

   public void mouseClicked(java.awt.event.MouseEvent arg0){
       target.rotateOnX();
   }

   ...

   private sunw.demo.molecule.Molecule target;
}
```

Abbildung 2.3: Beispiel für eine von der Beanbox generierte Adapterklasse

ActiveX

Das von der Firma *Microsoft* angebotene *ActiveX* [ActX 00] stellt weniger eine Bausteinarchitektur im herkömmlichen Sinne dar, sondern eine Sammlung unterschiedlicher Technologien, die in ihrer Gesamtheit als Bausteinarchitektur betrachtet werden können [Szyp 98]. Hierzu zu zählen ist beispielsweise das *Component Object Model (COM)* [COM 00], das die Infrastruktur für die transparente Kommunikation zwischen Objekten innerhalb eines Systems zur Verfügung stellt. Es definiert sowohl die Struktur der Objektschnittstellen als auch einige wenige Basisschnittstellen, die von jedem Objekt implementiert werden müssen, einen Verzeichnisdienst für die Lokalisierung aufzurufender Objekte sowie weitere Basisdienste für die Objektkommunikation. Das *Distributed Component Object Model (DCOM)* erweitert COM dahingehend, daß es die Kommunikation zwischen Objekten über Systemgrenzen hinweg unterstützt.

Der folgende Abschnitt soll einen kleinen Einblick in die bausteinorientierte Programmierung mit *ActiveX* geben und kurz die beteiligten Technologien darstellen.

- Bausteinbegriff

  Eine *ActiveX*-Komponente stellt einen wiederverwendbaren Software-Baustein dar, der in beliebigen Anwendungen eingesetzt werden kann. Ähnlich wie bei den *JavaBeans* kann eine *ActiveX*-Komponente beliebige Funktionalität implementieren, die von einfachen Oberflächenbausteinen (wie z.B. *Buttons*) bis zu vollständigen Anwendungen reichen kann. Es handelt sich um COM-Objekte, die um Schnittstellen zur Kommunikation mit dem ausführenden *Container* erweitert wurden. Ihre Erstellung ist nicht auf eine bestimmte Programmiersprache beschränkt.

  Grundsätzlich lassen sich drei Arten von *ActiveX*-Komponenten unterscheiden [Appl 98]:

  - *DLLs*
  - *EXE-Server*
  - *Controls*

  Dynamic Link Libraries (DLLs) werden innerhalb des aufrufenden *Threads* ausgeführt und stellen somit eine Möglichkeit der Wiederverwendung von *Code* dar, ohne dabei Einbußen bezüglich der Performanz in Kauf nehmen zu müssen. Im Gegensatz dazu laufen *EXE-Server* innerhalb eines eigenen *Threads*, was zur Verschlechterung der Performanz beim Aufruf dieser Objekte führt, dafür aber eine Parallelisierung der Abläufe gestattet. *Controls* hingegen sind spezielle Komponenten für die Erstellung von Benutzeroberflächen; sie laufen immer innerhalb des aufrufenden *Threads*.

- Schnittstellen

  Da jede *ActiveX*-Komponente gleichzeitig ein COM-Objekt ist, muß sie zumindest die Standardschnittstellen, die von jedem COM-Objekt implementiert werden müssen, zur Verfügung stellen. Darüber hinaus existieren einige optionale Schnittstellen, die der Kommunikation mit dem ausführenden *Container* dienen.

  Die eigentliche Funktionalität der Komponenten kann über ihre darüber hinaus verfügbaren öffentlichen Methoden aufgerufen werden. Änderungen des Zustandes werden über sogenannte *outgoing interfaces* bekannt gegeben. Hierbei handelt es sich um die Beschreibung einer Schnittstelle, die von der jeweiligen Komponente aufgerufen werden kann, sofern sie von einer anderen Komponente implementiert wird. Somit steht auch hier eine Möglichkeit zur Verfügung, ereignisbasierte Kommunikation zwischen den Komponenten zu realisieren.

  Die an den Schnittstellen angebotenen Methoden können mit Hilfe generischer COM-Mechanismen erfragt werden oder können in einer sogenannten *type library* explizit beschrieben werden.

- Customization

  Die Anpassung von *ActiveX*-Komponenten an die Gegebenheiten einer bestimmten Anwendung erfolgt über das Setzen von öffentlichen Attributen. Hierzu müssen wiederum Methoden zur Verfügung gestellt werden, die sowohl ein Auslesen der Attributwerte als auch deren Veränderung gestatten.

- Verknüpfung von Bausteinen

  Die Verknüpfung einzelner Bausteine zu einer lauffähigen Anwendung kann mit jeder beliebigen Programmiersprache, die COM unterstützt, erfolgen. Dem bei den *JavaBeans* bereits umgesetzten Gedanken der visuellen Programmierung, also der automatischen Generierung der Verknüpfung von Bausteinen durch die Entwicklungsumgebung, sehr nahe kommt z.B. die Verknüpfung mit Hilfe von *Visual Basic*.

  Visual Basic gestattet es, Komponenten auf einem sogenannten *Form* zu platzieren und die Verknüpfung der Komponenten durch die Ausprogrammierung sogenannter *Event*-Prozeduren zu realisieren. Dies bedeutet, daß die Entwicklungsumgebung dem Anwendungsentwickler die von einer Komponente angebotenen *Events* präsentiert, so daß dieser das gewünschte daraus auswählen kann. Daraufhin wird ein einfaches Gerüst einer Prozedur erzeugt, die bei Eintreten des *Events* aufgerufen wird. Diese ist nun vom Anwendungsentwickler geeignet zu programmieren und kann insbesondere den einfachen Aufruf einer Methode einer anderen Komponente enthalten.

  Das in Abbildung 2.4 dargestellte, sehr einfache Beispiel verdeutlicht dies nochmals. Es stellt eine Prozedur dar, die aufgerufen wird, wenn die Komponente mit dem Namen `Button1` angeklickt wird. Der Rumpf der Prozedur wurde automatisch mit Hilfe von *Visual Basic* erzeugt. Vom Anwendungsentwickler ist der Aufruf der Methode `Move` des Objektes `ObjectXY` eingefügt worden. Die Programmierung einer derartigen *Event*-Prozedur ist dabei nicht auf den einfachen Aufruf einer weiteren Komponenten beschränkt sondern kann beliebige Anweisungen und Aufrufe weiterer Komponenten enthalten.

```
Private Sub Button1_Click()

    ObjectXY.Move(100)

End Sub
```

Abbildung 2.4: Mit Hilfe von Visual Basic teilweise automatisch generierte Prozedur

2.1.1.2 Verknüpfung im Client

Die zweite Variante, wie die Verknüpfung von Bausteinen zu lauffähigen Anwendungen erfolgen kann, ist die Verknüpfung innerhalb der *Clients*. Verfügen die Bausteine über keine Schnittstelle, über die Änderungen des Bausteinzustandes propagiert werden können, so ist die oben beschriebene Verknüpfung mit Hilfe von Adaptern ausgeschlossen. Die Aufrufe der einzelnen Bausteine müssen stattdessen in einem *Client* programmiert werden, der auf diese Weise die Verknüpfung der Bausteine realisiert. Es ergibt sich also ein Ablauf, wie er in Abbildung 2.5 skizziert ist. Bei diesen Bausteinarchitekturen ist es somit auch nicht möglich, eine Automation der Instrumentierung durch Generierung von Meßpunkten in Adaptern zu erreichen.

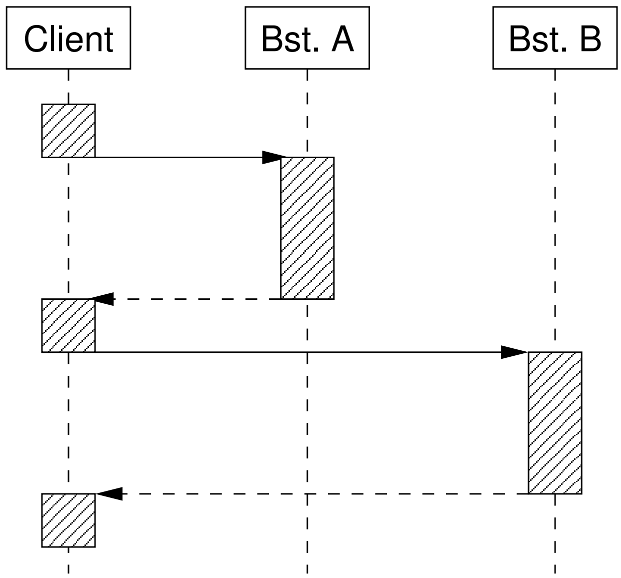

Abbildung 2.5: Verknüpfung zweier Bausteine im Client

Stattdessen kann hier nur eine Methodik für den Anwendungsentwickler vorgegeben werden, nach der er die erforderlichen Meßpunkte bei der Erstellung des *Clients* selbständig einzufügen hat. Die im folgenden kurz vorgestellten Architekturen *Enterprise JavaBeans (EJB)* und *CORBA Component Model (CCM)* verfügen allerdings über einen weiteren interessanten Ansatzpunkt für die Automatisierung: Bei diesen Architekturen rufen *Clients* die Methoden eines Bausteins nie direkt auf, sondern immer über einen ausführenden *Container*, der wiederum den tatsächlichen Aufruf durchführt (man spricht von sogenannter *Interception*). Dies ist erforderlich, um Dienste wie z.B. Transaktionskontrolle für den Entwickler transparent erbringen zu können. Es ergibt sich also ein Ablauf, wie er in Abbildung 2.6 dargestellt wird. Durch Instrumentierung des *Containers* wäre es auch hier denkbar, eine Automation der

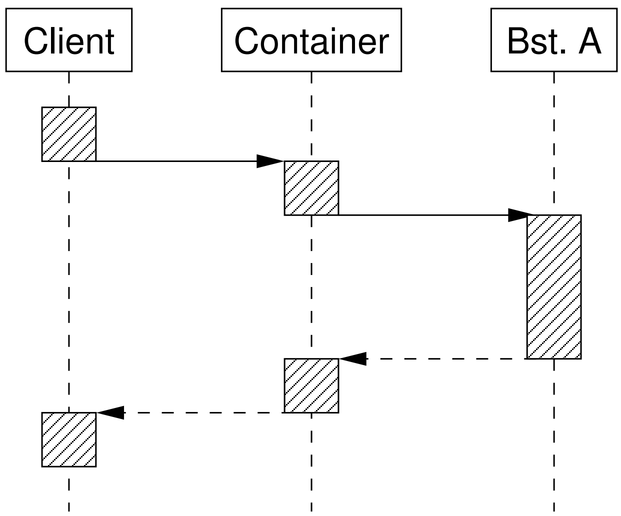

Abbildung 2.6: Interception durch den Container

Anwendungsüberwachung zu erreichen. Die vorliegende Arbeit beschränkt sich allerdings auf eine Automation durch Instrumentierung von generierten Adaptern.

Enterprise JavaBeans

Die *Enterprise JavaBeans (EJB)* Spezifikation [EJB 2.0] stammt – wie die bereits beschriebenen *JavaBeans* – ebenfalls von *Sun*. EJBs werden als logische Erweiterung der *JavaBeans* bezeichnet, auch wenn kein formaler Zusammenhang der beiden Spezifikationen besteht. Die Konzepte der *JavaBeans* können auch bei der Entwicklung von EJBs Verwendung finden, was aber nicht verpflichtend vorgeschrieben ist [Mons 00].

Die Idee der EJBs ist die Bereitstellung einer Bausteinarchitektur für Geschäftsanwendungen (*business applications*), die üblicherweise besondere Anforderungen z.B. im Bereich der Skalierbarkeit, der Transaktionssicherheit oder der persistenten Datenhaltung stellen. Ausgehend von der Beobachtung, daß unterschiedliche Varianten von Anwendungsservern (z.B. Transaktionsmonitore, Datenbanksysteme, etc.) am Markt existieren, auf denen die entsprechenden Geschäftsanwendungen ausgeführt werden, ist die Idee, eine generische Ablaufumgebung zur Verfügung zu stellen, die die wesentlichen Basisdienste an einer generischen Schnittstelle bereitstellt. Innerhalb dieser Ablaufumgebung können dann EJBs installiert und zu Geschäftsanwendungen verknüpft werden, die somit auf unterschiedlichen Anwendungsservern identisch ausgeführt werden können. Ein Anwendungsentwickler muß bei der Erstellung einer Geschäftsanwendung keine besonderen Kenntnisse z.B. über die Realisierung von Transaktionsüberwachung haben, sondern kann auf die vom Container angebotenen Dienste zurückgreifen.

- Bausteinbegriff

  Unter EJBs werden in mehreren Anwendungen verwendbare Softwarebausteine verstanden, die auf unterschiedlichen Anwendungsservern installiert und ausgeführt werden können. Im Gegensatz zu den *JavaBeans* stellen die EJBs ausschließlich relativ grobgranulare Geschäftsobjekte, wie z.B. einen kompletten Bestellvorgang, dar.

  Man kann drei grundsätzliche Arten von EJBs unterscheiden: *Session Beans*, *Entity Beans* und *Message-driven Beans*. Im Gegensatz zu den *Entity Beans* sind *Session Beans* in ihrer Lebensdauer an ihren *Client* gebunden und besitzen keinen persistenten Zustand. Die *Message-driven Beans* halten keinen Zustand und unterscheiden sich von den anderen Typen dadurch, daß sie mit Hilfe des *Java Messaging Service (JMS)* aufgerufen werden (siehe unten).

- Schnittstellen

  Session Beans und *Entity Beans* können mittels *Remote Method Invocation (RMI)* über ihr *Remote Interface* aufgerufen werden. An dieser Schnittstelle stellt eine EJB die von ihr angebotene Funktionalität zur Verfügung. Ein *Client* kann darüber hinaus über das *Home Interface* der EJB Instanzen des Bausteins erzeugen oder zerstören sowie eine Referenz auf das *Remote Interface* erfragen.

Wie bereits erwähnt, erfolgt der Aufruf einer *Message-driven Bean* im Gegensatz hierzu nicht über ihr *Remote Interface*, sondern über den JMS. Somit wird es möglich, asynchron aufrufbare EJBs zu implementieren.

Im Rahmen der EJB-Spezifikation ist keinerlei standardisierter Mechanismus vorgesehen, über den eine EJB Änderungen ihres Zustands ihrer Umwelt bekannt macht.

- Customization

  Die Anpassung einer EJB an spezielle Anwendungen kann über den *Deployment Descriptor* erfolgen. Hierbei handelt es sich um eine XML-basierte Datenstruktur, die zu jeder EJB Metainformationen enthält, die bei der Installation der EJB in einem konkreten *Container* ausgewertet werden. Beim Zusammenstellen mehrerer EJBs zu einem Anwendungspaket ist es dem Anwendungsersteller möglich, Eintragungen im *Deployment Descriptor* vorzunehmen und so die EJB anwendungsspezifisch zu konfigurieren. Dies erstreckt sich insbesondere auf Sicherheitseinstellungen und die Konfiguration der Transaktionsüberwachung.

- Verknüpfung der Bausteine

  Da im Gegensatz zu den *JavaBeans* bei den EJBs kein generischer Mechanismus vorgesehen ist, um Zustandsänderungen an die Umwelt zu propagieren, ist auch keine einfache Verknüpfung von EJBs möglich. Stattdessen erfolgt die Verknüpfung von Beans durch explizite Aufrufe des *Remote Interface* der jeweils aufzurufenden EJB. Die Verknüpfung gegebener EJBs zu einer Anwendung muß somit im *Client* durch den Anwendungsentwickler manuell programmiert werden. Somit sind zum heutigen Zeitpunkt auch keine Entwicklungsumgebungen, die eine einfache grafische Verknüpfung von EJBs zu einer Anwendung gestatten, verfügbar.

- Container

  Da es eines der wesentlichen Ziele bei der Entwicklung der EJBs war, die Entwicklung von Geschäftsanwendungen wesentlich zu erleichtern, werden sehr hohe Anforderungen an die *Container* gestellt, in denen die EJBs ausgeführt werden. Diese müssen zusätzlich zur *Java*-Ablaufumgebung eine große Anzahl an Basisdiensten, wie z.B. die Transaktionsüberwachung oder das Persistenzmanagement, anbieten.

  Aus diesem Grund erzeugt der *Container* Klassen, die die *Home* bzw. *Remote Interfaces* der EJBs repräsentieren und eingehende Aufrufe an die entsprechende EJB weiterleiten. Es ist also nicht möglich, EJBs direkt ohne Einbeziehung des *Containers* aufzurufen. Durch diese sogenannte *Interception* wird der *Container* somit z.B. in die Lage versetzt, die Transaktionsüberwachung für die jeweilige EJB transparent durchzuführen. Diese muß selbstverständlich geeignete Schnittstellen, wie z.B.

für das *Rollback* einer fehlgeschlagenen Transaktion, anbieten. Wie bereits erwähnt könnte diese Tatsache ausgenutzt werden, um durch Instrumentierung des *Containers* ebenfalls eine Automation der Managementinstrumentierung zu erreichen.

CORBA Component Model (CCM)

Das *CORBA Component Model (CCM)* stellt die wesentlichste Erweiterung der Version 3.0 der CORBA-Spezifikation dar. Der *Request for Proposals (RFP)* [OMG 97-06-12] entstand bereits 1997 und die hier vorgestellte jüngste Einreichung [OMG 99-07-01, OMG 99-07-02, OMG 99-07-03] stammt aus dem Jahr 1999. Das CCM ist bis heute allerdings noch nicht endgültig verabschiedet worden.

Es ist stark an der EJB-Spezifikation der Firma *Sun* orientiert und stellt gewissermaßen eine von einer konkreten Programmiersprache unabhängige Verallgemeinerung der EJBs dar. Es existiert eine standardisierte Abbildung, mit deren Hilfe EJBs als CCMs (oder umgekehrt) betrachtet werden können.

- Bausteinbegriff

  Das CCM beschränkt sich explizit auf *Server*-seitige Bausteine mit relativ grober Granularität. Diese können in beliebigen Programmiersprachen implementiert werden, solange sie über eine CORBA-konforme Schnittstellenbeschreibung verfügen.

  Ähnlich wie bei den EJBs werden unterschiedliche Arten von Bausteinen unterschieden: *Session*- und *Entity*-Bausteine, die ihrem jeweiligen Äquivalent in der EJB-Architektur entsprechen; darüber hinaus *Service*-Komponenten, die einen zustandslosen Dienst bereitsstellen sowie *Process*-Komponenten zur Modellierung von Abläufen.

- Schnittstellen

  Die Schnittstellen, über die ein CCM-Baustein angesprochen werden kann werden als sogenannte *Ports* bezeichnet. Es werden die folgenden vier *Ports* unterschieden:

  - Facets
  - Receptacles
  - Event Sources
  - Event Sinks

  Unter einer *Facet* wird eine Schnittstelle zum Zugriff auf die angebotene Funktionalität bezeichnet. Ein CCM-Baustein kann mehrere unterschiedliche *Facets* anbieten. *Receptacles* dienen ebenfalls der Verknüpfung mit anderen Bausteinen. Über ein *Receptacle* kann ein Baustein eine Referenz auf einen anderen Baustein erhalten, um Methoden dieses Bausteins auszuführen. Das Verschicken bzw. Empfangen von *Events* wird mit Hilfe der sogenannten *Event Sources* bzw. *Event Sinks* erreicht.

Darüber hinaus muß jeder CCM-Baustein über eine Vielzahl von Schnittstellen verfügen, die die Kommunikation mit dem ausführenden *Container* betreffen.

- Customization

  Ein CCM-Baustein macht seine Konfigurationsattribute über Methoden zugänglich. Somit ist eine Entwicklungsumgebung in der Lage, die Bausteine für die Verwendung in einer speziellen Umgebung bzw. in einer bestimmten Anwendung zu konfigurieren.

- Verknüpfung von Bausteinen

  Die Verknüpfung von CCM-Bausteinen zu Anwendungen erfolgt typischerweise wie bei den EJBs innerhalb der *Clients*. Darüber hinaus existiert die Möglichkeit der Verknüpfung mittels *Receptacles* bzw. *Events*, über die aufgrund der bislang fehlenden Implementierung allerdings keine Aussage getroffen werden kann.

- Container

  Genau wie bei den EJBs erfolgen Aufrufe eines CCM-Bausteins niemals direkt, sondern immer über den ausführenden *Container* (*Interception*). Auch hier wäre somit durch Instrumentierung des *Containers* eine Automation der Anwendungsüberwachung zu erreichen.

2.1.2 Begriffsbildung

Wie sich in den vorangegangenen Ausführungen gezeigt hat, unterscheiden sich die unterschiedlichen am Markt verfügbaren Bausteinarchitekturen zum Teil erheblich. Der folgende Abschnitt soll daher für den restlichen Verlauf der Arbeit eindeutige Definitionen treffen. Da die in dieser Arbeit vorgestellte Architektur die Existenz von expliziten und zumindest teilweise automatisch generierten Adaptern zur Verknüpfung der Bausteine voraussetzt, basieren die hier getroffenen Definitionen größtenteils auf den Spezifikationen der *JavaBeans* bzw. *ActiveX*.

Baustein
Unter einem Baustein soll im Kontext dieser Arbeit folgendes verstanden werden:

- Eine Software-Komponente,
- die über eindeutig definierte Schnittstellen mit ihrer Umgebung kommuniziert,
- deren Funktionalität genau bekannt ist,
- die mit anderen Bausteinen zu einer Anwendung verknüpft werden kann,
- die bei der Erstellung beliebiger Anwendungen verwendet werden kann, also wiederverwendbar ist, und

- die konfigurierbar ist, um den speziellen Anforderungen konkreter Anwendungen zu genügen.

- Die Implementierung sei für den Anwendungsersteller dabei transparent; der Baustein stellt sich für ihn als Blackbox dar.

Unterschiedliche Bausteine lassen sich weiterhin folgendermaßen unterscheiden:

- Oberflächen-/Serverbausteine

  Im Gegensatz zu *Server*-Bausteinen verfügen **Oberflächenbausteine** über eine grafische Repräsentation zur Laufzeit und ermöglichen somit Benutzerinteraktionen (siehe unten). Sie finden also bei der Realisierung des Dienstzugangspunktes Verwendung. **Server-Bausteine** können von Benutzern nicht direkt angesprochen werden, sondern werden bei der Bearbeitung von Benutzertransaktionen (siehe Abschnitt 2.2.2) mit Hilfe der Anwendungslogik aufgerufen.

  Abhängig davon, ob ein Oberflächenbaustein eine Benutzereingabe (eine Benutzerinteraktion vom Benutzer zum System) oder die Präsentation eines Ergebnisses (eine Benutzerinteraktion vom System zum Benutzer) ermöglicht, lassen sich diese weiterhin unterteilen in

  - Eingabebausteine und
  - Präsentationsbausteine

- Aktive/passive Bausteine

  Passive Bausteine verfügen über keinen eigenen Kontrollfluß (siehe unten), sondern werden immer im aufrufenden Kontrollfluß ausgeführt. Im Gegensatz dazu verfügen **aktive Bausteine** über eigene Kontrollflüsse. Aktive Bausteine stellen an ihrer Schnittstelle die Möglichkeit zur Verfügung, Aufträge in eine Warteschlange einzufügen. Diese werden dann asynchron in einem Kontrollfluß des aktiven Bausteins bearbeitet.

Bausteinarchitektur
Unter einer Bausteinarchitektur soll die Menge an Vorschriften verstanden werden, die angibt, wie Bausteine beschaffen sein müssen, um von einer Entwicklungsumgebung verwendet werden zu können und um mit anderen Bausteinen zu einer Anwendung zusammengesetzt werden zu können. Die Bausteinarchitektur schreibt auch vor, wie die Kommunikation zwischen Bausteinen zu erfolgen hat und wie die Anpassung einzelner Bausteine an spezielle Anwendungen erfolgt.

Anwendungslogik

Die Anwendungslogik einer Anwendung beschreibt die konkrete Verknüpfung von Bausteinen zu einer Anwendung. Sie schreibt also die Aufrufreihenfolge der einzelnen Bausteine sowie die Parameterübergabe innerhalb der Anwendung vor.

Im Verlauf dieser Arbeit soll als Anwendungslogik also nur der die einzelnen Bausteine verknüpfende *Code* bezeichnet werden (im englischen häufig als *glue code* bezeichnet) und nicht der *Code* der Bausteine selbst. Die Anwendungslogik kann beispielsweise durch Adapter explizit realisiert sein, kann aber auch innerhalb des *Clients* liegen oder implizit durch direkte Verknüpfung der Bausteine existieren.

Für den weiteren Verlauf der Arbeit wird eine explizite Anwendungslogik in Form von die einzelnen Bausteinen verknüpfenden Adaptern vorausgesetzt. Weiterhin wird vorausgesetzt, daß die Anwendungslogik nicht vollständig vom Anwendungsentwickler manuell implementiert werden muß, sondern daß geeignete Entwicklungswerkzeuge vorliegen, die eine (zumindest teilweise) automatische Generierung der Anwendungslogik gestatten. Dies setzt insbesondere voraus, daß nicht nur die Aufrufschnittstellen eines Bausteins spezifiziert sind, sondern daß auch eine definierte Schnittstelle vorliegt, über die Zustandsänderungen der Bausteine der Umwelt übermittelt werden können.

Entwicklungsumgebung

Eine Entwicklungsumgebung ist ein Werkzeug, mit dessen Hilfe ein Anwendungsentwickler vorgefertigte Bausteine zu einer Anwendung zusammenfügen kann, also die Anwendungslogik erstellen kann. Der Anwendungsentwickler definiert die einzufügenden Verknüpfungen beispielsweise über eine grafische Oberfläche. Die Entwicklungsumgebung generiert daraufhin die erforderlichen Adapter, die vom Entwickler evtl. noch manuell erweitert werden müssen. Die einzelnen Bausteine lassen sich im Rahmen der *Customization* an die Anforderungen der jeweiligen Anwendung anpassen.

Customization

Unter der *Customization* eines Bausteins wird die Anpassung des Bausteins an die speziellen Gegebenheiten der zu erstellenden Anwendung verstanden. Vom Bausteinentwickler werden Freiheitsgrade vorgesehen, die erst zum Zeitpunkt der Anwendungserstellung vom Anwendungsentwickler endgültig festzulegen sind.

Kontrollfluß

Unter einem Kontrollfluß soll ein *Thread* innerhalb eines Systems verstanden werden. Innerhalb eines Kontrollflusses werden Anweisungen eines Programmes sequentiell ausgeführt. Es existieren Systemmechanismen, mit deren Hilfe Kontrollflüsse weitere Kontrollflüsse erzeugen und starten können. Somit können mehrere Kontrollflüsse zum selben

Zeitpunkt aktiviert sein, wenngleich zu jedem Zeitpunkt auf jedem Prozessor nur ein Kontrollfluß tatsächlich ausgeführt werden kann.

Da die Sequenzdiagramme der *Unified Modeling Language (UML)* [OMG 00-03-01] keine Möglichkeit vorsehen, darzustellen, in welchem Kontrollfluß ein bestimmtes Objekt zu einem bestimmten Zeitpunkt ausgeführt wird, wird für den weiteren Verlauf der Arbeit folgende Konvention getroffen: Der ein Objekt zu einem bestimmten Zeitpunkt ausführende Kontrollfluß wird durch Überlagerung der Lebenslinie des Objektes mit einem schraffierten Rechteck dargestellt (siehe Abbildung 2.7). Identisch schraffierte Rechtecke beziehen sich dabei auf den selben Kontrollfluß, während unterschiedlich schraffierte Rechtecke unterschiedliche Kontrollflüsse veranschaulichen. Darüber hinaus werden die üblichen Konventionen der UML für Sequenzdiagramme angewandt.

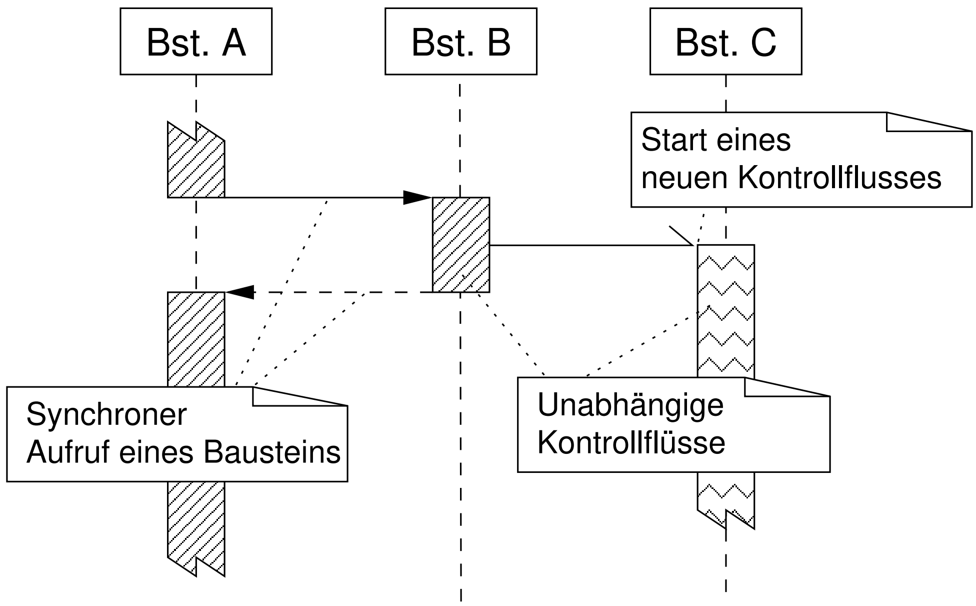

Abbildung 2.7: Darstellung unterschiedlicher Kontrollflüsse in Sequenzdiagrammen

Im Unterschied zum (in der Standard UML Notation vorgesehenen) Steuerungsfokus (*Focus of Control*) wird es somit möglich, innerhalb eines Sequenzdiagramms mehrere unabhängige Kontrollflüsse darzustellen. Durch diese Art der Modellierung läßt sich darüber hinaus ausdrücken, wie beim synchronen Aufruf eines Bausteins durch einen anderen der Kontrollfluß vom aufrufenden Baustein auf den aufgerufenen übergeht. Weiterhin ist in der Abbildung der Start eines neuen Kontrollflusses durch einen asynchronen Aufruf aus einem existierenden Kontrollfluß heraus zu erkennen.

2.2 Anwendungsmanagement

Das Anwendungsmanagement umfaßt einen breiten Aufgabenbereich. Die vorliegende Arbeit beschränkt sich auf einen Ausschnitt des Anwendungsmanagements, die Anwendungsüberwachung. Zur genauen Abgrenzung des Begriffs Anwendungsüberwachung werden im folgenden die vielfältigen Aufgaben des Anwendungsmanagements dargestellt. Um dies zu erreichen, wird einerseits anhand des Lebenszyklus der zu managenden Anwendung, andererseits anhand einer funktionalen Einteilung der Managementaufgaben vorgegangen. Daraufhin wird der – für die vorliegende Arbeit besonders wichtige – Bereich der Transaktionsüberwachung eingeführt. Gerade hier haben sich noch keine allgemein akzeptierten Begriffe ergeben, so daß eine genaue Definition der Bedeutung einzelner Begriffe unerläßlich ist.

Klassifikation nach Lebenszyklus

Zur Beschreibung des Lebenszyklus einer Anwendung existieren verschiedene Ansätze, die sich aber nicht wesentlich voneinander unterscheiden. Laut [HAN 99a] besteht der Lebenszyklus einer Anwendung z.B. aus den folgenden Phasen:

- Vorbereitung der Anwendung für die Verteilung und Installation
- Prüfung der Ausstattung und Konfiguration der Zielsysteme
- Verteilung der Software-Pakete
- Installation und Konfiguration der Software
- Überwachung der Anwendung im Betrieb
- Deinstallation der Anwendung

Ziel der vorliegenden Arbeit ist es, eine Architektur anzugeben, mit deren Hilfe Managementlösungen für die Überwachung einer Anwendung im laufenden Betrieb erzeugt werden können. Die Fokussierung auf diesen Bereich begründet sich zum einen mit der Tatsache, daß existierende Lösungen in diesem Bereich noch erhebliche Defizite aufweisen (vgl. Kapitel 4), und zum anderen damit, daß eine Ausnutzung der Bausteinstruktur einer Anwendung für das Management in dieser Phase großen Nutzen verspricht.

Klassifikation nach Funktionsbereichen

Auch zur Klassifikation nach Funktionsbereichen existiert eine Vielzahl verschiedener Ansätze. Insbesondere im Bereich des OSI-Managements weit verbreitet ist die Unterteilung in die fünf Managementfunktionsbereiche [ISO 7498-4]

- Fehlermanagement
- Konfigurationsmanagement
- Abrechnungsmanagement
- Leistungsmanagement
- Sicherheitsmanagement

Im System- und Anwendungsmanagement ist diese Unterteilung laut [HAN 99a] allerdings weniger gebräuchlich, da sie die tatsächliche Zuordnung von Aufgaben zu Rollen und Personen nicht widerspiegelt. Günstiger ist hier z.B. eine Einteilung in

- Überwachung (*Monitoring*)
- Software-Verteilung und -Installation (*SW-Distribution, SW-Deployment*)
- Bestandsführung (*Inventory Management*)
- Benutzeradministration (*User Administration*)

Der Phase des Betriebs ist hier insbesondere die Überwachung zuzuordnen, die Benutzeradministration findet natürlich während dieser Phase ebenfalls statt. Nachdem sich im Rahmen der Benutzeradministration keine Vorteile aus einer Ausnutzung der Bausteinstruktur der zu managenden Anwendung erwarten lassen, wird sich diese Arbeit also insbesondere auf die Anwendungsüberwachung während der Phase des Betriebs beschränken.

2.2.1 Anwendungsüberwachung

Wie bereits dargestellt, findet die Anwendungsüberwachung hauptsächlich in der Phase des Betriebs statt. Die hier anfallenden Tätigkeiten umfassen im wesentlichen die OSI-Funktionsbereiche des Fehler- und Leistungsmanagements.

Die Anwendungsüberwachung ist in der Lage, unmittelbare Aussagen über die Anwendung selbst zu liefern. Die hierzu erforderlichen Informationen werden den entsprechenden Managementsystemen direkt von der Anwendung zur Verfügung gestellt. Dies kann nicht mit Hilfe der herkömmlichen Techniken des *Server* und *Desktop* Managements erreicht werden, sondern erfordert eine Managementinstrumentierung der Anwendung.

Managementinstrumentierung

Unter der Managementinstrumentierung einer Anwendung wird das Einbringen von speziellem Management-*Code* in den *Source-Code* der zu überwachenden Anwendung verstanden. Somit wird es möglich, Informationen, die nur innerhalb der Anwendung gemessen werden können, an Managementsysteme zu kommunizieren.

Server Management
Aufgabe des Server Managements ist die Überwachung der Serversysteme, auf denen die Anwendungen zum Ablauf kommen. Häufig stellt eine Überlastung oder ein Ausfall eines derartigen Systems den Grund für eine Beeinträchtigung des von einer Anwendung angebotenen Dienstes dar. Daher ist das Server Management ein wichtiger Teil im Rahmen der Gesamtaufgabe des Managements der Unternehmensanwendungen. Nachdem es aber nur mittelbare Aussagen über den Zustand der Anwendungen durch Überwachung der darunterliegenden Serversysteme gestattet, soll es im Rahmen dieser Arbeit nur am Rande betrachtet werden.

Desktop Management
Das Desktop Management, also die Administration der Arbeitsplatzsysteme, befaßt sich im wesentlichen mit Aufgaben des Inventory Management, der Software-Verteilung und Fehlerbehebung mittels Fernsteuerung. Es liegt somit ebenfalls außerhalb des eigentlichen Fokus der vorliegenden Arbeit.

2.2.2 Transaktionsüberwachung

Ein wesentlicher Aspekt der Anwendungsüberwachung ist die Transaktionsüberwachung, also die Überwachung der vom Benutzer wahrnehmbaren Transaktionen. Diese Benutzertransaktionen werden durch Benutzerinteraktionen angestoßen und beendet und können in Subtransaktionen unterteilt werden. Wesentliche Parameter, die es bezüglich der Benutzertransaktionen zu überwachen gilt, sind die Transaktionsdauer sowie die Antwortzeit. Diese Begriffe werden im folgenden Abschnitt definiert und ihre Zusammenhänge in Abbildung 2.9 nochmals verdeutlicht.

Benutzertransaktion
Unter einer Benutzertransaktion (BTA) wird eine an der Benutzerschnittstelle erbrachte, in sich abgeschlossene und vom Dienstnutzer als solche wahrnehmbare Aufgabe verstanden. Aus Abbildung 2.8 läßt sich der allgemeine Ablauf einer BTA erkennen: Eine Benutzerinteraktion (BI) liefert den Anstoß für die BTA, der Benutzer gibt also ein Kommando (z.B. drückt einen *Button* der Oberfläche oder betätigt die `RETURN`-Taste) und löst somit die BTA aus. Dann erfolgt eine beliebige, von der jeweiligen BTA abhängige Aktivität (z.B. Berechnungen, Informationsrecherche). Das Ende stellt typischerweise wiederum eine BI dar, mit der dem Benutzer das Ergebnis der BTA präsentiert wird. Ebenso vorstellbar sind BTAs, die ohne explizite Präsentation eines Ergebnisses beendet werden. Die BTA endet dann einfach mit dem Ende der ausgeführten Aktivität.

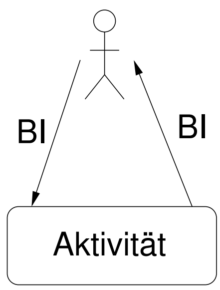

Abbildung 2.8: Allgemeine Darstellung einer Benutzertransaktion

Benutzerinteraktion
Eine Benutzerinteraktion (BI) ist eine Kommunikation zwischen Benutzer und Anwendung. Diese kann beispielsweise durch das Drücken einer Taste oder das Anklicken eines *Buttons* erfolgen. Andererseits wird auch die Kommunikation von der Anwendung zum Benutzer hin, also z.B. die Präsentation eines Ergebnisses, als BI bezeichnet. BIs erfolgen immer an der Dienstzugangsschnittstelle zwischen Benutzer und System. Sie finden also immer an der Schnittstelle zwischen (menschlichem) Benutzer und einem Oberflächenbaustein und nicht an der Schnittstelle zwischen zwei Bausteinen innerhalb der Anwendungslogik statt.

Subtransaktion
Unter einer Subtransaktion wird eine in sich abgeschlossene Teilaufgabe einer BTA verstanden, die vom Benutzer nicht unbedingt als solche wahrgenommen werden kann. Subtransaktionen werden somit nicht direkt durch BIs begonnen und beendet, sondern werden mittelbar im Laufe der Bearbeitung einer BTA angestoßen.

Transaktionsdauer
Unter der Transaktionsdauer soll im folgenden die Zeit verstanden werden, die vom Anstoß einer BTA (durch eine BI) bis zum vollständigen Ende der daraus resultierenden Aktivität vergeht (vgl. Abbildung 2.9). Diese kann länger als die vom Benutzer wahrgenommene Antwortzeit sein, da z.B. noch Aufräumarbeiten oder Einträge in einer Log-Datei vorzunehmen sind.

Antwortzeit
Die Antwortzeit ist die Zeit, die zwischen dem Anstoß einer BTA (durch eine BI) und dem Ende der BTA (wiederum durch eine BI) vergeht (vgl. Abbildung 2.9). Es handelt sich also um die vom Benutzer wahrnehmbare Dauer der Transaktion. Diese muß aus oben genannten Gründen von der eigentlichen Transaktionsdauer unterschieden werden. Handelt es sich um eine BTA ohne explizite Präsentation eines Ergebnisses, kann die Zeit bis zur Rückkehr des Kontrollflusses zum Aufrufer als Antwortzeit betrachtet werden. Die Antwortzeit wird in der englischsprachigen Literatur häufig als *Application Response Time* bezeichnet.

2.3 Dienstorientierung und -management

Die vorliegende Arbeit geht von einer stark dienstorientierten Sicht der Anwendungsüberwachung aus. Ziel der vorgestellten Architektur soll es sein, die Überwachung so zu gestalten, daß insbesondere für den Dienstnutzer relevante Informationen gemessen werden

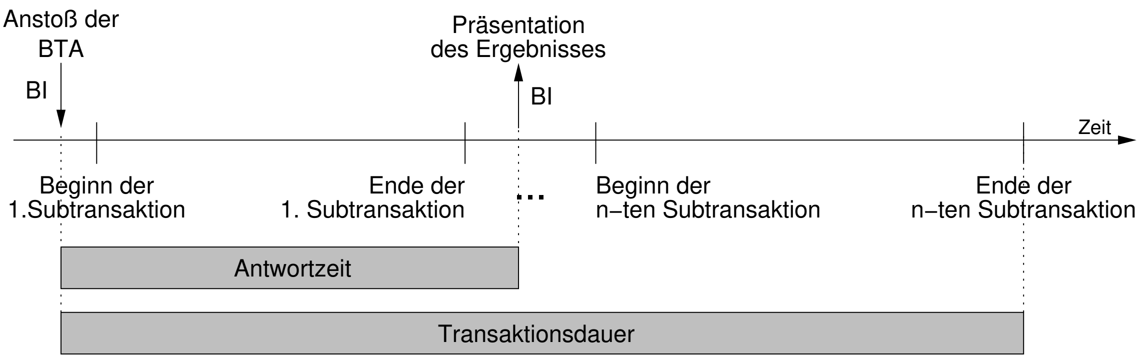

Abbildung 2.9: Darstellung der Zusammenhänge im Bereich der Transaktionsüberwachung

können. Obwohl in den Bereichen der Dienstorientierung und des Dienstmanagements in den letzten Jahren erhebliche Forschungsaktivitäten zu beobachten waren, hat sich immer noch keine allgemein anerkannte und umfassende Definition der verwendeten Begriffe herauskristallisieren können. Die folgenden Definitionen legen das Verständnis eines Dienstes sowie der damit verbundenen Entitäten für den weiteren Verlauf der Arbeit fest. Abbildung 2.10 stellt die Zusammenhänge der beschriebenen Elemente grafisch dar. Für eine detailliertere Betrachtung der Modellierung von Diensten sei auf [SMTF 01] erwiesen.

- Dienst
  Unter einem Dienst soll im folgenden Funktionalität verstanden werden, die einem Dienstnehmer an einer Schnittstelle mit gewissen Dienstgütemerkmalen von einem Diensterbringer zur Verfügung gestellt wird. Die angebotene Funktionalität umfaßt dabei sowohl Nutzungs- als auch Managementfunktionalität.
- Rollen
  - Diensterbringer
    Der Diensterbringer ist dafür verantwortlich, den Dienst, wie in der Dienstvereinbarung vereinbart, dem Dienstnehmer zur Verfügung zu stellen. Er betreibt hierzu eine Dienstimplementierung sowie das Dienstmanagement.
  - Dienstnehmer
    Auf Dienstnehmerseite lassen sich zwei unterschiedliche Rollen unterscheiden, die für die folgenden Betrachtungen von Bedeutung sind, der Dienstnutzer und der Dienstkunde.
    * Dienstnutzer
      Der Dienstnutzer nutzt die Nutzungsfunktionalität des Dienstes über den Dienstzugangspunkt.
    * Dienstkunde
      Der Dienstkunde beauftragt den Dienst, schließt eine Dienstvereinbarung mit

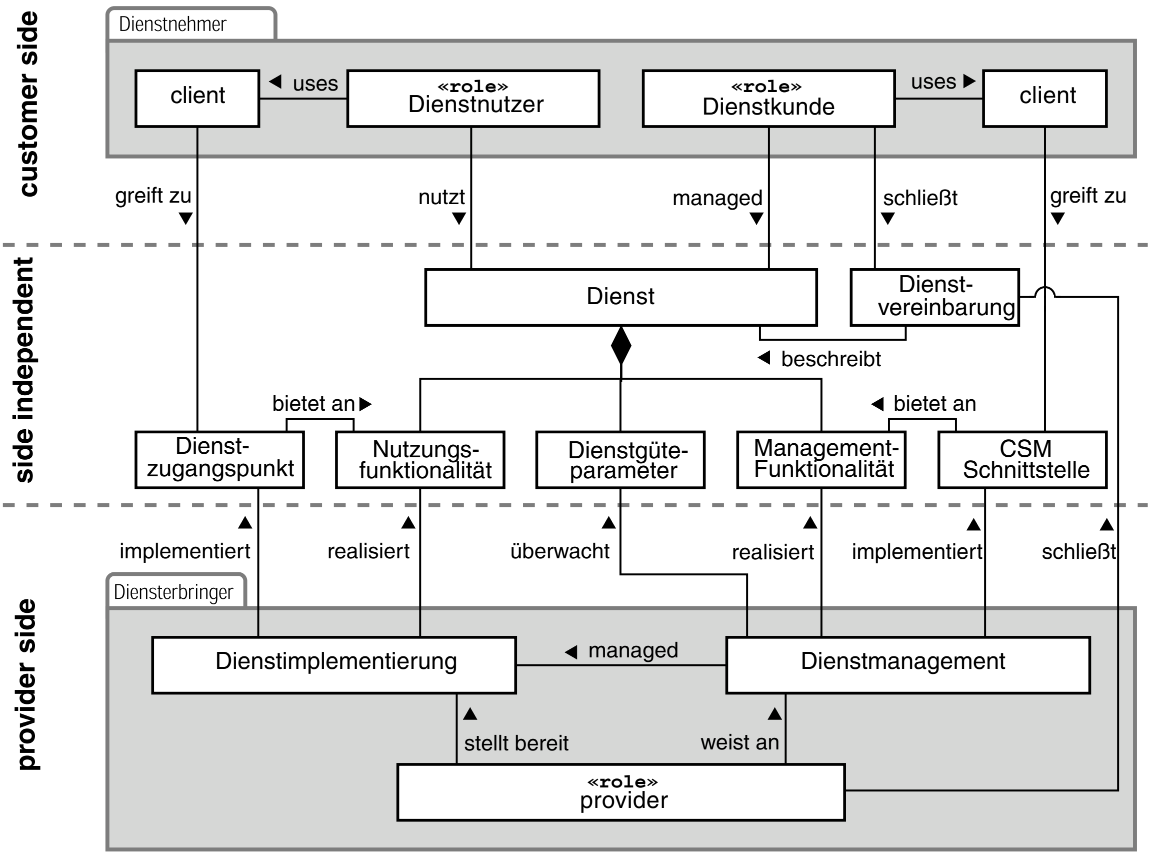

Abbildung 2.10: Dienstmodell nach [SMTF 01]

dem Diensterbringer und überwacht die Diensterbringung. Für die Überwachung des Dienstes nutzt er die Managementfunktionalität des Dienstes.

- Funktionalität
  - Nutzungsfunktionalität
    Die Nutzungsfunktionalität umfaßt alle Transaktionen, die vom Dienstnutzer benötigt werden. Sie stellt also den eigentlichen Zweck des Dienstes dar.
  - Managementfunktionalität
    Transaktionen, die nicht dem eigentlichen Zweck des Dienstes dienen, aber dennoch erforderlich sind, um z.B. den Dienst zu konfigurieren oder zu überwachen, werden unter dem Begriff der Managementfunktionalität zusammengefaßt.
- Schnittstellen
  Ebenso wie die Funktionalität aufgeteilt wurde, existieren zwei Schnittstellen, an denen diese Funktionalität dem Dienstnehmer zur Verfügung steht: der Dienstzugangspunkt und die *Customer Service Management (CSM)* Schnittstelle.

  - Dienstzugangspunkt
    Am Dienstzugangspunkt steht dem Dienstnutzer die Nutzungsfunktionalität zur Verfügung.
  - *Customer Service Management* Schnittstelle
    An der CSM Schnittstelle steht dem Dienstkunden die Managementfunktionalität zur Verfügung.

- Dienstgüteparameter
  Beide Arten von Funktionalität müssen bestimmten Dienstgüteparametern genügen. Diese Parameter definieren die Dienstgüte, die minimal erforderlich ist, um dem Dienstnutzer einen nützlichen Dienst zur Verfügung zu stellen.

- Dienstvereinbarung
  Die zwischen Diensterbringer und Dienstkunden geschlossene Dienstvereinbarung beschreibt den Dienst mit seiner Funktionalität und den zugehörigen Dienstgüteparametern. Darüber hinaus werden dort die Schnittstellen festgelegt, die dem Dienstnehmer Zugriff auf die jeweilige Funktionalität gestatten.

- *Client*
  Der Dienstnehmer benötigt *Clients*, um über die Schnittstellen des Dienstes auf die Funktionalität zugreifen zu können.

- Dienstimplementierung
  Der Diensterbringer betreibt eine Dienstimplementierung, um die Funktionalität des Dienstes zur Verfügung zu stellen. Die Dienstimplementierung implementiert auch die Schnittstelle für den Zugriff auf die Funktionalität. Unter der Dienstimplementierung verstehen wir die Kombination des gesamten Wissens, des Personals sowie der Hard- und Software, die zur Realisierung des gewünschten Dienstes erforderlich ist.

- Dienstmanagement
  Die Hauptaufgabe des Dienstmanagements ist es, dafür Sorge zu tragen, daß der Dienst wie in der Vereinbarung festgelegt dem Dienstnehmer zur Verfügung gestellt wird. Das bedeutet unter anderem, die Einhaltung der Dienstgüteparameter zu überwachen und sicherzustellen. Hierzu wird die Dienstimplementierung überwacht und konfiguriert. Außerdem implementiert das Dienstmanagement die CSM Schnittstelle, über die dem Dienstkunden eingeschränkte Managementfunktionalität des Dienstes zur Verfügung steht.

Kapitel 3

Anforderungen an die Überwachung bausteinbasierter Anwendungen

In diesem Kapitel werden die Anforderungen, die an eine Managementlösung für die Überwachung bausteinbasierter Anwendungen gestellt werden müssen, herausgearbeitet. Grundsätzlich müssen an die Überwachung bausteinbasierter Anwendungen natürlich mindestens dieselben Anforderungen gestellt werden wie man sie an die Überwachung jeder beliebigen Anwendung stellen würde. Aus diesem Grund werden in Abschnitt 3.1 zunächst die Anforderungen dargestellt, die eine entsprechende Managementlösung mindestens erbringen sollte. Aus den Besonderheiten der Bausteinorientierung ergeben sich allerdings eine Vielzahl weiterer Anforderungen, die über die Leistungsfähigkeit derzeitiger Managementsysteme hinausgehen (vgl. Kapitel 4). Diese Anforderungen werden ausführlich in Abschnitt 3.2 untersucht.

3.1 Allgemeine Anforderungen an die Überwachung von Anwendungsdiensten

Im folgenden wird das in Abschnitt 2.3 vorgestellte allgemeine Dienstmodell dahingehend erweitert, daß es die spezifischen Gegebenheiten bei der Erbringung von Anwendungsdiensten beschreibt. Hieraus lassen sich dann Anforderungen herleiten, die an eine Managementlösung in diesem Umfeld zu stellen sind.

3.1.1 Modellierung von Anwendungsdiensten

Handelt es sich bei der Dienstimplementierung eines gegebenen Dienstes um eine Anwendung, so läßt sich von einem Anwendungsdienst sprechen. Der Diensterbringer wird dann als *Application Service Provider (ASP)* bezeichnet. Das Management eines Anwendungsdienstes stützt sich auf das Anwendungsmanagement.

Die Benutzeroberfläche, über die ein Dienstnutzer typischerweise auf den Anwendungsdienst zugreift, kann entweder Teil des *Clients* sein, also im Verantwortungsbereich des Dienstnehmers liegen oder aber einen Teil des Anwendungsdienstes darstellen. Als *Client* benötigt der Dienstnutzer dann nur die entsprechenden Rechensysteme sowie unter Umständen bestimmte Software wie z.B. einen Webbrowser. Dem entsprechend kann den Dienstzugangspunkt somit entweder die Aufrufschnittstelle zwischen *Client* und eigentlicher Anwendung oder aber die Oberfläche selbst darstellen.

Da die Dienstimplementierung für den Dienstnehmer mit Ausnahme des Dienstzugangspunktes vollständig transparent ist, ergeben sich darüber hinaus keine weiteren Unterschiede zum allgemeinen Dienstmodell. Abbildung 3.1 stellt die beschriebene Situation noch einmal grafisch dar.

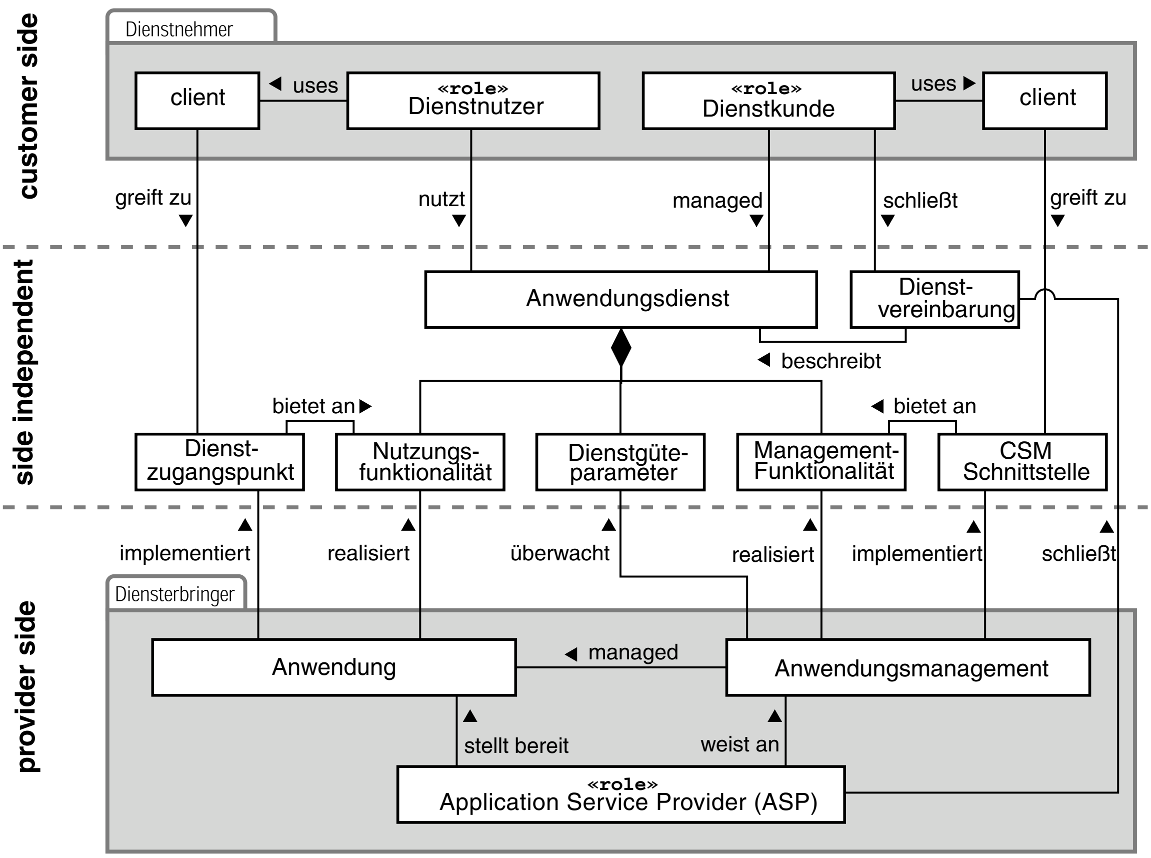

Abbildung 3.1: Modellierung von Anwendungsdiensten

3.1.2 Ermittlung der Anforderungen

Aus dem oben beschriebenen Szenario und insbesondere aus den darin zu identifizierenden Rollen lassen sich die Anforderungen, die an eine Managementlösung für die Überwachung von Anwendungsdiensten gestellt werden müssen, herleiten. Diese werden im folgenden detailliert beschrieben:

- Funktionale Anforderungen:
  - Messung von dienst- bzw. nutzerorientierten Parametern

    Zur Überwachung der in der Dienstvereinbarung festgelegten Schwellwerte ist es erforderlich, daß entsprechende dienst- bzw. nutzerorientierte Parameter gemessen werden können. Hierbei handelt es sich typischerweise um Parameter, die sich auf vom Benutzer angestoßene und wahrnehmbare Transaktionen des Dienstes beziehen. Nach [Wald 01] sind für einen Nutzer bei der Nutzung eines Anwendungsdienstes zwei Größen von herausragender Bedeutung, die Verfügbarkeit (*availability*) und das Antwortzeitverhalten (*responsiveness*). Sowohl die Anzahl erfolgreicher bzw. nicht erfolgreicher Transaktionen als auch die Zeitdauer für die Erbringung einer Transaktion müssen somit gemessen werden können. Die zu überwachenden Transaktionen ergeben sich aus der Dienstvereinbarung. Ebenso wird häufig ein Wert für die Mindestverfügbarkeit eines Dienstes festgelegt und muß somit überwacht werden können. Eine ausführliche Untersuchung über die Erstellung von Dienstvereinbarungen und die Festlegung der zu überwachenden Parameter findet sich in [Schm 00]
  - Messung der tatsächlichen Dienstnutzung

    Die Entscheidung, ob ein Dienst ordnungsgemäß funktioniert oder nicht, läßt sich nur durch Messung der tatsächlichen Dienstnutzung treffen. Dies bedeutet, daß die jeweils vom Benutzer tatsächlich wahrgenommene Dienstqualität gemessen und als Basis für weitere Entscheidungen herangezogen werden muß. Beschränkt man sich stattdessen z.B. auf stichprobenartige Überprüfungen der Dienstqualität, so kann man keine sichere Aussage darüber treffen, ob die vereinbarten Parameter bei einer konkreten Nutzung eingehalten wurden oder nicht.
  - Detaillierte Messung des Anwendungszustands

    Für den Dienstnehmer ist die oben beschriebene Identifikation von Leistungsabnahmen oder Ausfällen des abonnierten Dienstes bereits ausreichend. Aufgabe des Diensterbringers ist es aber nicht nur, derartige Probleme festzustellen, sondern für die Einhaltung der vereinbarten Parameter Sorge zu tragen. Dies bedeutet, daß er detailliertere Informationen über den Zustand der Anwendung sowie der darunterliegenden Systeme benötigt, die es ihm bei Auftreten eines Fehlers gestatten, die Ursache des Problems zu ermitteln und zu beheben.

- Korrelation von Detailinformationen

  Sowohl um die Fehlersuche zu erleichtern als auch, um bei Auftreten eines Problems die betroffenen Dienste und Nutzer ermitteln zu können, ist es für den Diensterbringer erforderlich, eine Korrelation der Teilinformationen vornehmen zu können. Beispielsweise sollte es bei einer fehlerhaften Transaktion möglich sein, die Teiltransaktion zu ermitteln, die für die Fehlfunktion verantwortlich war, um somit die mögliche Ursache des Problems schnell eingrenzen zu können. Daraus ergibt sich die Anforderung, daß die Informationen über die einzelnen Komponenten aus verläßlicher Quelle stammen und in einer vergleichbaren Form vorliegen müssen, um eine automatische Bewertung und Korrelation zu ermöglichen. Außerdem sollte der Diensterbringer bei Identifikation eines Problems in der Lage sein, die davon betroffenen Dienste und Benutzer ermitteln zu können, um somit die wirtschaftlichen Auswirkungen des aufgetretenen Problems abschätzen und geeignet reagieren zu können.

- Messung des Ressourcenverbrauchs von Anwendungen

  Sowohl aus Gründen der Abrechnung als auch aus Gründen der Kapazitätsplanung und Fehlersuche ist es wünschenswert, den Ressourcenverbrauch einer Anwendung bestimmen zu können. Es ist also erforderlich, Systemparameter wie z.B. Verbrauch von Prozessorzeit oder Speicherbelegung (vgl. Abschnitt 4.1.1.2) zu ermitteln und dem jeweiligen Anwendungsdienst zuordnen zu können. Ideal wäre hierbei eine detaillierte Aufschlüsselung des Ressourcenverbrauchs nach Dienst, Transaktion und Nutzer.

• Nicht-funktionale Anforderungen:

- Geringer Aufwand

  Eine Managementlösung wird nur dann verbreitet eingesetzt werden, wenn der Aufwand für alle beteiligten Rollen so gering wie möglich ist. Dies bedeutet, daß zum einen der Diensterbringer durch weitestgehende Automatisierung des Managements entlastet werden muß. Darüber hinaus ist es aber ebenfalls von großer Bedeutung, dem Anwendungsentwickler bei der Bereitstellung der gewünschten Managementfunktionalität der Anwendung zu unterstützen. Insbesondere durch die Verwendung einfacher Konzepte und Verfahren, die möglichst dieselben Methoden verwenden, die bei der Erstellung der eigentlichen Anwendung ohnehin zum Einsatz kommen, ist damit zu rechnen, daß Anwendungsentwickler die Anwendungen mit entsprechender Managementfunktionalität versehen werden.

- Geringe Auswirkung auf den überwachten Dienst

  Jede Messung hat selber wieder Auswirkungen auf das gemessene System. Ziel jeder Managementlösung muß es sein, diese Auswirkungen so gering wie möglich zu halten.

– Zeitnahe Überwachung

  Leistungsabnahmen oder Ausfälle eines Dienstes müssen umgehend erkannt werden, so daß eine Behebung des zugrundeliegenden Problems möglichst noch stattfinden kann, bevor eine Verletzung der Dienstvereinbarung vorliegt (proaktives Management). Daher unterliegt die Messung und Auswertung der Information zwar keinen harten Echtzeitbedingungen, muß aber dennoch zeitnah erfolgen, um schnelle Reaktionen zu gestatten.

– Generische Verwendbarkeit

  Die Lösung sollte für beliebige (oder zumindest für eine große Klasse von) Anwendungen einsetzbar sein und nicht auf eine bestimmte Infrastruktur (z.B. Betriebssystem) oder eine bestimmte Programmiersprache beschränkt sein.

3.2 Spezielle Anforderungen bausteinbasierter Anwendungen

Nachdem im vorangegangenen Abschnitt allgemeine Anforderungen an die Überwachung von Anwendungsdiensten angegeben wurden, soll in diesem Abschnitt auf die besonderen Anforderungen eingegangen werden, die sich durch die Zusammensetzung einer Anwendung aus Bausteinen ergeben. Hierzu wird das in Abschnitt 3.1.1 angegebene allgemeine Modell eines Anwendungsdienstes dahingehend erweitert, daß es die Erstellung und den Betrieb bausteinbasierter Anwendungen beschreibt. Davon ausgehend werden dann die Anforderungen an eine Managementlösung für die Überwachung bausteinbasierter Anwendungen ermittelt. Bei den resultierenden Anforderungen handelt es sich überwiegend um weitere funktionale Anforderungen; nicht-funktionale Anforderungen ergeben sich lediglich durch die Einführung der Rollen des Anwendungs- und Bausteinentwicklers.

3.2.1 Modellierung bausteinbasierter Anwendungsdienste

Der Unterschied zum in Abbildung 3.1 dargestellten allgemeinen Modell eines Anwendungsdienstes beschränkt sich auf die bausteinbasierte Struktur der Dienstimplementierung, also der den Dienst erbringenden Anwendung. Aus diesem Grund wird in Abbildung 3.2 nur die Verfeinerung der Dienstimplementierung betrachtet, das restliche Modell bleibt unverändert bestehen.

- Bausteinbasierte Anwendung
  Eine bausteinbasierte Anwendung stellt eine Unterklasse einer allgemeinen Anwendung dar.

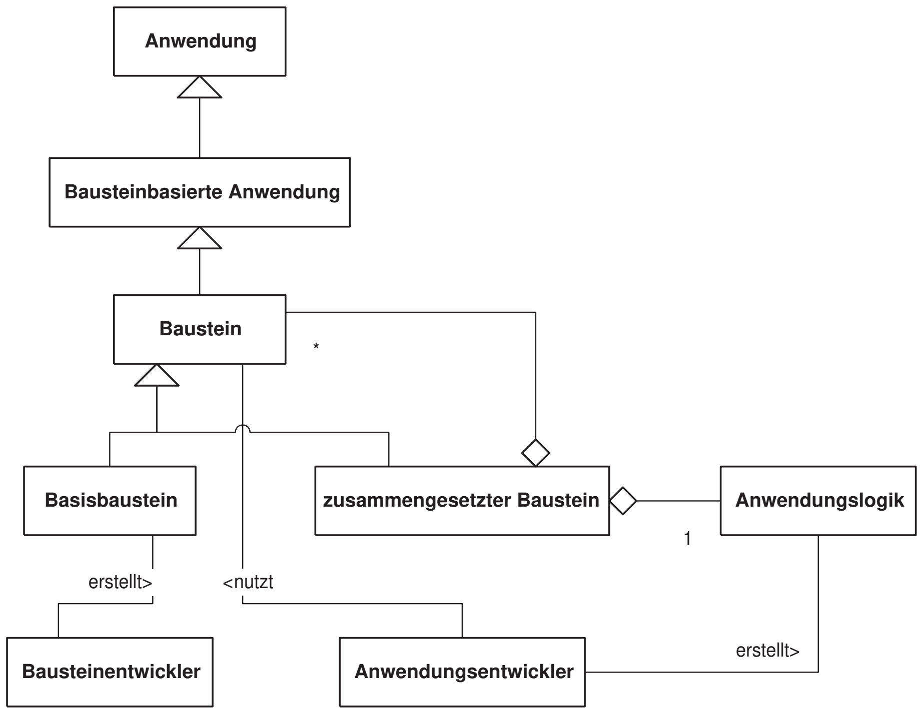

Abbildung 3.2: Modellierung bausteinbasierter Anwendungsdienste

- Baustein
  Jeder Baustein kann wiederum als bausteinbasierte Anwendung betrachtet werden und realisiert somit einen Anwendungsdienst, also Nutzungsfunktionalität in Form von Transaktionen an einer Nutzungsschnittstelle.
- Bausteinarten
  Man kann zwei Arten von Bausteinen unterscheiden, Basisbausteine und zusammengesetzte Bausteine, also Bausteine, die wiederum aus anderen Bausteinen zusammengesetzt sind:
  - Basisbausteine
    Basisbausteine sind nicht aus anderen Bausteinen zusammengesetzt und werden von Bausteinentwicklern erstellt.
  - Zusammengesetzte Bausteine
    Zusammengesetzte Bausteine bestehen aus Bausteinen (sowohl Basisbausteinen als auch wiederum aus zusammengesetzten) und einer Anwendungslogik, die die Verknüpfung der einzelnen Bausteine untereinander beschreibt.
- Anwendungslogik
  Die Anwendungslogik wird vom Anwendungsentwickler erstellt und beschreibt die Verknüpfung einzelner Bausteine zu einem zusammengesetzten Baustein.

Es lassen sich weiterhin also die folgenden zwei Rollen einführen und unterscheiden:

- Bausteinentwickler
- Anwendungsentwickler

3.2.2 Ermittlung der Anforderungen

In den folgenden Abschnitten werden die speziellen Managementanforderungen, die sich aus der bausteinbasierten Struktur von Anwendungen ergeben, herausgearbeitet. Als wesentliche Anforderungsquellen dienen dabei die rekursive Struktur bausteinbasierter Anwendungen, die beteiligten Rollen, die unterschiedlichen Varianten von Anwendungslogik, die bei der Erstellung von Anwendungen Verwendung finden sowie die unterschiedlichen Arten von Bausteinen, die zum Einsatz kommen.

3.2.2.1 Anforderungen aus der rekursiven Struktur der Anwendung

Betrachtet man die Struktur einer bausteinbasierten Anwendung, so erkennt man, daß jeder Baustein wiederum aus Bausteinen zusammengesetzt sein kann. Um einen einheitlichen Zugriff auf all diese Bausteine zu ermöglichen, ist es erforderlich, daß alle Bausteine, unabhängig von der Ebene auf der sie sich befinden, über dieselbe Struktur der Managementschnittstelle verfügen müssen.

3.2.2.2 Anforderungen aus den beteiligten Rollen

Im oben modellierten Szenario lassen sich die folgenden vier Rollen unterscheiden:

- Diensterbringer
- Dienstnehmer
- Anwendungsentwickler
- Bausteinentwickler

Aus jeder dieser Rollen ergeben sich vor dem Hintergrund der Erbringung von Anwendungsdiensten auf Basis bausteinbasierter Anwendungen unterschiedliche Anforderungen. Hierbei ist zu beachten, daß unterschiedliche Zuordnungen der angegebenen Rollen zu organisatorischen Einheiten denkbar sind: Neben der klassischen, gemeinsamen Zuordnung von Anwendungsentwickler und Diensterbringer zu einer organisatorischen Einheit sind weitere Varianten denkbar. Beispielsweise wird zunehmend auch der eigentliche Dienstnehmer die Rolle des Anwendungsentwicklers übernehmen und selbständig aus den am Markt verfügbaren Bausteinen individuelle Dienste erstellen. Die Rolle des Bausteinentwicklers kann in beiden Fällen durch die organisatorische Einheit des Anwendungsentwicklers sowie durch unabhängige, dritte Organisationen verteilt realisiert werden.

Anforderungen des Diensterbringers

Wie bereits bei den allgemeinen Anforderungen angegeben, ist es für den Diensterbringer nicht nur von Bedeutung, eine Fehlfunktion zu erkennen, sondern auch deren Ursache identifizieren zu können. Im Falle bausteinbasierter Anwendungen bedeutet dies, daß die Möglichkeit existieren muß, eine Fehlfunktion bis zum verantwortlichen Baustein zu verfolgen. Dieser Detaillierungsgrad ist sinnvoll, da einzelne Bausteine die kleinste Einheit darstellen, die vom Diensterbringer als Teil der Anwendung wahrgenommen werden kann. Somit kann er bei Fehlverhalten oder nicht ausreichender Dimensionierung eines Bausteins Gegenmaßnahmen wie einen Neustart oder die Ersetzung des Bausteins ergreifen.

Andererseits ist es für den Diensterbringer von Bedeutung, mögliche Fehler bereits zu erkennen, bevor sie zu Dienstausfällen beim Dienstnehmer führen. Für den Fall bausteinbasierter Anwendungen bedeutet dies, daß der Status der einzelnen Bausteine überwacht werden muß und aufgrund dieser Information auf den Zustand der jeweiligen Anwendung und damit des zugehörigen Dienstes geschlossen werden muß. Es sollte möglich sein, sowohl die betroffenen Dienste als auch die betroffenen Benutzer identifizieren zu können.

Die Überwachung von Systemparametern, um den Ressourcenverbrauch einzelner Bausteine ermitteln zu können, ist nur in Ausnahmefällen möglich. Systemparameter können üblicherweise für Prozesse bzw. *Threads* gemessen werden. Da aber im allgemeinen keine eineindeutige Abbildung von Bausteinen auf Prozesse bzw. *Threads* möglich ist, ist auch keine Zuordnung der entsprechenden Parameter möglich. Beispielsweise läßt sich zwar der Speicherbedarf eines Prozesses bestimmen, da innerhalb dieses Prozesses aber mehrere Bausteine zur Ausführung kommen können, ist es nicht möglich, den Speicherbedarf eines einzelnen Bausteins zu ermitteln. Ein wesentlicher Parameter, für den eine Zuordnung möglich ist, ist die verbrauchte Prozessorzeit. Diese kann daher für jeden einzelnen Baustein angegeben werden. Darüber hinaus sollte zumindest die Möglichkeit bestehen, alle Bausteine anzugeben, die in einem bestimmten Prozeß bzw. *Thread* ausgeführt wurden, um eine grobe Zuordnung der Systemparameter zu Bausteinen zu gestatten.

Anforderungen des Dienstnehmers

Für den Dienstnehmer ist die Dienstimplementierung transparent. Somit ist ihm auch nichts von der bausteinbasierten Architektur der den Dienst erbringenden Anwendung bekannt. Dies bedeutet, daß auch keine speziellen Anforderungen des Dienstnehmers, die über die bereits angegeben allgemeinen Anforderungen hinausgehen, existieren können.

Anforderungen des Anwendungsentwicklers

Wie bereits erwähnt, ist es wünschenswert, daß der Anwendungsentwickler für die Bereitstellung der Managementfunktionalität nur die Mechanismen verwenden darf, die er auch für die Erstellung der eigentlichen Anwendung einsetzt. Für die bausteinbasierte Entwicklung, bei der der Anwendungsentwickler keinen Zugriff auf den Code der Bausteine besitzt und somit nur über das *Customizing* der Bausteine das Verhalten der einzelnen Bausteine beeinflussen kann, bedeutet dies, daß er auch nur bereits vorhandene Managementfunktionalität eines Bausteins der jeweiligen Anwendung entsprechend anpassen bzw. aktivieren kann. Die eigentliche Instrumentierung der Bausteine muß also bereits durch den Bausteinentwickler erfolgen. Eine darüberhinausgehende Instrumentierung der erstellten Anwendung ist nur durch die eingesetzte Entwicklungsumgebung möglich.

Anforderungen des Bausteinentwicklers
Die zentrale Anforderung des Bausteinentwicklers ist es, den zusätzlichen Aufwand so gering wie möglich zu halten, um sich auf sein primäres Ziel, nämlich die Erstellung von Bausteinen konzentrieren zu können. Dies bedeutet, daß ein möglichst großer Teil der Managementfunktionalität automatisch von der Entwicklungsumgebung in die zu erstellende Anwendung eingebracht werden muß. Nur wenn Informationen erforderlich sind, die außerhalb eines Bausteins nicht verfügbar sind, sollten diese vom Bausteinentwickler durch Instrumentierung des Bausteins zur Verfügung gestellt werden müssen.

3.2.2.3 Anforderungen aus den verschiedenen Varianten von Anwendungslogik

Die zu erstellende Managementlösung soll unabhängig davon verwendbar sein, wie die Verknüpfung der einzelnen Bausteine zu einer lauffähigen Anwendung erfolgt. Im folgenden wird auf die unterschiedlichen Methoden näher eingegangen.

Art der Verknüpfung
Unter der Art der Verknüpfung soll verstanden werden, wie die Verknüpfung von Bausteinen zu Anwendungen erfolgt. Es ergeben sich die folgenden Varianten:

- Synchrone Operationsaufrufe:
  Ein Baustein ruft eine Operation eines anderen Bausteins auf und blockiert bis zur Lieferung des Ergebnisses der Operation.

- Asynchrone Operationsaufrufe:
  Ein Baustein ruft eine Operation eines anderen Bausteins auf ohne dabei aber zu blockieren.

- Events:
  Ein Baustein versendet ein Event, das von beliebigen anderen, registrierten Bausteinen als Anstoß für eine Operation betrachtet werden kann. Hierbei ist dem „aufrufenden“ Baustein nicht bekannt, welche anderen Bausteine durch das Versenden des Events angestoßen werden. Events werden normalerweise asynchron ausgeliefert.

Im Gegensatz zur synchronen Kommunikation ist bei der asynchronen bzw. bei der eventbasierten Kommunikation nicht sichergestellt, daß der Kontrollfluß jemals zum Aufrufer zurückkehrt. Somit ist es z.B. nicht möglich, die Dauer einer Transaktion eines aufgerufenen Bausteins durch einfache Messung der Zeitdifferenz zwischen Aufruf und Rückkehr zu bestimmen. Es ist vielmehr erforderlich, jeweils beim Betreten und beim Verlassen eines Bausteines Messungen vorzunehmen und entsprechend zu korrelieren.

Verteilung der Anwendungslogik
Die Anwendungslogik kann sowohl auf ein einzelnes System beschränkt als auch über mehrere Systeme verteilt sein. Um eine Überwachung der gesamten Anwendung zu ermöglichen, ist es erforderlich, daß eine Korrelation der Informationen entfernter Bausteine zu der jeweiligen Anwendung auf dem lokalen System möglich ist.

Parallelität der Anwendungslogik
Der Ablauf einer Anwendung kann entweder auf einen einzelnen Kontrollfluß beschränkt sein oder aber mehrere Kontrollflüsse umfassen, um z.B. durch Ausnutzung dieser Parallelität eine gleichzeitige Bearbeitung mehrerer Transaktionen oder die effizientere Bearbeitung einer Transaktion zu gewährleisten. Eine Managementlösung muß in der Lage sein, die verschiedenen Kontrollflüsse einer Anwendung zu überwachen und mit den entsprechenden Transaktionen zu korrelieren.

3.2.2.4 Anforderungen aus den verschiedenen Arten von Bausteinen

Bausteine können als Teil einer Anwendung mit dieser geliefert, verteilt und installiert werden. Ist dies der Fall, dann ist bereits zum Zeitpunkt der Anwendungserstellung bekannt, welche Bausteine zum Einsatz kommen werden und somit überwacht werden müssen. Andererseits ist aber auch denkbar, daß Bausteine als Teil des Zielsystems, auf dem die Anwendung installiert werden soll, bereits existieren und von der Anwendung erst zur Laufzeit identifiziert und kontaktiert werden. In diesem Falle ist es nicht möglich, im Vorhinein zu bestimmen, welche Bausteine zur Laufzeit der Anwendung tatsächlich zum Einsatz kommen werden. Auch ist dann davon auszugehen, daß die einzelnen Bausteine nicht nur von einer einzelnen Anwendung genutzt werden, sondern von mehreren, unterschiedlichen Anwendungen aufgerufen werden können. Eine Managementlösung für bausteinbasierte Anwendungen sollte in der Lage sein, mit beiden vorgestellten Varianten umzugehen.

Die Managementlösung sollte die Auswahl an Bausteinen nicht einschränken. Dies bedeutet, daß keine besonderen Anforderungen an die Bausteine gestellt werden sollten. Schreibt die Managementlösung spezielle Anpassungen der Bausteine vor, so ist für die Einbindung von Legacy-Bausteinen, also Bausteinen, die nicht den neuen Anforderungen genügen, Sorge zu tragen.

3.3 Zusammenfassung

Im folgenden sollen sowohl die in Abschnitt 3.1 ermittelten allgemeinen Anforderungen als auch die in Abschnitt 3.2 ermittelten speziellen Anforderungen bausteinorientierter Anwendungen nochmals zusammengefaßt dargestellt werden:

- Transaktionsbasierte Überwachung
  Die Überwachung der in der Dienstvereinbarung festgelegten Benutzertransaktionen muß möglich sein. Insbesondere muß überwacht werden können:
  - Die Antwortzeit jeder Instanz einer BTA
  - Die Transaktionsdauer jeder Instanz einer BTA
  - Die erfolgreiche bzw. nicht erfolgreiche Erbringung jeder Instanz einer BTA
- Bausteinbasierte Überwachung
  An einer, von der Nutzungschnittstelle des Bausteins getrennten, Managementschnittstelle müssen, unabhängig davon, ob es sich um einen Basisbaustein oder einen zusammengesetzen Baustein handelt, die folgenden Informationen ermittelt werden können:
  - Zeitdauer jeder Instanz einer Transaktion des Bausteins
  - Erfolgreiche bzw. nicht erfolgreiche Erbringung jeder Instanz einer Transaktion des Bausteins
  - Verfügbarkeit des Bausteins
  - Ressourcenverbrauch (soweit möglich) des Bausteins
- Auswirkungen des Ausfalls von Bausteinen
  Es muß möglich sein, die Auswirkungen des Ausfalls eines Bausteins auf die Verfügbarkeit
  - von Diensten
  - von Transaktionen eines Dienstes
  - aufgeschlüsselt nach Nutzer

  angeben zu können.
- Aufwandsminimierung
  - ausschließliche Verwendung ohnehin eingesetzter Mechanismen
  - hoher Grad an Automatisierung
- Einsatzspektrum
  - für beliebige Arten von Anwendungslogik

  - für beliebige Arten von Bausteinen
  - Integration von Legacy-Bausteinen
- Sonstige Anforderungen
  - Informationen über Subtransaktionen müssen zur jeweils übergeordneten Transaktion zugeordnet werden können.
  - Möglichkeit der Abbildung von Systemparametern auf Bausteine bzw. Mengen von Bausteinen
  - Vergleichbarkeit und Verläßlichkeit der gelieferten Information
  - Messung der tatsächlichen Dienstnutzung
  - Geringe Auswirkung auf den überwachten Dienst
  - Zeitnahe Überwachung
  - Generische Verwendbarkeit

Kapitel 4

Status Quo: Überwachung von Anwendungsdiensten

Die Entwicklung der letzten Jahre hat eine große Anzahl an Ansätzen und Lösungen für das Management von Anwendungen hervorgebracht. Der folgende Abschnitt soll die aktuellen Ansätze vorstellen, bewerten und einander gegenüberstellen. Hierzu wird in Abschnitt 4.1 zunächst eine Klassifikation der aktuellen Ansätze eingeführt, die eine leichtere Bewertung der unterschiedlichen Ansätze ermöglicht. In Abschnitt 4.2 werden dann existierende Ansätze vorgestellt und bewertet.

4.1 Klassifikation unterschiedlicher Ansätze

Wie bereits in Abschnitt 2.2 dargestellt läßt sich das Anwendungsmanagement z.B. in folgende Bereiche unterteilen:

- Überwachung (*Monitoring*)
- Software-Verteilung und -Installation (*SW-Distribution, SW-Deployment*)
- Bestandsführung (*Inventory Management*)
- Benutzeradministration (*User Administration*)

Viele der im folgenden vorgestellten Lösungen beschäftigen sich nicht mit dem Teilbereich der Anwendungsüberwachung und sind somit für die vorliegende Arbeit nur am Rande von Bedeutung.

4.1.1 Klassifikation der Anwendungsüberwachung

Für diejenigen Ansätze, die sich mit der eigentlichen Anwendungsüberwachung beschäftigen, bietet sich eine Klassifizierung bezüglich der Gewinnung der benötigten Managementinformationen an [HaRa 00]. Abbildung 4.1 stellt ein einfaches Modell einer verteilten Anwendung und der darunterliegenden Infrastruktur dar. Die Managementinformation kann entweder aus der Anwendung selbst oder aber aus der darunterliegenden Infrastruktur (sowohl aus den Systemen als auch aus dem Netz) gewonnen werden. Bei Überwachung der Anwendung selbst kann weiterhin unterschieden werden, ob die Messung sich über die gesamte Anwendung erstreckt oder auf die *Client*-Teile beschränkt ist.

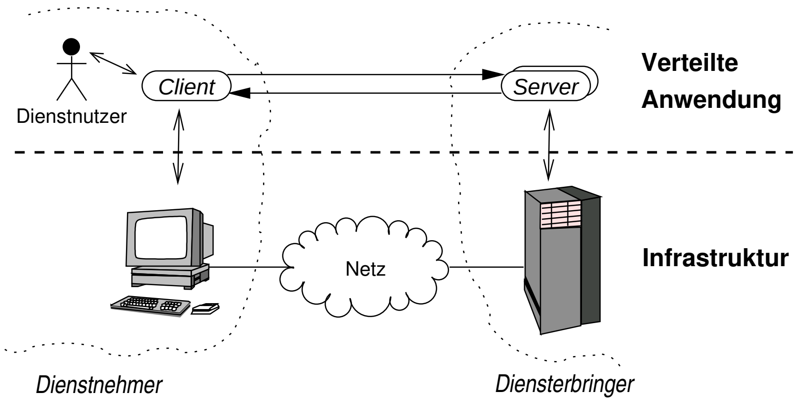

Abbildung 4.1: Möglichkeiten der Überwachung verteilter Anwendungen

Es ergeben sich also die folgenden vier grundlegenden Klassen von Techniken zur Gewinnung der benötigten Information bei der Anwendungsüberwachung:

- Überwachung des Netzverkehrs
- Überwachung von Systemparametern
- *Client*-seitige Anwendungsüberwachung
- Überwachung der Gesamtanwendung

In den folgenden Abschnitten werden diese vier Techniken detailliert beschrieben und bewertet, um später die Einordnung und Bewertung konkreter Lösungen zu erleichtern.

4.1.1.1 Überwachung des Netzverkehrs

Eine weitverbreitete Technik der Anwendungsüberwachung besteht in der Überwachung des Netzverkehrs. Idee ist hierbei, die Beobachtung der einzelnen übertragenen Pakete und die Zuordnung dieser Pakete zu Anwendungen und Transaktionen. Somit wird es möglich, Aussagen über den Zustand der einzelnen Anwendungen zu treffen.

Der wesentliche Vorteil dieser Technik ist, daß sie für beliebige verteilte Anwendungen eingesetzt werden kann, ohne hierbei in die eigentliche Anwendung eingreifen zu müssen.

Problematisch ist hierbei jedoch die relativ komplexe Abbildung der erhaltenen Information auf nutzerrelevante Daten. Daher sind die meisten Werkzeuge nur für die Überwachung einer geringen Zahl von Standardanwendungen ausgelegt. Eine echte Messung aus Nutzersicht ist ganz ausgeschlossen, da nur die Kommunikation zwischen Systemen, nicht aber zwischen Benutzer und System überwacht werden kann. Informationen über den Zustand der Anwendung oder ihrer einzelnen Bausteine sind auf diese Art und Weise ebenfalls nicht zu erhalten. Aufgrund der hohen Komplexität ist eine zeitnahe Überwachung häufig unmöglich.

Der Haupteinsatzbereich dieser Techniken liegt daher vorwiegend im Bereich der Netzplanung und Leistungsüberwachung der Netzinfrastruktur, nicht aber in der Überwachung von konkreten Dienstvereinbarungen und nutzerorientierten Parametern von Anwendungsdiensten.

4.1.1.2 Überwachung von Systemparametern

Eine weitere Technik der Anwendungsüberwachung besteht darin, die darunterliegenden Systeme zu überwachen. Hierbei werden z.B. Parameter wie die Prozessornutzung oder der Speicherbedarf einer Anwendung gemessen. Weitere Beispiele sind die Anzahl offener Dateien einer Anwendung oder der aktuelle Status eines Prozesses oder Threads. Diese Systemparameter können jeweils nur für Prozesse bzw. Threads ermittelt werden; um zu Aussagen über den Zustand einer bestimmten Anwendung zu gelangen ist eine Abbildung der gewonnenen Information erforderlich.

Der Hauptvorteil dieser Technik ist die große Erfahrung, die in den letzten Jahren mit der Ermittlung dieser Informationen gesammelt werden konnte. Es ist somit verhältnismäßig einfach möglich, die Information aus dem System auszulesen und über definierte Schnittstellen zur Verfügung zu stellen. Diese Informationen können für den Diensterbringer von großer Bedeutung bei der Bestimmung der Systembelastung und bei der Identifikation spezifischer Probleme sein.

Die Überwachung von Systemparametern zum Zwecke der Anwendungsüberwachung besitzt darüber hinaus jedoch eine große Anzahl an Nachteilen: Das grundlegende Problem ist, daß die üblicherweise in Dienstvereinbarungen festgelegten und somit zu überwachenden Parameter (wie z.B. Antwortzeiten oder erfolgreiche Beendigung einer Transaktion) mit dieser Technik nicht ermittelt werden können. Die Abbildung der Systemparameter auf aussagekräftige und nutzer-orientierte Parameter ist schwierig oder ganz unmöglich. So würde beispielsweise ein Prozeß, der eine Endlosschleife ausführt, aus Systemsicht als einwandfrei funktionierend beurteilt werden, obwohl die zu erbringenden Benutzertransaktionen ausnahmslos scheitern.

4.1.1.3 *Client*-seitige Anwendungsüberwachung

Die Überwachung einer Anwendung ausschließlich auf *Client*-Seite stellt die dritte Überwachungstechnik dar. Im Gegensatz zu den bisher vorgestellten Ansätzen ermöglicht es diese Methode, die in der Dienstvereinbarung festgelegten, nutzer-orientierten Parameter zu messen. Grundproblem hierbei ist, daß diese Technik zwar die Überwachung der in der Dienstvereinbarung festgelegten Parameter gestattet, bei Auftreten eines Fehlers jedoch keinerlei Hinweise auf die Ursache des Problems liefern kann. Hierfür müssen immer zusätzliche Techniken zum Einsatz kommen.

Die *Client*-seitige Anwendungsüberwachung läßt sich wiederum in zwei Methoden unterteilen:

- Simulation von Benutzertransaktionen
- Oberflächenbasierte Techniken

Diese werden in den folgenden Abschnitten vorgestellt und bewertet.

Simulation von Benutzertransaktionen
Durch Generierung von simulierten Benutzertransaktionen (BTAs) und Messung der Zeit für die Bearbeitung einer derartigen BTA ist es möglich, die Verfügbarkeit von Anwendungsdiensten zu überprüfen. Mehrere – evtl. im Netz verteilte – Simulationsagenten schicken Testanfragen an die Serversysteme und ermitteln die Zeit, die bis zur Beantwortung der Anfrage vergeht. Falls die erzielte Antwortzeit einen festzulegenden Schwellwert überschreitet oder die erhaltene Antwort ein nicht korrektes Ergebnis enthält, kann der Simulationsagent ein Managementsystem beispielsweise mittels eines Events informieren.

Die Verwendung simulierter Transaktionen ist nicht geeignet, um die Erfüllung einer Dienstvereinbarung zu überprüfen. Es werden nicht die tatsächlich von einem Nutzer ausgeführten Transaktionen gemessen, sondern künstlich erzeugte Testtransaktionen. Somit kann diese Technik nur verwendet werden, um stichprobenartig die Verfügbarkeit von Serversystemen zu überprüfen. Um zu Ergebnissen zu gelangen, die die tatsächlichen Erfahrungen des Nutzers annähernd widerspiegeln, ist es erforderlich, Messungen in sehr kurzen Abständen auszuführen. Eine Folge hieraus ist eine erhebliche zusätzliche Belastung des Servers und des Netzes durch die synthetischen Transaktionen, die zu einer weiteren Leistungsabnahme führen kann. Ein weiteres Problem stellt die Platzierung der Simulationsagenten dar: Um zu aussagekräftigen Ergebnissen zu gelangen, sollten diese möglichst im selben Subnetz platziert werden wie die tatsächlich von den Nutzern eingesetzten *Clients*. Dies führt zu Verteilungsproblemen in großen Umgebungen. Ein weiterer Nachteil ist, daß Testtransaktionen auf den tatsächlichen Unternehmensdaten zu inkonsistenen Daten führen können bzw. jeweils nach Durchführung der Testtransaktionen einen *Rollback* der Transaktion erforderlich machen.

Oberflächenbasierte Techniken
Um in der Lage zu sein, die tatsächlich durchgeführten BTAs zu messen, ohne dabei aber den *Source-Code* der *Clients* verändern zu müssen, wurden kürzlich Werkzeuge auf den Markt gebracht, die als oberflächenbasierte Techniken bezeichnet werden können. Ausgehend von der Beobachtung, daß jede Benutzertransaktion durch Verwendung eines Elements der Benutzeroberfläche ausgelöst wird und endet (z.B. Klicken auf einen *Weblink* und Darstellung der geladenen Seite), kann die benötigte Information durch einfache Beobachtung der auftretenden Oberflächenereignisse ermittelt werden. Hierzu bedarf es eines Software-Agenten, der auf dem *Client*-Rechner installiert wird. Dieser sammelt die Informationen über die unterschiedlichen BTAs aus Nutzersicht.

Der wesentliche Vorteil dieser Technik ist, daß die tatsächlichen BTAs aus Nutzersicht gemessen werden können, ohne hierzu Zugriff zum *Source-Code* der Anwendung zu erfordern.

Größtes Problem dieses Ansatzes – neben den allgemeinen Problemen ausschließlich *Client*-seitiger Überwachung – ist die große Komplexität, die durch die nachträgliche Abbildung von Oberflächenereignissen auf BTAs hervorgerufen wird. Alle Möglichkeiten zu identifizieren, wie eine BTA gestartet bzw. beendet werden kann, ist nur zum Zeitpunkt der Anwendungserstellung sinnvoll möglich und kann nachträglich kaum durchgeführt werden. Somit ist der offensichtliche Vorteil, keinen Zugriff auf den *Source-Code* der Anwendung zu benötigen, hinfällig.

4.1.1.4 Überwachung der Gesamtanwendung

Wie bereits erwähnt können reine *client*-seitige Ansätze bei Identifikation eines Problems keine Informationen darüber liefern, was die eigentliche Ursache des Problems ist. Daher sind Techniken erforderlich, die sowohl auf *Client*- als auch auf *Server*-Seite messen. Diese Verfahren basieren auf Informationen, die von den Anwendungen selbst geliefert werden. Je nach Art und Umfang der Managementinstrumentierung lassen sich wiederum zwei unterschiedliche Verfahren unterscheiden:

- Anwendungsinstrumentierung
- Anwendungsbeschreibung

Die folgenden Abschnitte beschreiben diese beiden Verfahren und stellen ihre Vor- und Nachteile heraus.

Anwendungsinstrumentierung
Man spricht von Anwendungsinstrumentierung, wenn spezieller Managementcode in den *Source-Code* der Anwendung eingefügt wird. Dieser Managementcode ermittelt die gewünschte Information und überträgt sie über eine Managementschnittstelle an beliebige Managementsysteme.

Dieser Ansatz erlaubt es, alle geforderten Informationen zu ermitteln, den tatsächlichen Zustand der Anwendung anzugeben und die tatsächliche Dauer der einzelnen BTAs zu bestimmen. Durch Messung von Subtransaktionen ist es möglich, bei auftretenden Problemen die genaue Fehlerquelle zu lokalisieren.

Dennoch wird Anwendungsinstrumentierung nur sehr vereinzelt eingesetzt. Dies liegt im wesentlichen an der großen Komplexität und dem damit verbundenen Aufwand, den die Managementinstrumentierung einer Anwendung mit sich bringt. Es gibt wenig Werkzeuge, die den Anwendungsentwickler bei der Instrumentierung unterstützen, ein Großteil der anfallenden Aufgaben besteht im manuellen Erstellen von entsprechendem Programmcode. Auch die Korrelation von Subtransaktionen zu BTAs erfolgt manuell durch den Anwendungsentwickler. Eine nachträgliche Instrumentierung einer Anwendung ist nahezu ausgeschlossen, da zum einen der *Source-Code* der Anwendung meist nicht vorliegt, zum anderen aber auch ein tiefes Verständnis des Codes erforderlich ist, um eine sinnvolle Instrumentierung durchführen zu können. Die Verbreitung wird weiterhin erschwert durch mehrere konkurrierende Ansätze, aus denen sich bislang kein *de-facto*-Standard etablieren konnte.

Anwendungsbeschreibung

Aus oben bereits beschriebenen Gründen sind managementinstrumentierte Anwendungen nur in wenigen Ausnahmefällen verfügbar. Dennoch liefern die meisten Anwendungen in irgendeiner Form Statusinformationen. Diese werden aber meist nicht explizit zu Managementzwecken an einer definierten Schnittstelle zur Verfügung gestellt, sondern existieren anwendungsspezifisch z.B. in Form von *Logfiles* oder Status-*events*.

Um diese Information für Managementsysteme verfügbar zu machen, kommen Anwendungsbeschreibungs-Techniken zum Einsatz. Die Beschreibung umfaßt hierbei einerseits, wie die entsprechende Information ermittelt werden kann, andererseits aber auch, wie diese Information zu interpretieren ist.

Der wesentliche Vorteil der Anwendungsbeschreibung ist, daß eine Anwendungsbeschreibung auch nachträglich erfolgen kann, ohne Zugriff zum *Source-Code* der Anwendung zu erfordern. Sinnvoll wäre natürlich wiederum die Erstellung der Beschreibung durch den Anwendungsentwickler.

Neben dem nicht unerheblichen Aufwand für die Erstellung der Anwendungsbeschreibung ist der wesentliche Nachteil dieses Ansatzes die Schwierigkeit der Abbildung der verfügbaren Information. Die ohnehin verfügbare Information läßt sich im allgemeinen nur schwer auf die benötigte Information abbilden. Insbesondere im Bereich des Leistungsmanagements sind meist nur wenige Informationen erhältlich. Darüber hinaus sind häufig sogenannte Monitore erforderlich, um die Information aus der Anwendung zu gewinnen. Hierbei kann in der Regel nur sehr wenig Information mit Hilfe von Standardmonitoren ermittelt werden; stattdessen ist häufig die aufwendige Erstellung anwendungsspezifischer Spezialmonitore erforderlich.

4.1.1.5 Zusammenfassung

Die unterschiedlichen Techniken der Anwendungsüberwachung werden in Abbildung 4.2 noch einmal zusammenfassend den in Abschnitt 3.1 ermittelten allgemeinen Anforderungen an die Überwachung von Anwendungsdiensten gegenübergestellt. Hierbei bedeutet „++“ sehr gute Erfüllung einer Anforderung, während „– –“ sehr schlechte Erfüllung bedeutet. „+“ und „–“ stellen weitere Abstufungen dar, während „o“ einen unentschiedenen Wert repräsentiert. Aus der Tabelle ist ersichtlich, daß keines der vorgestellten Verfahren in der Lage ist, alle aufgestellten Anforderungen zu erfüllen. Die Tabelle zeigt weiterhin, daß nur die Anwendungsinstrumentierung in der Lage ist, die benötigte Information zu liefern, dabei allerdings einen erheblichen zusätzlichen Aufwand erfordert.

Techniken	Anforderungen	Dienst–/Nutzerorientierung	Tatsächliche Dienstnutzung	Detailinformationen	Geringer Aufwand	Geringe Auswirkung	Zeitnahe Überwachung	Generische Verwendbarkeit
Überwachung des Netzverkehrs		o/–	o	–	–	o/+	–	++
Überwachung von Systemparametern		––	o	o/+	o	++	++	++
Client–seitige Überwachung	Simulation von BTAs	+	––	––	++/+	––	++	++
	Oberflächenbasierte Techniken	++	++	––	–	++	++	–
Überwachung der Gesamtanwendung	Anwendungsinstrumentierung	++	++	++	––	+	++	o
	Anwendungsbeschreibung	o	o	o	–	++	+	++

Abbildung 4.2: Zusammenfassung: Gegenüberstellung von Überwachungstechniken und Anforderungen

4.2 Untersuchung aktueller Ansätze zur Anwendungsüberwachung

Anhand der oben angegeben Klassifikation der unterschiedlichen Techniken für die Anwendungsüberwachung sollen im folgenden Abschnitt konkrete Ausprägungen dieser Techniken vorgestellt und bewertet werden. Hierbei werden die Ansätze von Standardisierungsgremien, aktuelle Forschungsansätze sowie am Markt verfügbare herstellerspezifische Lösungen bestimmter Produkte betrachtet.

4.2.1 Standards

In diesem Abschnitt sollen die Ansätze aus dem Bereich der Anwendungsüberwachung der unterschiedlichen Standardisierungsorganisationen vorgestellt und bewertet werden. Wesentliche Vertreter sind beispielsweise die *Internet Engineering Task Force (IETF)*, die diverse MIBs zum Thema Anwendungsmanagement standardisiert hat und die *Desktop Management Task Force (DMTF)* mit dem im Rahmen des *Common Information Model (CIM)* spezifizierten *Application Management Model* sowie der *Distributed Application Performance Working Group (DAP WG*, mittlerweile *Metrics WG*). Die *Open Group* standardisiert zwei Programmierschnittstellen für die Anwendungsinstrumentierung, die *Application Response Measurement API (ARM API)* sowie die *Application Instrumentation & Control API (AIC API)*.

4.2.1.1 Internet Engineering Task Force

Die *Internet Engineering Task Force (IETF)* hat sich als das Standardisierungsgremium für auf dem Internet-Protokoll basierende Datennetze durchgesetzt. Mitarbeit in der *IETF* steht jedermann offen. Sie setzt sich derzeit vorwiegend zusammen aus Herstellern und Betreibern von Datennetzen, aber auch aus Forschern, Regierungsbehörden und sonstigen interessierten Gruppen und Personen. Ende der achtziger Jahre wurde mit der Festlegung der Internet-Managementarchitektur, die nach dem verwendeten Managementprotokoll oft auch als „SNMP-Management“ bezeichnet wird, begonnen.

Der Standardisierungsprozeß der IETF wird in [Brad 96] beschrieben. Die Spezifikationen werden als *Request for Comments (RFCs)* bezeichnet und mit einer laufenden Nummer versehen. Je nach Status der Standardisierung spricht man von *Proposed Standards*, *Draft Standards* bzw. *Internet Standards*. Nicht mit den *Draft Standards* zu verwechseln sind die sogenannten *Internet Drafts*. Als *Internet Draft* wird eine Spezifikation bezeichnet, bevor sie beginnt, den Standardisierungsprozeß zu durchlaufen.

Ziel bei der Festlegung des Internet-Managements war es, eine einfache Managementarchitektur für das Management von Datennetzen zu schaffen. Die wesentlichen, zunächst im

Rahmen der ersten Version (SNMPv1) entstandenen Standards sind RFC 1155 [RoMc 90], der das Informationsmodell beschreibt und RFC 1157 [CFSD 90], der das *Simple Network Management Protocol (SNMP)* definiert. Wenig später entstand RFC 1213 [McRo 91], der die ursprüngliche Internet-MIB, also die Managementinformation, die von jedem überwachten System zur Verfügung gestellt werden muß, beschreibt.

In der Folgezeit wurde versucht, gewisse Mängel des SNMP-Managements – z.B. bzgl. der Übertragung großer Datenmengen und im Bereich der Sicherheit – durch die Nachfolgeversion SNMPv2 zu beheben [MPS 99, GCM^{+} 96a, GCM^{+} 96b]. Diese konnte sich aber am Markt nicht durchsetzen. Mittlerweile sind die Arbeiten an der dritten Version des Internet-Management in Gange [WHP 99].

In den letzten Jahren erkannte die IETF, daß sich integriertes Management nicht auf den Bereich des Netzmanagements beschränken kann, sondern ebenfalls Aspekte des System-, Anwendungs- und Dienstmanagements berücksichtigen muß. So entstanden im Bereich des generischen Managements von Anwendungen und von Anwendungsdiensten die folgenden MIBs:

- Host Resources MIB (RFC 1514) [GrWa 93]
- Network Services Monitoring MIB (RFC 1565) [KiFr 94a]
- „System Application MIB" (RFC 2287) [KrSa 98]
- Application Management MIB (RFC 2564) [KKPS 99]
- Application Perfomance Measurement MIB (Internet Draft) [Wald 01]

Darüber hinaus wurden folgende MIBs entwickelt, die die oben angegebenen, generischen MIBs um Informationen spezifischer Anwendungsfelder erweitern:

- Mail Monitoring MIB (RFC 1566) [KiFr 94b]
- X.500 Directory Monitoring MIB (RFC 1567) [MaKi 94]
- Relational Database Management System MIB (RFC 1697) [BEP^{+} 94]
- „WWW Service MIB" (RFC 2594) [HKS 99]

Neben diesen MIBs werden in den folgenden Abschnitten die Arbeiten der *Application Configuration Access Protocol (ACAP) Working Group* und der mittlerweile aufgelösten *Realtime Traffic Flow Measurement (RTFM) Working Group* der IETF vorgestellt.

Host Resources MIB

Die *Host Resources MIB (HR MIB)* [GrWa 93] war der erste Versuch, das Internet-Management in Richtung des System- und Anwendungsmanagements zu erweitern. Wie der Name der MIB schon nahelegt, war der eigentliche Fokus dieser MIB das Management von Rechensystemen. Drei der sechs Gruppen der

HR MIB befassen sich jedoch mit der auf den Systemen installierten Software. Es sind dies die `Host Resources Running Software Group`, die `Host Resources Running Software Performance Group` sowie die `Host Resources Installed Software Group`.

Mit Hilfe der in diesen drei Gruppen modellierten Information läßt sich feststellen, welche Software auf dem jeweiligen System installiert ist. Darüber hinaus läßt sich ermitteln, welche Software aktuell ausgeführt wird, sowie wieviel Ressourcen (CPU und Speicher) dabei benötigt werden. Die Überwachung laufender Software findet prozeßbasiert statt, es existiert also ein Eintrag für jeden auf dem System ablaufenden Prozeß.

Die HR MIB ermittelt die benötigte Managementinformation durch Überwachung von Systemparametern und ist somit nicht in der Lage, die für die Überwachung von Anwendungsdiensten benötigte Information zur Verfügung zu stellen.

Network Services Monitoring MIB

Die *Network Services Monitoring MIB (NSM MIB)* [KiFr 94a] definiert Attribute, die die Überwachung spezieller Netzdienste gestatten. Unter einem Netzdienst wird hierbei eine spezielle Anwendung wie z.B. ein Datei-, Druck- oder Verzeichnisdienst verstanden. Die NSM MIB fokussiert ausdrücklich auf dynamische Informationen und geht davon aus, daß statische Informationen (z.B. eine Liste der installierten Anwendungen) in anderen MIBs zur Verfügung stehen (z.B. *Host Resources MIB*). Sie besteht aus nur zwei Tabellen, der `applTable` sowie der `assocTable`.

Die `applTable` enthält einen Eintrag für jede auf dem System laufende Anwendung, die einen Netzdienst erbringt. Die hier verfügbaren Informationen konzentrieren sich auf allgemeine Informationen über die Kommunikationsbeziehungen der Anwendung (z.B. Anzahl ein- bzw. ausgehender Verbindungen, Anzahl fehlerhafter Verbindungsaufbauwünsche, etc.). In der `assocTable` finden sich dann detailliertere Informationen zu einzelnen Verbindungen, wie z.B. das verwendete Protokoll oder die Dauer der Verbindung.

Die Spezifikation der NSM MIB erwähnt nicht, wie die erforderliche Information zu ermitteln ist. Man kann aber davon ausgehen, daß sowohl Überwachung von Systemparametern als auch Anwendungsinstrumentierung zum Einsatz kommen dürfte. Trotzdem ist sie nicht in der Lage, die benötigte Information anzubieten, da die hier erhältliche Information sich auf Netzdienste und nicht auf einem Dienstnutzer angebotene und somit zu überwachende Anwendungsdienste bezieht.

Die NSM MIB wurde mit dem Ziel entwickelt, nur Information zu enthalten, die für alle Netzdienste gleichermaßen von Bedeutung ist. Gleichzeitig wurde vorgesehen, daß für spezielle Netzdienste bzw. Klassen von Netzdiensten spezielle MIBs zu entwickeln sind, die die spezifischen Gegebenheiten dieser Dienste berücksichtigen. Dies führte zur Entwicklung der folgenden drei Erweiterungen der NSM MIB:

- Mail Monitoring MIB (RFC 1566) [KiFr 94b]
- X.500 Directory Monitoring MIB (RFC 1567) [MaKi 94]
- Relational Database Management System MIB (RFC 1697) [BEP^{+} 94]

System Application MIB

Die sogenannte *„System Application MIB“* (offizieller Name: *Definitions of System-Level Managed Objects for Applications*) [KrSa 98] stellt eine Erweiterung der *Host Resources MIB* um Informationen bzgl. der auf dem System installierten und ablaufenden Anwendungen dar. Genau wie die HR MIB bezieht sich die *System Application MIB* somit nur auf ein einzelnes System und enthält nur Informationen, die ohne eine Managementinstrumentierung der Anwendung aus dem darunterliegenden System zu ermitteln sind. Sie enthält zwei wesentliche Gruppen, die `sysApplInstalled Group` und die `sysApplRun Group`. Die dritte Gruppe, die `sysApplMap Group` dient lediglich dazu, eine Abbildung zwischen den Prozessen des Systems und den entsprechenden Anwendungen bzw. Anwendungspaketen herzustellen.

Die `sysApplInstalled Group` liefert Informationen über die auf einem System installierten Anwendungspakete sowie deren Bestandteile, also ausführbare und nicht-ausführbare Dateien. Da die *System Application MIB* mit dem Ziel entworfen wurde, keine Managementinstrumentierug zu erfordern, ist dies nur möglich für Anwendungen, die mit speziellen Werkzeugen installiert wurden und somit in zentralen Systemverzeichnissen (z.B. *Windows-Registry*) enthalten sind. In der `sysApplRun Group` sind die Informationen über in Ausführung befindliche Anwendungen zusammengefaßt. Eine Tabelle enthält die aktuell ausgeführten Anwendungspakete, während eine weitere Tabelle Informationen über die auf dem System ausgeführten Prozesse enthält. Diese Informationen beziehen sich beispielsweise auf CPU-Nutzung, den benötigten Speicher oder offene Dateien des Prozesses. Die Abbildung zwischen den beiden Tabellen erfolgt, wie oben bereits erwähnt, über die `sysApplMap Group`. Beendete Prozesse und Anwendungspakete werden aus den Tabellen entfernt. Es existiert je eine weitere Tabelle, in die diese Informationen dann zur späteren Auswertung eingefügt werden.

Auch die *System Application MIB* kann die für eine Überwachung von Anwendungsdiensten benötigten Informationen nicht bereit stellen. Schon aufgrund der Tatsache, daß sie mit dem Ziel entwickelt wurde, nur Information zu verwenden, die aus dem darunterliegenden System ermittelbar ist, ist es nicht möglich, die gewünschte Information zur Verfügung zu stellen.

Application Management MIB

Die *Application Management MIB* [KKPS 99] erweitert die *System Application MIB* um die Informationen, die aufgrund des expliziten Verzichtes auf Instrumentierung der

Anwendung dort nicht berücksichtigt werden konnten. Sie führt eine dienstorientierte Sicht (*service-level view*) auf die Anwendungen ein, wobei mit Hilfe der Tabellen der `applServiceGroup` eine Abbildung zwischen einem Anwendungspaket bzw. einer zugehörigen Datei und einem angebotenen Dienst möglich ist. Es werden drei weitere Gruppen definiert, die `applChannelGroup`, die `applPastChannelGroup` sowie die `applElmtRunControlGroup`.

Die `applChannelGroup` der *Application Management MIB* liefert Informationen über die E/A-Kanäle der Anwendung. Im Rahmen der *Application Management MIB* werden unter E/A-Kanälen die offenen Dateien und Netzverbindungen einer Anwendung verstanden. Diese Informationen beziehen sich vorwiegend auf den Durchsatz des jeweiligen Kanals. Jedem Kanal wird zusätzlich mit Hilfe der `applTransactionStreamTable` ein Transaktionsstrom zugeordnet. Für jeden Transaktionsstrom kann eine Beschreibung der Transaktionen, die darüber ausgeführt werden, durch Angabe der *Unit of Work* des Transaktionsstroms angegeben werden. Mit Hilfe von Zählern werden Zeiten und Häufigkeiten von Transaktionen festgehalten. Nach Schließung eines Kanals wird die zugehörige Information zur späteren Verwendung in der `applPastChannelGroup` gespeichert. Das Starten und Stoppen der Anwendung sowie die Aufforderung zum Neueinlesen der Konfigurationsdateien kann mit Hilfe der `applElmtRunControlGroup` durchgeführt werden.

Die *Application Management MIB* greift zwar auf Instrumentierung der Anwendung zurück und enthält insbesondere auch Informationen über Transaktionen der Anwendung, sie ist aber dennoch nicht geeignet, den gestellten Anforderungen gerecht zu werden. Das liegt zunächst an der Beschränkung auf ein einzelnes System, die im Umfeld des *Application Service Provisioning* nicht gegeben ist. Da die Transaktionen im Rahmen von Kanälen definiert werden, ist es nicht möglich, Benutzertransaktionen zu definieren. Diese erfolgen typischerweise nicht durch einen Netz- oder Dateizugriff, sondern durch eine Benutzerinteraktion an der Oberfläche der Anwendung. Desweiteren läßt sich festhalten, daß die *Application Management MIB* zwar gute Ansätze bzgl. der Definition von erforderlicher Managementinformation im Rahmen der Transaktionsüberwachung liefert, allerdings, wie heute im Bereich der Standardisierung üblich, keinerlei Aussage darüber trifft, wie eine entsprechende Managementinstrumentierung aussehen könnte. Stattdessen beschränkt man sich auf einen Verweis auf die *ARM API*, die in Abschnitt 4.2.1.3 noch detailliert diskutiert wird.

Als eine Erweiterung der *Application Management MIB* läßt sich die *„WWW Service MIB“* [HKS 99] betrachten, die sie um spezifische Informationen, die für das Management von WWW Diensten benötigt werden, erweitert. Die *Application Management MIB* ist allerdings nicht Voraussetzung für eine Implementierung der *„WWW Service MIB“*. Da sie nur spezifische Informationen (z.B. Anzahl von Zugriffen auf bestimmte Webseiten) enthält, kann auf eine weitere Betrachtung verzichtet werden.

Application Perfomance Measurement MIB

Die *Application Performance Measurement MIB (APM MIB)* befindet sich aktuell noch im Stadium eines *Internet Drafts* [Wald 01]. Sie stellt eine Erweiterung der RMON MIB [Wald 95] bzw. der RMON2 MIB [Wald 97] dar. Mit Hilfe der RMON MIBs wurde es möglich, die Ressourcen eines Netzes zu überwachen, ohne daß eine ständige Erreichbarkeit des Netzes durch die Managementstation erforderlich ist. Die RMON MIB kann auf den Agentensystemen selbständig Daten sammeln, vorverarbeiten und auf Anfrage einer Managementstation mittels SNMP zur Verfügung stellen.

Durch Platzierung auf einem Agentensystem in der *Client*-Domäne, ist es der APM MIB möglich, Transaktionen aus Nutzersicht zu überwachen. Die `APM Application Directory Group` beschreibt die Anwendungsprotokolle, die überwacht werden können (z.B. HTTP, FTP, RealVideo). Für jedes Protokoll ist es möglich, eine von drei Arten der Antwortzeitüberwachung anzugeben. Im einzelnen sind dies die Transaktionsorientierung, die Durchsatzorientierung und die Stromorientierung. Für jede der drei Varianten ist dann eine Metrik für die Antwortzeitüberwachung vorgegeben. Im Falle der Transaktionsorientierung ist dies die Zeit vom Zeitpunkt der Anfrage bis zum Ende der Transaktion, im Falle der Durchsatzorientierung der erreichte Durchsatz (in KBit/s) und im Falle der Stromorientierung ist es der Prozentsatz der Zeit, zu dem nicht die vereinbarte Übertragungsrate eingehalten werden konnte. Weiterhin können Schwellwerte für das Antwortzeitverhalten definiert werden.

Die zweite Gruppe, die `APM Report Group` liefert statistische Informationen über die erfolgten Transaktionen. Beispiele hierfür sind die Anzahl erfolgreicher Transaktionen, die Gesamtanzahl an Transaktionen sowie das durchschnittliche Antwortzeitverhalten aller Transaktionen einer Art. Darüber hinaus liefert die `APM Current Transaction Group` Information über die aktuell in Ausführung befindlichen Transaktionen, während die `APM Exception Group` sowie die `APM Notification Group` dazu dienen, Grenzwerte für Transaktionen festzulegen (z.B. bzgl. der gemessenen Antwortzeiten) sowie im Falle einer Grenzwertüberschreitung die entsprechende Managementstation zu informieren.

Ähnlich der *Application Management MIB* stellt die APM MIB einen Schritt in die richtige Richtung dar. Sie ermöglicht die Definition von Transaktionen aus Benutzersicht, die Festlegung von Schwellwerten und die statistische Aufbereitung der ermittelten Information. Die Frage, wie die entsprechenden Informationen zu ermitteln sind, wird jedoch weiterhin offen gelassen.

Application Configuration Access Protocol Working Group

Die *Application Configuration Access Protocol Working Group (ACAP WG)* der IETF beschäftigt sich mit Themen der einheitlichen Konfiguration von Anwendungen unabhängig davon, wo sich der Nutzer aktuell befindet. Ausgehend von der Beobachtung, daß Nutzer zunehmend den Wunsch haben, Anwendungen von unterschiedlichen Orten aus zu nutzen, dort jeweils aber identische Gegebenheiten vorfinden möchten, wurde das *Application Configuration Access Protocol (ACAP)* (RFC 2244) [NeMy 97] definiert. Es ermöglicht entfernten Zugriff auf die Konfigurationsdaten der Anwendung einschließlich der benutzerspezifischen Einstellungen.

Da der Fokus der *ACAP WG* ausschließlich im Bereich der Benutzeradministration liegt, kann auf eine weitere Untersuchung der in diesem Umfeld entstandenen Spezifikationen verzichtet werden.

Realtime Traffic Flow Measurement Working Group

Die mittlerweile wieder aufgelöste *Realtime Traffic Flow Measurement Working Group (RTFM WG)* beschäftigte sich mit der Überwachung sogenannter *Traffic Flows* [BMR 97]. Ein *Traffic Flow* bezeichnet in irgendeiner Form zusammengehörigen Verkehr, beispielsweise zwischen einem bestimmten Paar von Endgeräten. Anhand von Regeln können einzelne Pakete einem bestimmten *Flow* zugeordnet werden. Allerdings stellt die Ermittlung dieser Regeln eine sehr komplexe Aufgabe dar, für die bisher keine allgemeine Methodik angegeben werden konnte. Abgesehen von einigen Standarddiensten, für die bereits Regeln existieren, bleibt es somit den einzelnen Systemadministratoren überlassen, für ihre spezifischen Dienste manuell geeignete Abbildungen zu definieren.

Die eigentliche Motivation für die Überwachung von *Traffic Flows* liegt im Bereich der Netzplanung und Bewertung der existierenden Netzinfrastruktur. Es ist jedoch angedacht, die Definition eines *Flows* auszuweiten auf *Flows* der Anwendungsebene. Somit wäre es möglich, das Antwortzeitverhalten von Anwendungen zu ermitteln. Hierzu finden sich jedoch derzeit keinerlei Spezifikationen. Eine echte Überwachung von nutzerorientierten Parametern mit Hilfe von *Flows* ist dabei aber ohnehin nicht zu erwarten.

4.2.1.2 Distributed Management Task Force

Die *Distributed Management Task Force (DMTF)* wurde im Jahre 1992 (unter dem Namen *Desktop Management Task Force*) gegründet, um Standards für das Management von PC Systemen zu schaffen. Es handelt sich um einen Zusammenschluß von mehr als 130 Industrieunternehmen unter der Führung von Unternehmen wie z.B. *Cisco*, *Hewlett-Packard*, *Microsoft* oder *Sun Microsystems* [DMTF 00].

Im Rahmen dieser Aktivitäten wurde das *Distributed Management Interface (DMI)* [DMI 2.0s] (früher *Desktop Management Interface*) entwickelt, das zwei unterschiedliche Schnittstellen definiert. Zum Zugriff auf Managementinformation zwischen Managementanwendung und Agent (dem sogenannten *Service Provider*), dient das sogenannte *Management Interface*; zwischen Agent und zu überwachender Komponente befindet sich das *Component Interface*. Die Beschreibung der zu überwachenden Komponenten erfolgt in *Management Information Format (MIF)* Dateien. Im Rahmen des Anwendungsmanagements wurde vom *Software Working Commitee* eine MIF Datei zur Beschreibung von

Anwendungen entworfen, die *Software Standard Groups Definition* [SWMIF 2]. Diese befaßt sich allerdings ausschließlich mit Aufgaben der Bestandsführung und ist daher für diese Arbeit nur von geringer Bedeutung. Sie diente allerdings als Ausgangspunkt für viele weiterführende Arbeiten, wie das *Application Management Model* der DMTF (siehe unten) oder die *Application Management Specification (AMS)* der Firma Tivoli (vgl. Abschnitt 4.2.3.1).

Common Information Model
Die einheitliche Beschreibung von Managementinformation mit Hilfe des *Common Information Model (CIM)* steht heute im Vordergrund der Aktivitäten der DMTF. Das CIM ist ein objektorientiertes Informationsmodell, das, unabhängig von einer konkreten Implementierung, die managementrelevante Information unterschiedlicher Bereiche beschreibt.

Die Beschreibung der Information kann grafisch erfolgen. Es existiert ein Meta-Schema [CIM 2.2], das auf der UML [OMG 00-03-01] basiert und Begriffe wie *Class*, *Property*, *Association* oder *Qualifier* einführt und festlegt. Ergänzend existiert eine textuelle Beschreibungssprache, das *Managed Object Format (MOF)*, das auf der DCE IDL basiert.

Die in CIM modellierte Information ist in drei Schichten aufgeteilt:

- Core Model
- Common Model
- Extension Schemas

Das *Core Model* [CoreMOF 25] [Core 24] beschreibt die Information, die in vielen Managementbereichen gleichermaßen benötigt wird. Beispiele hierfür sind *ManagedElement*, *Product* oder *LogicalElement* sowie die Abhängigkeiten zwischen den Entitäten. Es dient somit als Ausgangspunkt für jegliche Untersuchung und Festlegung von Managementinformation. Die *Common Models* erweitern die Information des *Core Models* um Informationen spezifischer Bereiche. Die derzeit modellierten Bereiche sind

- System
- Application
- Network
- Device
- Physical
- Events
- Metrics
- Policy
- Support

Die *Common Models* beschreiben die Bereiche unabhängig von einer konkreten Technologie oder Implementierung. Dies kann im Rahmen des *Extension Schemas* durch weitere Verfeinerung der Information geschehen.

CIM Application Model

Der für die vorliegende Arbeit wichtigste Bereich des *Common Models* ist das *Application Management Model* [ApplMOF 25] [Appl 24]. Es modelliert und beschreibt managementrelevante Information verteilter Anwendungen. Hierzu wurde eine Vorgehensweise entlang des Lebenszyklus einer Anwendung gewählt, der wie folgt definiert ist:

- verteilbarer Zustand (*deployable state*)
- installierbarer Zustand (*installable state*)
- ausführbarer Zustand (*executable state*)
- laufender Zustand (*running state*)

Die im jeweiligen Zustand benötigten Managementinformationen können ebenso beschrieben werden wie die Übergänge zwischen den Zuständen. Hierzu können Bedingungen angegeben werden, die erfüllt sein müssen, bevor eine Anwendung in den nächsten Zustand übergehen kann, sowie Aktionen definiert werden, die den Übergang in den nächsten Zustand durchführen.

Die beschriebene Managementinformation beschränkt sich jedoch ausschließlich auf den Bereich der Bestandsführung und der Softwareverteilung und -installation. Im Bereich der Anwendungsüberwachung kann das *Application Management Model* somit nicht zum Einsatz kommen.

Distributed Application Performance Schema

Die *Distributed Application Performance Working Group (DAP WG)* der DMTF definiert aktuell ein *Common Model*, das das Standard-Laufzeitverhalten von verteilten Anwendungen beschreibt [DapMOF 24]. Hierzu wird eine einheitliche Definition einer *Unit of Work (UoW)* benötigt. Beispiele für UoWs sind Stapelaufträge, Datenbankabfragen und E/A-Operationen, aber auch Benutzertransaktionen wie in 2.2.2 definiert. Die DAP WG wurde kürzlich in *Metrics WG* umbenannt, in der die Bemühungen zukünftig vorangetrieben werden.

Abbildung 4.3 stellt den aktuellen Status der Standardisierungsbemühungen dar. Es wird zwischen einer *UnitOfWork* und der zugehörigen *UnitOfWorkDefinition* unterschieden.

- UnitOfWorkDefinition
  Bei der *UnitOfWorkDefinition* handelt es sich um ein logisches Element, das die UoW eindeutig beschreibt. Hierzu enthält sie einen Namen (`Name`), einen Kontext

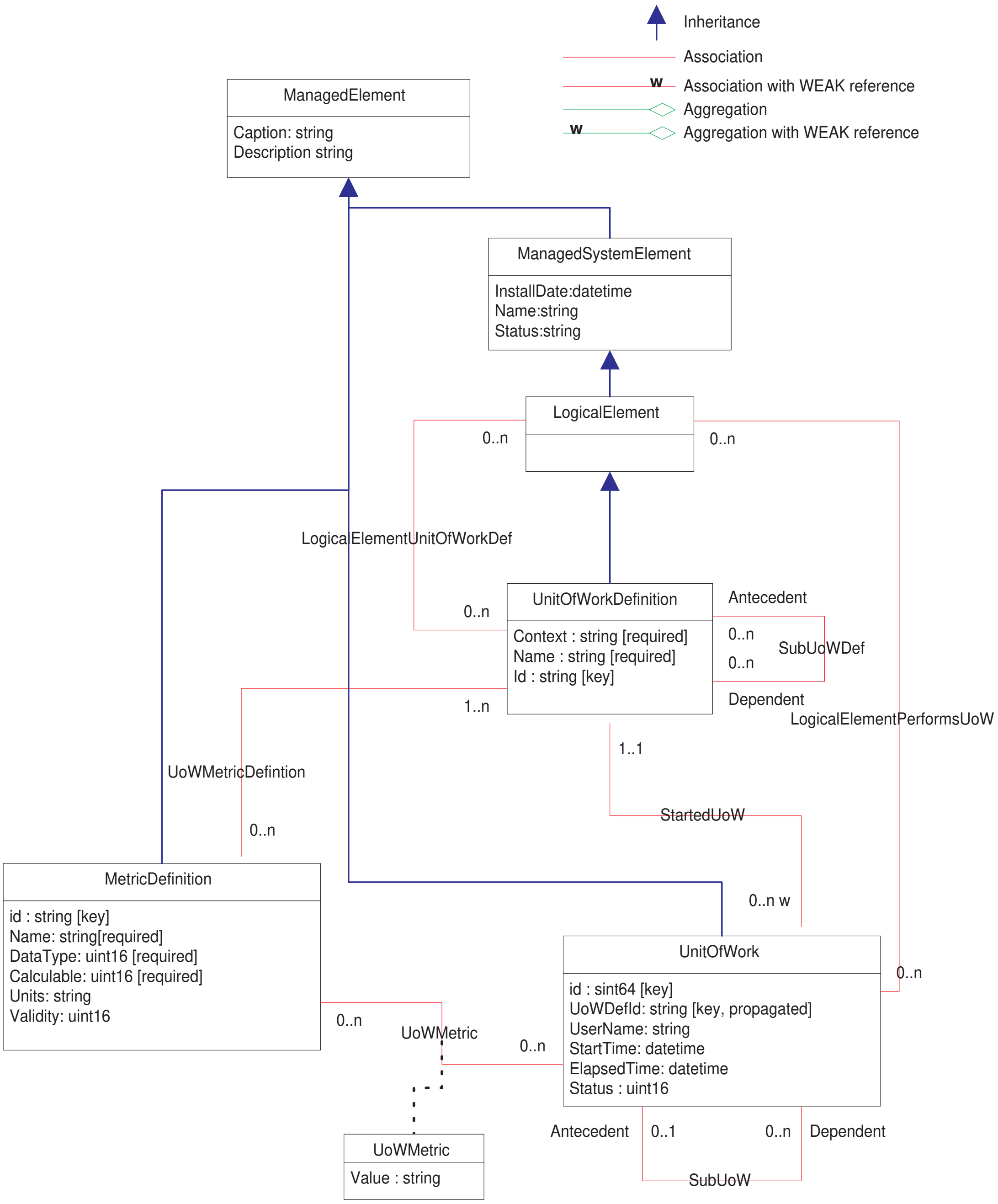

Abbildung 4.3: CIM Distributed Application Performance Schema 2.4 [Dap 24]

(`context`), der eine natürlichsprachige Beschreibung der Transaktion enthält sowie einen eindeutigen Identifikator (`ID`). Jedem *LogicalElement* können die UoWs, die es erbringen kann mit Hilfe der *LogicalElementUnitOfWorkDef*-Assoziation zugeordnet werden.

- UnitOfWork
  Jede Instanz der Klasse *UnitOfWork* beschreibt eine tatsächliche Transaktion, die entweder aktuell ausgeführt wird oder bereits beendet ist. Eine Instanz einer UoW wird eindeutig identifiziert über die zugehörige *UnitOfWorkDefinition* (`UoWDefId`) und einen eindeutigen Identifikator (`Id`). Die Parameter, die die UoW näher beschreiben sind

  - StartTime:
    Die Zeit, zu der die UoW begonnen hat.
  - ElapsedTime:
    Die Zeit, die seit Beginn der UoW vergangen ist. Im Falle einer bereits abgeschlossenen UoW die Antwortzeit der UoW.
  - UserName:
    Der Name des Benutzers, der die UoW angestoßen hat.
  - Status:
    Information darüber, ob die UoW aktuell noch in Ausführung befindlich ist oder bereits abgeschlossen ist. Im Falle bereits abgeschlossener UoWs zusätzlich Information über die erfolgreiche bzw. nicht erfolgreiche Beendigung der UoW [DeSo 97].

  Das *LogicalElement*, das die UoW ausführt, wird über die *LogicalElementPerformsUoW*-Assoziation identifiziert.

- MetricDefinition
  Eine *MetricDefinition* gestattet es, weitere Metriken mit einer UoW zu assoziieren. Diese werden unter anderem beschrieben über ihren Namen (`Name`), den Datentyp (`DataType`) und ihre Einheit (`Units`).

- UoWMetric
  Mit Hilfe der Assoziation *UoWMetric* werden Metriken bestimmten UoWs zugeordnet und mit einem tatsächlichen Wert (`Value`) versehen.

Mit Hilfe der Assoziationen *SubUoWDef* und *SubUoW* ist es möglich, Subtransaktionen zu definieren und mit ihren übergeordneten Transaktionen zu korrelieren.

Das *Distributed Application Performance Schema* (bzw. *Metrics Schema*) der DMTF stellt einen Mechanismus für die einheitliche Beschreibung von Transaktionen zur Verfügung. Hersteller sind somit in der Lage, auf diesem Informationsmodell aufbauend

Werkzeuge zu entwickeln, die die Überwachung und Verarbeitung der Information über Transaktionen durchführen. Es handelt sich jedoch (ähnlich wie bei der *Application Performance Measurement MIB* (vgl. Abschnitt 4.2.1.1) der IETF) ausschließlich um die Definition der benötigten, generischen Information. Es wird nicht darauf eingegangen, wie die benötigte Information aus den Anwendungen zu ermitteln ist.

4.2.1.3 Open Group

Die *Open Group* ist ein herstellerunabhängiges, internationales Konsortium mit derzeit 228 Mitgliedern. Der größte Teil der Mitglieder sind Unternehmen aus dem Informations- und Kommunikationsbereich (z.B. *Hewlett-Packard*, *IBM*, *Intel*, *Microsoft*, etc.). Darüber hinaus finden sich aber auch Unternehmen aus anderen Bereichen (z.B. *Toyota*, *Chase Manhattan Bank*, etc.) sowie öffentliche Behörden (z.B. *US Department of Defense*, etc.). Das zentrale Ziel der *Open Group* ist nicht nur die Schaffung und Bereitstellung von Standards, sondern insbesondere auch die Zertifizierung der Standardkonformität von Produkten.

Die *Open Group* wurde im Jahre 1996 durch den Zusammenschluß der *X/Open Company Ltd.* und der *Open Software Foundation (OSF)* gegründet. Unter anderem ist sie verantwortlich für das `UNIX`-Warenzeichen. Im Bereich des Anwendungsmanagements und insbesondere im Bereich der Anwendungsüberwachung standardisiert sie aktuell zwei unterschiedliche Instrumentierungsansätze, die *Application Response Measurement API (ARM API)* und die *Application Instrumentation & Control API (AIC API).*

Application Response Measurement API

Die *Application Response Measurement API (ARM API)* wurde ursprünglich (Version 1.0) im Jahre 1996 in einer gemeinsamen Initiative der Firmen *Hewlett Packard* und *Tivoli* entwickelt. Die Nachfolgeversionen 2.0 und 3.0 wurden von der *ARM Working Group (ARM WG)* der *Computer Measurement Group (CMG)* entwickelt. Die ARM WG der CMG ist ein Zusammenschluß von 17 Firmen unter der Leitung von *Hewlett-Packard* und *Tivoli*. 1998 wurde die Version 2.0 der ARM API von der *Open Group* im Rahmen der *IT Dialtone* Initiative als technischer Standard [C807] übernommen. Aktuell ist Version 3.0 der ARM API verfügbar, die allerdings zum heutigen Zeitpunkt noch nicht als offizieller Standard übernommen wurde. Aus diesem Grund wird im folgenden zunächst die Version 2.0 vorgestellt und im Anschluß daran ein Überblick über die wesentlichen Erweiterungen der Version 3.0 gegeben. Die Arbeiten zur ARM API der ARM WG und die Arbeiten der *DAP/Metrics WG* der DMTF (siehe Abschnitt 4.2.1.2) ergänzen sich und werden gemeinsam vorangetrieben [John 99].

Die ARM API erlaubt die transaktionsbasierte Überwachung des Antwortzeitverhaltens verteilter Anwendungen auch in heterogenen Umgebungen. Um dieses Ziel zu erreichen wurde eine sehr einfache API definiert. Vor Beginn und nach Ende jeder (vom Anwendungsentwickler festzulegenden) zu überwachenden Transaktion müssen Aufrufe der entsprechenden Funktionen der API in den *Source Code* der Anwendung eingefügt werden. Die Implementierung der Funktionen der API erfolgt in sogenannten *Measurement Agents*, die zur zu überwachenden Anwendung gebunden werden. Diese bestimmen und speichern den Zeitpunkt der Aufrufe und gestatten so die Berechnung der Antwortzeiten einzelner Transaktionen.

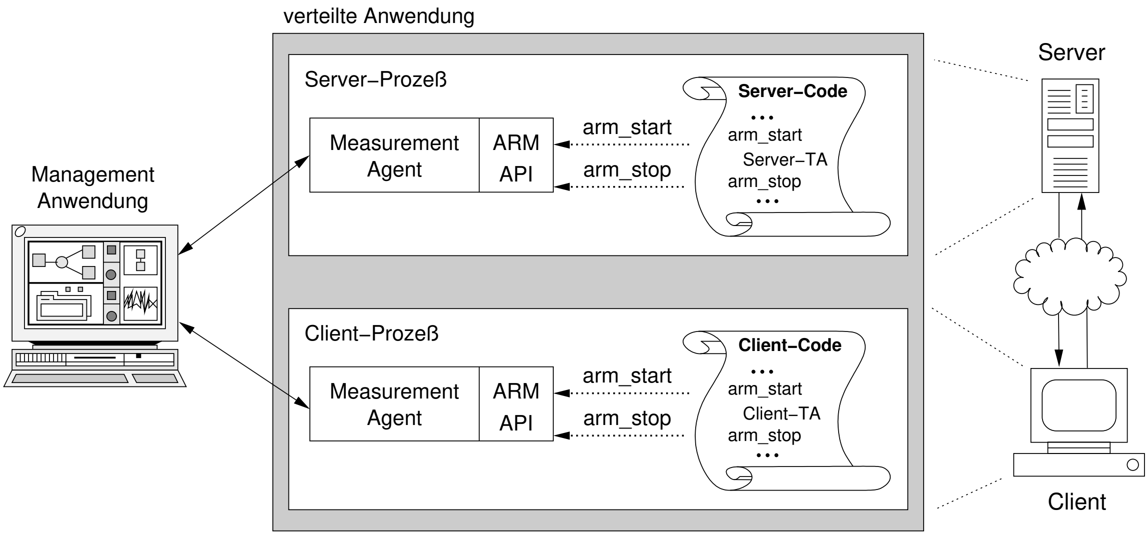

Abbildung 4.4: ARM API: Architektur

Abbildung 4.4 stellt die resultierende Architektur dar: Eine – möglicherweise verteilt realisierte – Anwendung wird instrumentiert, vor Beginn und nach Ende jeder Transaktion eine Funktion der ARM API aufzurufen [HaRe 00]. Ein auf dem jeweiligen System vorhandener *Measurement Agent* wird dynamisch zur Anwendung gebunden und wird somit im selben Prozeß wie die zu überwachende Anwendung ausgeführt. Er nimmt die Aufrufe entgegen, bestimmt die aktuelle Systemzeit und leitet die Information an beliebige Managementanwendungen weiter. Die Schnittstelle zwischen Managementanwendung und *Measurement Agent* ist hierbei allerdings nicht vorgegeben, sondern bleibt der Implementierung des Managementsystems überlassen. Aus Gründen der Performanz des resultierenden Systems wird allerdings vorgeschlagen, daß der *Measurement Agent* nur teilweise innerhalb des Prozesses der zu überwachenden Anwendung abläuft und der Hauptteil des Agenten (z.B. die Kommunikation mit der eigentlichen Managementanwendung) außerhalb der Anwendung stattfindet [John 97]. Damit instrumentierte Anwendungen auch dann lauffähig sind, wenn kein *Measurement Agent* auf dem System vorhanden ist, kann die sogenannte *Null Library* zur Anwendung gebunden werden. Diese implementiert die ARM API, verfügt aber über keinerlei Funktionalität, sondern kehrt unmittelbar zur aufrufenden Anwendung zurück.

In Version 2.0 der ARM API sind die folgenden sechs Funktionen definiert:

- `arm_init`
  Wird während der Initialisierung der zu überwachenden Anwendung (bzw. einer Instanz einer Anwendung) aufgerufen und dient dazu, die Anwendung eindeutig zu benennen und den aktuellen Benutzer zu identifizieren. Diese Funktion liefert einen eindeutigen Identifikator für die Instanz der Anwendung.

- `arm_getid`
  Dient dazu, die unterschiedlichen Klassen von Transaktionen, die innerhalb der Anwendung auftreten können, zu definieren und einen eindeutigen Identifikator für jede Transaktionsklasse zu generieren. Für jede Klasse läßt sich optional angeben, welche zusätzlichen Informationen in den nachfolgenden Aufrufen zu liefern sind.

- `arm_start`
  Dieser Aufruf wird in den *Code* der Anwendung eingefügt, unmittelbar bevor eine zu überwachende Transaktion beginnt. Die Klasse der Transaktion wird über den mittels `arm_getid` ermittelten Identifikator übergeben. Als Rückgabewert wird ein eindeutiger Identifikator der Transaktionsinstanz zurückgegeben. Weitere Daten, die den aktuellen Status der Anwendung beschreiben, können – wie mittels `arm_getid` beschrieben – als Parameter mit übergeben werden.

- `arm_stop`
  Ebenso, wie zum Start einer Transaktion die Funktion `arm_start` aufgerufen wird, wird zu deren Ende die Funktion `arm_stop` aufgerufen. Wiederum können zusätzliche Informationen, insbesondere über den Erfolg oder Mißerfolg der Transaktion übergeben werden. Die Identifikation der Instanz der Transaktion erfolgt über den vom `arm_start` Aufruf zurückgelieferten Identifikator.

- `arm_update`
  Dieser Aufruf kann beliebig oft zwischen `arm_start` und `arm_stop` Aufrufen verwendet werden, um zusätzliche Informationen zu übermitteln oder z.B. im Falle sehr lang dauernder Transaktionen zwischenzeitlich einen Status zu übergeben und mitzuteilen, daß die Transaktion weiterhin korrekt ausgeführt wird.

- `arm_end`
  Wenn die zu überwachende Anwendung beendet wird, kann dies dem *Measurement Agent* mit Hilfe von `arm_end` übermittelt werden, der daraufhin die belegten Ressourcen freigeben kann.

Seit Einführung von Version 2.0 gestattet die ARM API die Korrelation von Subtransaktionen zu übergeordneten Transaktionen. Dies kann über mehrere Systeme verteilt geschehen. Hierzu werden sogenannte Korrelatoren (*correlators*) eingesetzt. Beim Aufruf von `arm_start` kann die Anwendung vom *Measurement Agent* einen Korrelator anfordern, der bei nachfolgenden Aufrufen von `arm_start` wieder übergeben werden kann. Diese nachfolgenden Aufrufe werden dann vom *Measurement Agent* als Beginn von Subtransaktionen der ursprünglichen Transaktion betrachtet. Der Korrelator muß hierzu natürlich innerhalb der zu überwachenden Anwendung (z.B. als zusätzlicher Aufrufparameter) propagiert werden. Eine detailliertere Betrachtung der Korrelation von Subtransaktionen bei Verwendung der ARM API findet sich in Abschnitt 5.3.3.2.

Wie bereits erwähnt, ist mittlerweile Version 3.0 der ARM API bei der CMG verfügbar [John 99][Smea 99], allerdings noch nicht offiziell von der *Open Group* standardisiert worden. Die wesentlichen Erweiterungen sind die Unterstützung von *Java* sowie die Möglichkeit, komplette Transaktionen an den *Measurement Agent* zu übermitteln. Die bisherigen Versionen der ARM API beschränkten sich auf die Programmiersprache C, gingen aber davon aus, daß C-Bibliotheken aus nahezu jeder anderen Programmiersprache aufrufbar sind. Aus *Java* heraus besteht diese Möglichkeit mit Hilfe des *Java Native Interface (JNI)*. Da dies aber umständlich ist und Probleme bzgl. der Performanz bereitet und *Java* heutzutage immer weitere Verbreitung findet, wurde eine *Java*-Bibliothek definiert, die direkt in *Java*-Programme eingebunden und aus diesen heraus aufgerufen werden kann. Aufgrund der Objektorientiertheit von *Java* wurden einige Anpassungen erforderlich, die aber keine wesentlichen konzeptionellen Unterschiede darstellen. Wie schon erwähnt, ist darüber hinaus ein Aufruf eingeführt worden, der es Anwendungen gestattet, komplette Transaktionen an den *Measurement Agent* zu übermitteln. Dies wurde im Hinblick auf Anwendungen gemacht, die bereits über – nicht ARM-konforme – Instrumentierung verfügen und somit mit geringem Aufwand in die ARM-basierte Überwachung integriert werden können.

Trotz vieler Unzulänglichkeiten ist die ARM API die derzeit am weitesten verbreitete Methode für die generische Instrumentierung von Anwendungen zur Transaktionsüberwachung. Mehrere Hersteller von Managementsystemen (z.B. *Hewlett Packard*, *Tivoli*) bieten *Measurement Agents* an, die eine Überwachung von instrumentierten Anwendungen erlauben. Die ARM API stellt somit einen wichtigen ersten Schritt für die generische Managementinstrumentierung verteilter Anwendungen dar. Aus diesen Gründen wurde sie im Rahmen der Studien zur vorliegenden Arbeit einer eingehenden Untersuchung unterzogen [HaRe 99][Fisc 01][Alde 00][Hojn 99].

Dabei zeigte sich, daß zum heutigen Zeitpunkt keine instrumentierten Anwendungen verfügbar sind. Dies liegt insbesondere daran, daß trotz der sehr einfach gehaltenen Struktur der ARM API die Instrumentierung von Anwendungen erheblichen Aufwand verursacht. Insbesondere die nachträgliche Instrumentierung von Anwendungen ist – selbst wenn Zugang zum *Source Code* der Anwendung existiert – nur mit großer Mühe möglich. Auch bereitet die Korrelation von Subtransaktionen zu übergeordneten Transaktionen erhebliche Probleme. Dies liegt daran, daß der vom *Measurement Agent* gelieferte Korrelator innerhalb der zu überwachenden Anwendung propagiert werden muß, so daß er beim Start von Subtransaktionen übergeben werden kann. Dies bedeutet, daß z.B. bei nachträglicher Instrumentierung einer Anwendung die Aufrufparameter einzelner Funktionen der Anwendung verändert werden müssen, um die Propagierung der Korrelatoren zu ermöglichen. Dies macht sich insbesondere im Fall über mehrere Systeme verteilter Anwendungen negativ bemerkbar. Ein detaillierter Vergleich der ARM API mit der im folgenden Kapitel vorgeschlagenen Lösung findet sich in Abschnitt 5.5.1

Application Instrumentation & Control API

Die *Application Instrumentation & Control API (AIC API)* wurde gemeinsam von *J.P. Morgan* und *Computer Associates* entwickelt und Ende 1999 von der *Open Group* als *Technical Standard* übernommen [C910]. Sie besteht aus zwei Schnittstellen, die von entsprechenden Bibliotheken implementiert werden müssen (*Application Library* und *Client Library*) und die es Anwendungen gestatten, Informationen auf einfache Art und Weise für Managementsysteme zur Verfügung zu stellen. Außerdem spezifiziert sie einige einfache Basisdienste, die sogenannten *Host Services*.

Die grundsätzliche Funktionsweise der AIC API ist in Abbildung 4.5 skizziert. Eine Anwendung nutzt die *Application Library (AL)*, um Managementobjekte zu generieren und mit Werten zu belegen. Diese können von Managementanwendungen mit Hilfe der *Client Library (CL)* abgefragt und verändert werden. Die *Host Services* sind Basisdienste, die z.B. für die sichere Kommunikation zwischen den beiden Bibliotheken und für die Kommunikation mit anderen Managementarchitekturen verantwortlich sind.

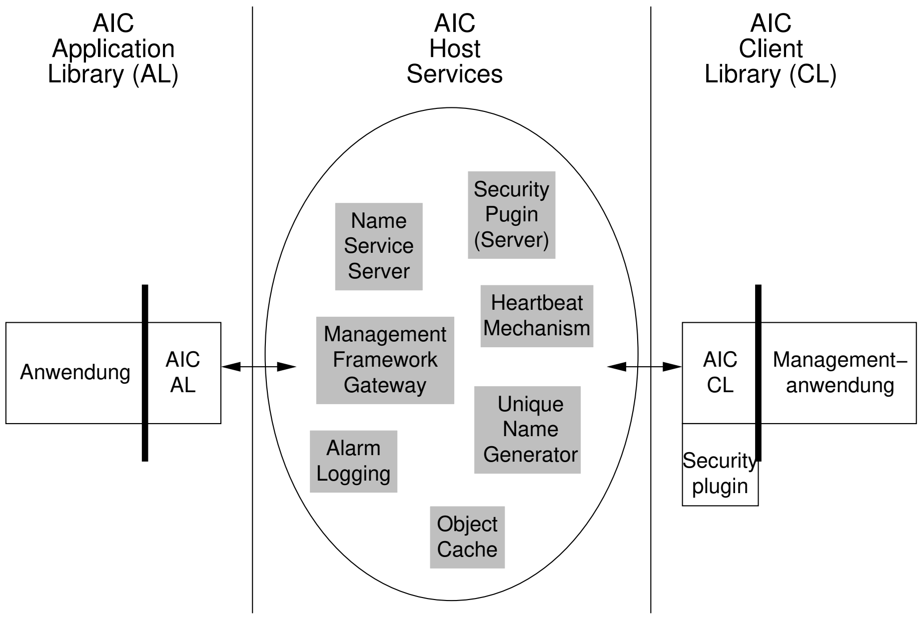

Abbildung 4.5: Funktionsweise der AIC API (aus [C910])

Der Anwendungsentwickler muß zum Zeitpunkt der Anwendungserstellung entscheiden, welche Information der Anwendung von Bedeutung für ein Managementsystem sein könnte. Er generiert die entsprechenden Managementobjekte (MOs) über die AL. Die Belegung der MOs mit Werten kann auf zweierlei Art geschehen: Entweder, der Anwendungsentwickler fügt explizite Aufrufe der AL in den Code der Anwendung ein, die den Wert des entsprechenden MOs aktualisieren, oder er registriert eine sogenannte *Polling Function* für das MO, die automatisch von der AL in regelmäßigen Abständen aufgerufen wird und für die Aktualisierung des Wertes sorgt. Für die Implementierung der *Polling Function* ist wiederum der Anwendungsentwickler verantwortlich.

Der Zugriff auf die MOs erfolgt dann, wie bereits erwähnt, durch Aufrufe der CL. Außerdem kann für jedes MO eine *Threshold Function* registriert werden, die ähnlich wie die *Polling Function* regelmäßig aufgerufen wird. Diese dient der Überwachung der MOs auf Überschreitung definierbarer Grenzwerte und muß ebenfalls vom Anwendungsentwickler implementiert werden. Im Falle einer Grenzwertüberschreitung kann die *Threshold Function* beispielsweise – wiederum durch Aufruf der AL – ein *Event* generieren, das die Information an die angeschlossenen Managementsysteme weitergibt.

Die AIC API erlaubt nicht nur die bloße Überwachung von Anwendungen, sondern darüber hinaus auch steuernde Eingriffe. Dies kann entweder über die explizite Belegung von MOs mit Werten über die CL erfolgen oder aber durch den Aufruf der sogenannten *DoAction Function*. Hierbei handelt es sich um eine beliebige, vom Anwendungsentwickler zu implementierende Funktion, die bei der AL registriert und über die CL aktiviert werden kann.

Neben den bereits erwähnten Beispielen für *Host Services*, die die sichere Kommunikation zwischen den Bibliotheken gewährleisten und die Anbindung an weitere Managementarchitekturen ermöglichen, existieren noch weitere Basisdienste: Bei einem Namensdienst (*Name Service*) lassen sich beispielsweise alle auf dem jeweiligen System überwachten Anwendungen erfragen. Für die regelmäßige Überwachung der Anwendungen auf Totalausfall existiert ein *Heartbeat Mechanism*, der in regelmäßigen Abständen überprüft, ob die jeweilige AL noch auf Anfragen reagiert. Ein *Object Cache* erhöht die Leistung des Systems, indem er Anfragen beantwortet, ohne die eigentliche Anwendung kontaktieren zu müssen.

Die AIC API ermöglicht es einem Anwendungsentwickler, Managementinformation aus der Anwendung mit Hilfe eines generischen Mechanismus Managementsystemen zur Verfügung zu stellen. Grundätzlich ist sie somit geeignet, die in Kapitel 3 geforderte Managementinformation zu liefern. Der Anwendungsentwickler erhält allerdings weder Hilfestellung in der Auswahl der benötigten Information noch bei der eigentlichen Instrumentierung der Anwendung. Ihm wird lediglich ein Mechanismus zur Verfügung gestellt, der die Kommunikation mit den eigentlichen Managementsystemen vor ihm verschattet. Der resultierende Instrumentierungsaufwand ist daher dennoch als zu hoch zu betrachten und wird die weitere Verbreitung der AIC API negativ beeinflussen.

4.2.1.4 TeleManagement Forum

Das *TeleManagement Forum (TMForum)* wurde 1988 (unter dem Namen *Network Management Forum (NMF)*) gegründet und beschäftigt sich mit Fragen des Betriebs und des Managements von Telekommunikationsdiensten und -infrastrukturen. Es besteht derzeit aus über 250 Unternehmen der Informations- und Kommunikationstechnik. Aktuell liegt ein wesentlicher Schwerpunkt der Arbeit des TMForum in der Automation der Geschäftsprozesse von Unternehmen des Telekommunikationsbereichs. Die wesentlichen Ergebnisse dieser Arbeit finden sich in der *Telecom Operations Map (TOM)* [TOM1.1], in der die Prozesse von Telekommunikationsunternehmen einschließlich ihrer Subprozesse identifiziert und beschrieben werden. Im Rahmen der *Technology Integration Map (TIM)* [TMF 909] werden aktuell diejenigen Technologien identifiziert, die für die Automation der Prozesse geeignet erscheinen. Die TIM wird von mehreren unterschiedlichen Projektteams erstellt. Die Arbeit des *Application Components Team (ACT)* besitzt für die vorliegende Arbeit die größte Relevanz und wird daher im folgenden Abschnitt detaillierter vorgestellt.

Application Components Team

Der vom *Application Components Team (ACT)* des TMForum entwickelte erste Teil der TIM liefert Anforderungen, die an sogenannte *Telecommunications Management Building Blocks*, also Bausteine, die für die Automation der Geschäftsprozesse von Telekommunikationsunternehmen eingesetzt werden sollen, zu stellen sind [OMG 00-06-02]. Ein Baustein ist hierbei als die Einheit der Softwareverteilung definiert, stellt gleichzeitig aber auch die zu managende Einheit dar. Somit werden also auch Anforderungen an eine Managementschnittstelle ermittelt, um das Management von Bausteinen unterschiedlicher Hersteller mit generischen Werkzeugen zu gestatten. Diese sind den im Rahmen der vorliegenden Arbeit erstellten Anforderungen gegenüberzustellen.

Das ACT trennt die Schnittstellen (*contracts*) eines Bausteins in Schnittstellen für die eigentliche Dienstnutzung (*service contracts*) und Schnittstellen für das Management des erbrachten Dienstes (*management contracts*). Der *management contract* eines Bausteins ist bausteinspezifisch und wird im Rahmen der TIM nicht näher spezifiziert. Darüber hinaus existiert eine Schnittstelle, über die generische Managementfunktionalität und -information, die für alle Bausteine identisch bereitgestellt werden kann, verfügbar ist (*Common Application Management Interface (CAMI)*). Über die CAMI läßt sich beispielsweise der Ressourcenverbrauch des Bausteins oder die Anzahl an verbundenen *Clients* ermitteln. Außerdem löst ein Baustein über die CAMI einen Alarm aus, wenn er eine Fehlersituation feststellt (entweder im Baustein selber oder aber in einer anderen Komponente wie z.B. dem Netz, dem darunterliegenden System oder einem anderen Baustein). Durch die Angabe von Abhängigkeiten wird ein Manager in die Lage versetzt, zu ermitteln, welche Bausteine vorhanden sein müssen, um den Betrieb eines Bausteins zu ermöglichen.

Die CAMI spezifiziert keine Managementschnittstelle, sondern gibt Anforderungen an, die an eine Managementschnittstelle für Bausteine zu stellen sind. Daher müssen wesentliche Details naturgemäß offen bleiben. So wird zwar spezifiziert, daß jeder Baustein die oben genannten Informationen zu liefern hat (Ressourcenverbrauch, Anzahl verbundener *Clients*, erkannte Fehlerzustände, Abhängigkeiten), wie diese Informationen zu ermitteln sind oder wie die genaue Semantik der Information ist, kann aber nicht angeben werden. Die CAMI ist auf das Management einzelner Bausteine beschränkt; es ist nicht vorgesehen, das Management einer bausteinbasierten Anwendung durch Zusammenführen der CAMIs der einzelnen Bausteine zu ermöglichen. Somit liegt der Fokus nicht auf der Benutzersicht auf den gesamten erstellten Anwendungsdienst, sondern vielmehr auf dem Management der einzelnen Teilkomponenten des Dienstes. Die CAMI ermöglicht also nicht die erforderliche transaktionsbasierte Überwachung des Anwendungsdienstes, kann aber wertvolle, zusätzliche Informationen für die Überwachung einzelner Bausteine liefern. Lediglich durch die Möglichkeit, Abhängigkeiten zu anderen Bausteinen anzugeben bzw. durch die Anforderung, Fehlersituationen in anderen Bausteinen erkennen und melden zu können, wird ein gewisser Grad an Integration der einzelnen Managementschnittstellen erreicht.

4.2.2 Forschungsansätze

Auch im Bereich der Forschung ist in den letzten Jahren eine große Menge an Ansätzen für das Anwendungsmanagement entstanden. Die folgenden Abschnitte sollen beispielhaften Einblick in diejenigen Bestrebungen geben, die inhaltlich eine enge Verwandschaft zur vorliegenden Arbeit aufweisen.

4.2.2.1 Biaggiolini und Harms: Automatisierung des Fehlermanagements bausteinbasierter Anwendungen

In [BiHa 97] beschreiben Biaggiolini und Harms einen Ansatz, wie das Management bausteinbasierter Anwendungen automatisiert werden könnte. Sie beschränken sich hierbei jedoch ausschließlich auf den Bereich des Fehlermanagements. Die Bausteinarchitektur, auf der der Ansatz basiert, ist eine sogenannte *data flow* oder *pipe-and-filter* Architektur. Dies bedeutet, daß ein Datenobjekt nacheinander an eine Reihe von Bausteinen zur Bearbeitung übergeben wird. Bausteine akzeptieren also ein Datenobjekt als Eingabeparameter, führen beliebige Berechnungen und Transformationen darauf aus und geben das Datenobjekt zur Weiterverarbeitung wieder aus. Typische Beispiele für eine derartige Architektur sind *UNIX shells* mit dem bekannten *Pipe*-Mechanismus, aber auch z.B. *Email*-Systeme, die *Mails* verarbeiten und weiterversenden.

Die Anwendungsüberwachung basiert auf den folgenden Mechanismen:

- Konsistenzprüfung der Daten
  Bevor ein Datenobjekt an einen Baustein zur Bearbeitung übergeben wird, werden in

das Datenobjekt integrierte Konsistenzprüfungen (z.B. Überprüfung von Invarianten) durchgeführt.

- Zählen der „Hops“
  Mit jedem Betreten eines Bausteins wird ein Zähler inkrementiert. Somit können z.B. Endlosschleifen zwischen Bausteinen erkannt werden.

- Überprüfung der Warteschlangen
  Durch Überprüfung der Länge der Warteschlangen der einzelnen Bausteine, können Engpässe erkannt werden.

- Selbsttest-Routinen der Bausteine
  Jeder Baustein muß eine Selbsttest-Routine enthalten, die in regelmäßigen Abständen oder bei Auftreten einer Fehlersituation ausgeführt werden kann.

Bei Auftreten eines Problems werden dann verschiedene Möglichkeiten angegeben, wie die tatsächliche Ursache des Problems zu bestimmen ist bzw. wie das Problem zu beheben ist. Im einzelnen besteht die Möglichkeit

- Testtransaktionen auf einzelnen Bausteinen durchzuführen,
- Diagnosebausteine in die Anwendung einzubauen,
- den Abhängigkeitsgraphen der Anwendung bis zum verantwortlichen Baustein zurückzuverfolgen.

Ist der fehlerhafte Baustein identifiziert, kann dieser umkonfiguriert, neu initialisiert oder auch ersetzt werden. Alternativ kann die Anwendungslogik angepaßt werden.

Der Ansatz von Biaggiolini und Harms stellt einen ersten Versuch dar, das Management von bausteinbasierten Anwendungen zu automatisieren. Dieser ist allerdings auf einen kleinen Teil der existierenden Bausteinarchitekturen (*data flow* bzw. *pipe-and-filter* Architekturen) beschränkt. Desweiteren fokussiert der Ansatz auf die Überwachung einzelner Bausteine und nicht auf die geforderte, transaktionsbasierte Überwachung des gesamten Anwendungsdienstes. Da die Bausteine und die Datenobjekte selbständig Managementinformation zur Verfügung stellen müssen, ist der Ansatz für die Entwickler relativ aufwendig. Die Instrumentierung der Bausteine muß manuell erfolgen. Eine Methodik zur Erstellung des benötigten Abhängigkeitsgraphen wird nicht angegeben. Auch über die Selbsttest-Routinen, die von den einzelnen Bausteinen zu implementieren sind, wird keine Aussage getroffen. Es bleibt also im Verantwortungsbereich des Bausteinentwicklers, geeignete Selbsttests zu entwerfen und zu implementieren.

4.2.2.2 Neumair: GAMOCs

Im Mittelpunkt des Ansatzes von Neumair [Neum 98] steht die Definition eines homogenen Managementmodells für verteilte Anwendungen unter Verwendung der Konzepte des

Open Distributed Processing (ODP) [OMG 96-09-03]. Dieses soll eine Grundlage für das generische Management verteilter Anwendungen darstellen, darüber hinaus aber auch als Ausgangsbasis für Weiterentwicklungen in vielen Bereichen dienen.

Anhand der für die Modellierung von Managementinformation wichtigsten *viewpoints* des *RM ODP*, des *engineering viewpoint* und des *computational viewpoint*, werden generische Managementklassen für verteilte Anwendungen (*Generic Application Managed Object Classes (GAMOCs))* hergeleitet (z.B.: *cluster*, *capsule*, *operational interface*, ...). In weiteren Schritten wird den einzelnen GAMOCs management-relevante Information zugeordnet, d.h. es wird die Managementinformation definiert, die von den einzelnen GAMOCs zu erbringen ist.

Der Ansatz von Neumair stellt eine interessante neue Variante der Modellierung generischer Managementinformation verteilter Anwendungen dar. Mit Hilfe der GAMOCs wird es möglich, Informationen aller Managementfunktionsbereiche einheitlich darzustellen. Allerdings wird eine Systemsicht der zu managenden Anwendung unterstellt, d.h. es ist nicht möglich, die Verfügbarkeit der Anwendung aus der Sicht des Benutzers zu bestimmen. Darüber hinaus beschränkt sich der Ansatz auf die Modellierung der benötigten Information und liefert keine Hilfestellung bei der tatsächlichen Gewinnung der Information.

4.2.2.3 Kaiser: Bestimung der Verfügbarkeit von anwendungsorientierten Diensten

Kaiser stellt in seiner Dissertation [Kais 99] eine Methodik zur Erstellung von Abhängigkeitsgraphen vor, mit deren Hilfe die Verfügbarkeit verteilter, anwendungsorientierter Dienste bestimmt werden kann. Hierbei werden die Abhängigkeiten eines Dienstes von seinen Subdiensten modelliert, was die Bestimmung der Verfügbarkeit des Dienstes aus den Verfügbarkeiten der Subdienste gestattet.

Die Methodik besteht aus drei Schritten: Zunächst wird ein generischer Abhängigkeitsgraph erstellt, der dann verfeinert und schließlich für eine konkrete Ausprägung eines Dienstes instantiiert wird. Es wird zwischen `UND`- und `ODER`-Relationen unterschieden. Aus den Informationen über die Subdienste kann somit mittels des Abhängigkeitsgraphen die Verfügbarkeit des Dienstes selbst bestimmt werden.

Die Arbeit liefert eine Methodik für die Generierung von Abhängigkeitsgraphen, was eine wichtige Grundvoraussetzung für die Bestimmung der Verfügbarkeit von anwendungsorientierten Diensten darstellt. Die Methodik beschreibt jedoch einen komplexen und manuell durchzuführenden Prozeß; um die Praxisrelevanz der vorgeschlagenen Lösung zu erhöhen, wäre eine Automatisierung dieses Vorganges erforderlich. Die Gewinnung der Ausfalldaten über die Teildienste wird in der Arbeit zwar allgemein untersucht, es wird jedoch keine Methodik angegeben, die es erlaubt, dies für beliebige Dienste automatisiert durchzuführen.

4.2.2.4 Kar, Keller und Calo: Automatische Ermittlung von Abhängigkeiten im Anwendungsmanagement

Ähnlich der bereits vorgestellten Arbeit von Kaiser liegt der Schwerpunkt dieser Arbeit [KKC 00] auf der Ermittlung der Abhängigkeiten der unterschiedlichen Anwendungen eines Systemes sowie der darunterliegenden Infrastrukturkomponenten. Im Gegensatz zur oben vorgestellten Arbeit erlaubt der Ansatz von Kar et al. allerdings die automatische Ermittlung der Abhängigkeiten.

Die Ermittlung der Abhängigkeiten erfolgt hierbei aus ohnehin im System vorhandenen Informationen über die Installationsvoraussetzungen bestimmter Anwendungen. Die Informationen sind in einem *System Repository*, wie z.B. der *Windows Registry* oder dem *AIX Object Data Manager (ODM)* gespeichert und können über definierte Schnittstellen ausgelesen werden. Ein Beispiel ist ein *Webserver*, der als Installationsvoraussetzung die Existenz eines geeigneten *TCP/IP-Stacks* aufweist. Man kann nun davon ausgehen, daß dieser *Webserver* nur dann verfügbar ist, wenn der entsprechende *TCP/IP-Stack* ebenfalls verfügbar ist. Aus diversen Konfigurationsdateien lassen sich darüber hinaus Abhängigkeiten über Systemgrenzen hinweg, sogenannte *Inter-System Dependencies*, ermitteln.

Der vorgestellte Ansatz dient der Automatisierung der Erstellung von Abhängigkeitsgraphen. Er stellt eine wesentliche Erleichterung gegenüber der manuellen Ermittlung der Abhängigkeiten dar. Die Bestimmung der Abhängigkeiten der einzelnen Anwendungen und Systemkomponenten allein ist jedoch nicht ausreichend, um die Verfügbarkeit von Anwendungsdiensten ermitteln zu können. Es sind Aussagen erforderlich, wie die Informationen der einzelnen Teilkomponenten zu ermitteln und zu interpretieren sind. Dies ist aber nicht der Fokus der vorgestellten Arbeit. Die Haupteinsatzgebiete dieses Ansatzes liegen im Bereich des Konfigurations- und Fehlermanagements; in den Bereichen Leistungs- und Abrechnungsmanagements ist die bloße Bestimmung von Abhängigkeiten nicht ausreichend. Die Autoren verweisen hier auf Instrumentierungsansätze wie beispielsweise die ARM API (vgl. Abschnitt 4.2.1.3).

4.2.2.5 Frolund et al.: SoLOMon

Frolund et al. stellen in [FJP 99] das System `SoLOMon` (*Service Level Objective Monitor*) vor. Hierbei handelt es sich um ein System, das es erlaubt, die typischerweise von Managementsystemen zur Verfügung gestellten Parameter auf Parameter abzubilden, die zur Überwachung der Dienstgüte von Diensten wünschenswert wären. Hierzu wird die *Activity Monitoring Language (AML)* definiert, eine deklarative Sprache, die es erlaubt, die von der Instrumentierung gelieferten Parameter auf die benötigten Parameter abzubilden. Es wird keine neue Instrumentierungsmethode vorgeschlagen, sondern eine Integration der existierenden Ansätze versucht.

Ein wesentliches Konstrukt der AML sind sogenannte *provider*, die eine Abstraktion realer Instrumentierungspunkte darstellen. Mit Hilfe von Metriken (*metrics*) wird für die unterschiedlichen *provider* festgelegt, wie ein bestimmter Parameter, beispielsweise die Antwortzeit, je nach Art der Instrumentierung zu bestimmen ist. Eine *source* stellt einen logischen Instrumentierungspunkt dar, der – evtl. nach Vorverarbeitung und Filterung – die eingegangene Information an beliebige Managementsysteme weiterleitet.

Ziel bei der Entwicklung von `SoLOMon` war die Integration der unterschiedlichen, am Markt vorhandenen Instrumentierungstechniken sowie die Reduktion der zu übertragenden Information durch geeignete Vorverarbeitung. Das System liefert aber keine Hilfestellung bzgl. der Instrumentierung von Anwendungen, sondern geht vielmehr von bereits instrumentierten Anwendungen aus, die geeignet in ein Managementsystem zu integrieren sind.

4.2.2.6 Hellerstein et al.: ETE

Der Ansatz von Hellerstein et al. [HMMT 99] verzichtet auf die Instrumentierung von Anwendungen zur direkten Überwachung von Transaktionen. Stattdessen werden an wichtigen Stellen im *Code* der Anwendung *Events* ausgelöst, mit deren Hilfe ein externer „Transaktionsgenerator“ dann in der Lage ist, die zu überwachenden Transaktionen zu rekonstruieren.

Die Autoren versprechen sich von dieser Lösung eine größere Flexibilität in der Definition der zu überwachenden Transaktionen, da aufgrund des generierten *Event*-Stroms ohne großen Aufwand unterschiedlichste Transaktionen zu definieren und somit zu überwachen sind. Somit könne ein Dienstanbieter die Überwachung besser dynamisch dem Nutzungsverhalten seiner Nutzer anpassen.

Auch wenn die Möglichkeit der dynamischen Anpassung von zu überwachenden Transaktionen an das Nutzungsverhalten mit der vorgeschlagenen Lösung erleichtert werden kann, gibt der Ansatz keine Hilfestellung bezüglich des zugrundeliegenden Problems der Identifikation der geeigneten Meßpunkte. Die Stellen im *Code* der Anwendung, an denen ein *Event* ausgelöst werden soll, müssen weiterhin vom Anwendungsentwickler manuell identifiziert und instrumentiert werden. Auch das Problem der Korrelation der einzelnen Subtransaktionen zu ihren übergeordneten Transaktionen wird von diesem Ansatz nicht berührt. Fraglich ist weiterhin, ob die durch den vorgeschlagenen Ansatz erreichte Flexibilität bezüglich der Definition der zu überwachenden Transaktionen überhaupt erforderlich ist, da in den meisten Fällen davon auszugehen ist, daß Benutzer, die identisch implementierte Dienste nutzen, keine unterschiedliche Überwachung der erbrachten Dienste fordern.

4.2.2.7 Weitere Forschungsansätze

Neben den oben im Detail vorgestellten Ansätzen existieren eine Reihe weiterer Ideen und Vorschläge im Bereich des Managements von Anwendungsdiensten, die aber nur geringen Bezug zur Aufgabenstellung der vorliegenden Arbeit aufweisen.

So geben beispielsweise Schade et al. in [TSK 96] und [STK 96] eine Sprache an, mit der man die Managementinformation verteilter Objekte beschreiben kann. Sie treffen allerdings keine Aussage darüber, welche Information tatsächlich benötigt wird, wie die Information der einzelnen Objekte eine Anwendung zu korrelieren ist oder wie eine Instrumentierung der Objekte zu erfolgen hat. Haworth beschreibt eine Möglichkeit, die ARM API (vgl.Abschnitt 4.2.1.3) zu verwenden, ohne dabei den *Source-Code* der Anwendung verändern zu müssen [Hawo 97]. Er schlägt vor, mit Hilfe von Skripten Benutzertransaktionen zu simulieren und die Skripte zu instrumentieren. Dieser Ansatz weist somit natürlich alle in Abschnitt 4.1.1.3 dargestellten Probleme der Simulation von Benutzertransaktionen auf.

4.2.3 Herstellerspezifische Lösungen

Neben den bereits vorgestellten Ansätzen von Standardisierungsgremien und Forschung soll nun beispielhafter Einblick in am Markt verfügbare Lösungen gegeben werden. Im Rahmen der Untersuchungen wurde eine Vielzahl von Produkten betrachtet, von denen in den folgenden Abschnitten exemplarisch die Produkte der Firma *Tivoli* und der Firma *Candle* vorgestellt werden. Diese Auswahl begründet sich damit, daß die Produkte dieser Firmen die hinsichtlich der vorliegenden Arbeit interessantesten Ansätze realisieren. Im Anschluß daran wird ein kurzer Überblick über weitere am Markt verfügbare Produkte gegeben.

4.2.3.1 Tivoli

Die Firma Tivoli bietet mit dem *Tivoli Management Framework (TMF)* den Rahmen für eine Vielzahl von Managementlösungen. Das TMF stellt neben einer CORBA-basierenden Kommunikationsinfrastruktur für Managementanwendungen weitere Basisdienste zur Verfügung, die für das Management großer IT-Infrastrukturen benötigt werden. Beispielsweise existieren Dienste für die Verteilung von Software auf eine Vielzahl von Systeme oder für das Anstoßen regelmäßiger Routinetätigkeiten. Über definierte und offengelegte Schnittstellen können Managementanwendungen (auch von Drittanbietern) auf die Dienste des TMF zugreifen. Auf eine detaillierte Diskussion des TMF muß mit Rücksicht auf den Rahmen der Arbeit verzichtet werden, hierfür sei der Leser auf [Tiv 99] verwiesen.

Die folgenden Abschnitte beschreiben zwei Ansätze für das Management verteilter Anwendungen, die auf dem TMF aufsetzen. Es handelt sich hierbei um den *Tivoli Application Performance Manager (TAPM)* sowie den *Global Enterprise Manager (GEM)*. GEM basiert auf der *Application Management Specification (AMS)*, die aus diesem Grunde ebenfalls kurz eingeführt wird.

Tivoli Application Performance Manager

Mit dem *Tivoli Application Performance Manager (TAPM)* [TAPM 99] stellt *Tivoli* ein Produkt bereit, das unterschiedliche Methoden der Anwendungsüberwachung miteinander verbindet. Die aktuelle Version (Version 1) erlaubt Anwendungsinstrumentierung und darüber hinaus, falls nicht-instrumentierte Anwendungen vorliegen, die Ausführung simulierter Transaktionen. Für zukünftige Versionen ist weiterhin die Integration einer oberflächenbasierten Technik (*client capture*) angekündigt.

Der Bereich der Anwendungsinstrumentierung wird durch Bereitstellung eines ARM-konformen Meßagenten abgedeckt, der, wie in Abschnitt 4.2.1.3 beschrieben, zur jeweils zu überwachenden Anwendung gebunden werden muß.

Die Überwachung simulierter Transaktionen erfolgt ebenfalls unter Verwendung der ARM API. Hierfür wurde folgender Ansatz gewählt: Mit Hilfe zweier Produkte der Firma *Mercury*, *WinRunner* und *LoadRunner*, ist es möglich, Skripte zu generieren, die Benutzereingaben auf einer Oberfläche oder aber direkten Zugriff über das Netz auf einen Server gestatten. Diese Skripte werden mit Aufrufen der ARM API versehen. Es wird somit möglich, mit dem Standard-ARM-Meßagenten die Zeiten der simulierten Transaktionen zu messen und in die instrumentierungsbasierte Überwachung der Anwendung zu integrieren.

Für Standardanwendungen werden bereits vorgefertigte Skripte angeboten, darüber hinaus besteht allerdings auch die Möglichkeit, eigene Skripte zu erstellen. Dies kann durch Aufzeichnung von Benutzereingaben mit verhältnismäßig geringem Aufwand geschehen.

Durch die Kombination der Anwendungsinstrumentierung mit dem Einsatz simulierter Transaktionen wird es möglich, die Vorteile der Instrumentierung auszunutzen, falls entsprechend instrumentierte Anwendungen vorliegen, gleichzeitig aber auch nicht-instrumentierte Anwendungen in die Überwachung mit einbeziehen zu können. Da die Verwendung simulierter Transaktionen aber nicht die erforderliche Managementinformation liefern kann und die Verbreitung geeignet instrumentierter Anwendungen nach wie vor als zu gering einzustufen ist, kann das Werkzeug die in Abschnitt 3 gestellten Anforderungen nicht zufriedenstellend erfüllen.

Application Management Specification

Die Firma Tivoli begann im Jahre 1995 mit der Spezifikation einer Beschreibungstechnik für Anwendungen, der *Application Management Specification (AMS)* [AMS 2]. Die AMS bietet die Möglichkeit, Information über Anwendungen in maschinenlesbarer Form anzulegen, die von Managementwerkzeugen für das Management der Anwendungen benötigt wird. Neben dieser Beschreibung sind anwendungsspezifische Werkzeuge und Skripte erforderlich, die ebenfalls mit Hilfe von AMS beschrieben werden können.

Mit Hilfe der AMS ist es möglich, Informationen über den kompletten Lebenszyklus einer Anwendung zu beschreiben, d.h. Informationen über ihre Verteilung und Installation beispielsweise ebenso wie Informationen über ihre Überwachung im laufenden Betrieb. Im Rahmen der vorliegenden Arbeit ist natürlich die Überwachung der Anwendung im laufenden Betrieb von größter Bedeutung, weswegen im folgenden die übrigen Aspekte nur am Rande betrachtet werden.

Die AMS definiert sechs sogenannte *Building Blocks*, also Komponenten, die für die Beschreibung einer Anwendung verwendet werden. Diese, sowie ihre Zusammenhänge und ihre Einordnung in das CIM der DMTF (siehe Abschnitt 4.2.1.2) sind in Abbildung 4.6 dargestellt. Neben dem *Application Building Block* und dem *Software Component Building Block*, die der Beschreibung der eigentlichen Anwendung dienen und daher im folgenden noch ausführlich betrachtet werden, existieren vier weitere Komponenten: Hierbei handelt es sich um die *Building Blocks* für *Business Systems*, *Business System Subsystems*, *Business System Components* und das *Business System Mapping*. Diese dienen dazu, die eventuell aus mehreren Anwendungen zusammengesetzten Systeme eines Unternehmens, die bestimmte Geschäftsprozesse erbringen, zu modellieren. Auf eine weitere Untersuchung dieser Komponenten muß aus Platzgründen an dieser Stelle verzichtet werden. Nähere Informationen finden sich in [AMS 2].

Mit Hilfe des *Application Building Blocks* läßt sich allgemeine Information über die Anwendung, wie z.B. Name der Anwendung, Hersteller oder Versionsnummer modellieren. Ebenso wird hier die Information über die von der Anwendung erbrachten Transaktionen abgelegt. Es bestehen enge Verknüpfungen zur ARM API (siehe Abschnitt 4.2.1.3). So lassen sich nicht nur die von der Anwendung instrumentierten und somit überwachbaren Transaktionen angeben, es lassen sich auch Schwellwerte definieren, die für eine ordnungsgemäße Erbringung einer Transaktion einzuhalten sind. Darüber hinaus dient der *Application Building Block* dazu, die einzelnen Komponenten der Anwendung, die in *Software Component Building Blocks* beschrieben werden, zu identifizieren.

Jeder *Software Component Building Block* beschreibt eine Komponente einer Anwendung. Unter einer *Software Component* wird hierbei ein Teil einer Anwendung verstanden, der auf einer bestimmten Plattform ausgeführt werden kann und der als Einheit gemanaged (z.B. verteilt und installiert) werden kann. Die Beschreibung enthält somit z.B. Information darüber, wie die Komponente verteilt und installiert werden muß, welche Abhängigkeiten zu anderen Komponenten oder zu darunterliegenden Systemen bestehen oder welche Parameter für den ordnungsgemäßen Betrieb der Komponente überwacht werden sollten.

Die eigentliche Überwachung der Anwendung erfolgt mit Hilfe sogenannter Monitore. Da explizit keine Instrumentierung der Anwendung vorausgesetzt wird (mit Ausnahme einer eventuell vorhandenen ARM-Instrumentierung) ist dies erforderlich, um die benötigten Managementinformationen zu ermitteln. Unter einem Monitor werden kleine Programme oder Skripte verstanden, die auf dem jeweiligen System ausgeführt werden und Informationen über den Zustand der Anwendung liefern. Es existieren eine Reihe von Standardmonitoren, die durch geeignete Konfiguration für jede beliebige Anwendung verwendet werden können und speziell für eine Anwendung konzipierte Monitore. Typische Beispiele für Standardmonitore sind Monitore, die die Logdateien von Anwendungen überwachen, aber auch Monitore, die anhand von Systeminformationen auf den Zustand der Anwendung schließen.

Die Beschreibung der Managementinformation basiert auf dem MIF der DMTF (siehe Abschnitt 4.2.1.2) und erfolgt in sogenannten *Application Description Files (ADF)*. Die *Application Building Blocks* werden dabei in *Global Description Files (GDF)*, die *Softwa-*

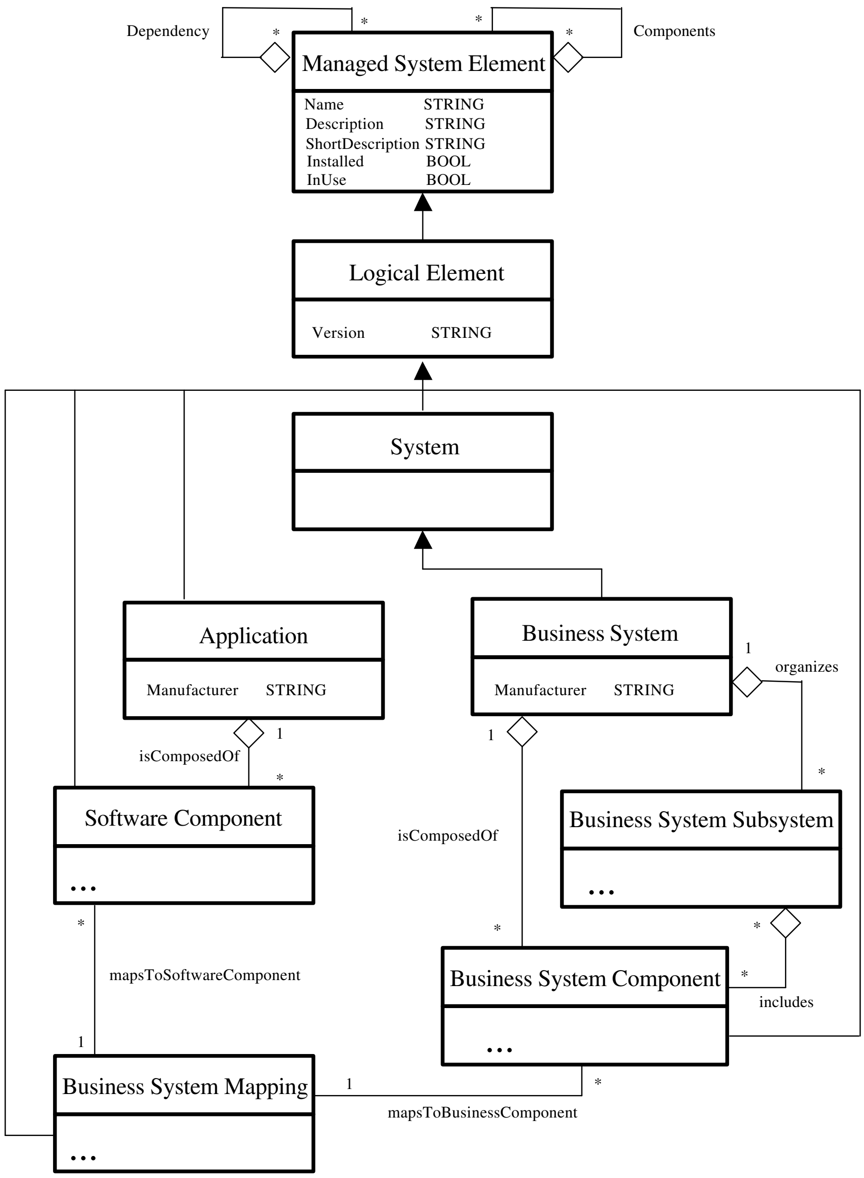

Abbildung 4.6: Die *Building Blocks* der AMS und ihre Einordnung im CIM der DMTF

re Component Building Blocks in *Component Description Files (CDF)* beschrieben. Die ADFs, die eine Anwendung beschreiben sowie die Monitore und Skripte, die für ihre Verteilung, Installation und Überwachung benötigt werden, können in sogenannten *Application Management Packages (AMP)* zusammengefaßt werden, die von Managementsystemen eingelesen werden können. Die Managementsysteme verteilen die Skripte und Monitore auf die entsprechenden Systeme und können anhand der in den ADFs enthaltenen Informationen das Management der Anwendung durchführen.

Die AMS bietet die Möglichkeit, Managementinformation über den gesamten Lebenszyklus einer Anwendung in maschinenlesbarer Form zu beschreiben und gestattet somit die Überwachung von anwendungsspezifischen Details mit Hilfe generischer Managementwerkzeuge. Dies stellt einen erheblichen Fortschritt gegenüber auf eine bestimmte Anwendung spezialisierten Managementwerkzeugen dar. Allerdings zeigte sich bei näherer Betrachtung [Alde 00], daß die Erstellung der ADFs mit einem erheblichen Aufwand verbunden ist und meist spezielle Skripte und Monitore für die jeweilige Anwendung zu erstellen sind. Auch dies stellt einen nicht unerheblichen Aufwand dar. Darüber hinaus ist es nur sehr eingeschränkt möglich, die eigentlich benötigte, nutzerorientierte Information zu ermitteln. Diese beschränkt sich auf die Definition von Transaktionen und Schwellwerten für deren Erbringung, die eigentliche Instrumentierung der Anwendung muß weiterhin z.B. mit Hilfe von ARM im *Source-Code* der Anwendung erfolgen.

Global Enterprise Manager

Der *Global Enterprise Manager (GEM)* [Tiv 99] erlaubt das Management der Einzelsysteme und -anwendungen aus Geschäftsprozeßsicht. Die mit Hilfe der AMS definierten Geschäftsprozesse können in GEM geladen und grafisch veranschaulicht werden. Mit Hilfe des oben nur kurz angesprochenen *Business System Mappings (BSM)* ist es möglich, die Abbildung zwischen Geschäftsprozessen bzw. ihren Komponenten und Subprozessen und den die Geschäftsprozesse erbringenden Softwarekomponenten herzustellen. Ende 2000 wurde der GEM von *Tivoli* vom Markt genommen und wird seitdem unter dem Namen *Tivoli Business System Manager (TBSM)* vertrieben. Zum heutigen Zeitpunkt lassen sich keinerlei Unterschiede hinsichtlich der Funktionalität der beiden Produkte angeben.

Neben der Möglichkeit, die eigentliche Überwachung und Veranschaulichung der Geschäftsprozesse durchzuführen (mit Hilfe des sogenannten *GEM-Servers* bzw. *GEM-Clients*) unterstützt der GEM den Anwendungsentwickler insbesondere auch in der Erstellung der AMS-Dateien sowie der Skripte und Monitore [GEM 99]. Hierzu existieren zwei Werkzeuge, der *Tivoli Module Builder (TMB)* und der *Tivoli Module Designer (TMD)*. Der TMB bietet unterschiedliche *Templates*, aus denen mit verhältnismäßig geringem Aufwand Skripte und Monitore erstellt werden können. Weiterhin ermöglicht er Tests der erstellten Werkzeuge und generiert daraus unter Hinzunahme der zugehörigen AMS-Beschreibung AMPs, die dann vom GEM-Server geladen werden können. Der TMD dient hingegen der Erstellung der eigentlichen AMS-Beschreibung. Er stellt eine grafische Oberfläche bereit, mit deren Hilfe die benötigte Information eingegeben werden kann. Anschließend generiert er daraus ADFs, die als Eingabe für den TMB für die Erstellung von AMPs verwendet werden können.

Eine ausführliche Untersuchung des GEM [Maye 00] hat gezeigt, daß er in der Lage ist, den Aufwand für die Erstellung einer Anwendungsbeschreibung zu reduzieren und teilweise zu automatisieren. Die Überwachung der Benutzertransaktionen erfolgt allerdings auch bei Verwendung des GEM (wie bei AMS beschrieben) mittels einer ARM-

Instrumentierung der zu überwachenden Anwendung. Mit Ausnahme der Möglichkeit, die instrumentierten Transaktionen zu beschreiben sowie Schwellwerte für deren Erbringung festzulegen, gibt es also auch hier keinerlei Unterstützung für die weiterhin erforderliche Instrumentierung der zu überwachenden Anwendung.

4.2.3.2 Candle

Die Firma *Candle* bietet mit dem Produkt *ETEWatch* [ETE 00] einen vielversprechenden und interessanten Ansatz zur Anwendungsüberwachung. Es handelt sich um einen oberflächenbasierten Ansatz, der durch das Abfangen von *Events* der *MS Windows*-Oberfläche die Zeiten für die Erbringung von Benutzertransaktionen ermittelt.

Hierfür kommen sogenannte Kollektoren zum Einsatz, die auf den jeweiligen *Client*-Systemen installiert werden müssen und die gewonnenen Daten an die zugehörigen Managementsysteme weiterleiten können. Zur Integration in Managementanwendungen anderer Hersteller verfügt *ETEWatch* über die Möglichkeit, die ermittelten Transaktionszeiten durch Aufrufe der ARM API weiterzuleiten.

In [Fisc 01] wurde eine detaillierte Untersuchung von *ETEWatch* durchgeführt. Es zeigte sich, daß zur Identifikation der Benutzertransaktionen zwei Möglichkeiten vorgesehen sind, eine generische und eine anwendungsspezifische Variante: Bei der vollständig generischen Variante wird jede Betätigung eines Elements der Anwendungsoberfläche als der Beginn einer Benutzertransaktion interpretiert, während der Neuaufbau des Bildschirms als das Ende der Transaktion betrachtet wird. Hierbei ergeben sich allerdings Probleme hinsichtlich der Zuordnung der Transaktionsenden. Ein Bildschirmaufbau findet nicht zwangsläufig nur am Ende einer Benutzertransaktion statt, sondern kann beispielsweise auch vom Benutzer manuell angefordert werden. Auch läßt sich auf diese Weise nicht zwischen erfolgreich und nicht erfolgreich beendeten Transaktionen unterscheiden.

Als Alternative hierzu existiert die Möglichkeit, anwendungsspezifisch Oberflächen-*Events* auf Transaktionen abzubilden. Für einige Anwendungen existieren bereits Standard-Abbildungen, außerdem kann man mit Hilfe eines sogenannten *Customizers* eigene Anwendungen entsprechend anpassen. Es zeigte sich hierbei allerdings, daß die vollständige Abbildung der Events einer Anwendung auf Transaktionen nur durch den Anwendungsentwickler sinnvoll durchgeführt werden kann. Die untersuchten Standard-Abbildungen wiesen teilweise erhebliche Mängel auf. So wurde beispielsweise bei einem *Webbrowser* das *Event* zum Abbruch des Ladevorgangs nicht mit berücksichtigt, was zu Fehlinterpretationen von Transaktionen führte.

In einer neueren Version von *ETEWatch* wurde zusätzlich zur oberflächenbasierten Überwachung noch die Ausführung von Testtransaktionen auf Netzebene integriert. Dies bedeutet, daß nach Abschluß einer Transaktion ein `PING`-Paket an den Sender geschickt wird. Da die Größe des Paketes an die im Rahmen der Transaktion übertragene Datenmenge angepaßt wird, korreliert die Netzlaufzeit dieses Testpakets somit annähernd mit der Laufzeit der tatsächlichen Anfrage. Somit wird es möglich, die gemessene Antwortzeit in die Netzverzögerung und die Bearbeitungszeit des *Servers* aufzuteilen.

ETEWatch stellt aktuell einen der interessantesten Ansätze im Bereich der Antwortzeitüberwachung verteilter Awendungen dar. Allerdings ergeben sich erhebliche Probleme bei der nachträglichen Erstellung der Abbildung zwischen Oberflächen-*Events* und Transaktionen sowie bei der Betrachtung des Bildschirmneuaufbaus als generischem Transaktionsende. Durch die Hinzunahme der Testtransaktionen auf Netzebene läßt sich ein typischer Nachteil oberflächenbasierter Methoden, die fehlende Möglichkeit der Überwachung von Subtranksaktionen, zwar geringfügig verbessern, eine detaillierte Betrachtung der einzelnen Subtransaktionen innerhalb des *Servers* ist damit aber keinesfalls zu erreichen. Ein weiteres Problem ergibt sich daraus, daß das Produkt zum heutigen Zeitpunkt ausschließlich auf *MS Windows*-Systeme beschränkt ist, also nicht in beliebigen heterogenen Umgebungen zum Einsatz kommen kann.

4.2.3.3 Weitere herstellerspezifische Lösungen

In Rahmen der Studien zur vorliegenden Arbeit wurden weitere Produkte aus dem Bereich des Anwendungsmanagements untersucht, die sich für die eigentliche Anwendungsüberwachung allerdings als nur bedingt einsatzfähig gezeigt haben:

In [Skla 00] wurde das Produkt *Unicenter* der Firma *Computer Associates* untersucht. Der Bereich der Anwendungsüberwachung erstreckt sich bei diesem Werkzeug vornehmlich auf die Überwachung von Systemparametern. Weiterhin ist es möglich, sogenannte betriebsablauforientierte Sichten (*Business Process View (BPV))* zu definieren, die aber ebenfalls keine Überwachung nutzerorientierter Parameter gestatten.

Das Werkzeug *Firehunter* der Firma *Hewlett Packard* wurde in [Fisc 01] untersucht. Es handelt sich hierbei um eine auf simulierten Transaktionen basierende Lösung. Ein aktiver Agent (*active agent*) eines Meßsystems (*Diagnostic Measurement Server*) generiert Testanfragen an die zu überwachenden *Server* und mißt die Zeitdauer bis zu ihrer Beantwortung. Darüber hinaus existieren passive Agenten (*passive agent*) auf den zu überwachenden Systemen, die eine Überwachung ausgewählter Systemparameter gestatten um somit detailliertere Aussagen über den Zustand der *Server* zu ermöglichen.

Auch das Werkzeug *INFRA-XS* der Firma *Geyer & Weinig* [Fisc 01] kombiniert zwei unterschiedliche Verfahren der Anwendungsüberwachung. Die beiden zum Einsatz kommenden Verfahren sind die Simulation von Transaktionen sowie die Überwachung des Netzverkehrs. Benutzereingaben können aufgezeichnet und zu einem späteren Zeitpunkt wiederholt werden. Somit gelangt man zu Referenztransaktionen, die in regelmäßigen Abständen durchgeführt werden. Durch Überwachung des Netzverkehrs ist es möglich, die gemessene Zeitdauer der Testransaktionen in Subtransaktionen aufzuteilen und somit die Fehlerdiagnose zu erleichtern.

4.3 Zusammenfassung: Möglichkeiten und Defizite existierender Ansätze

Durch eine Klassifikation der existierenden Methoden zur Überwachung von Anwendungsdiensten konnte die Untersuchung und Einordnung existierender Produkte und Ansätze auf Basis der zugrundeliegenden Konzepte erfolgen. Es wurden vier Klassen von Überwachungswerkzeugen identifiziert, die sich bezüglich der Informationsquelle unterscheiden: die Überwachung des Netzverkehrs, die Überwachung von Systemparametern, die *Client*-seitige Anwendungsüberwachung sowie die Überwachung der Gesamtanwendung.

Es zeigte sich, daß derzeit keine der vorgestellten Klassen den gestellten Anforderungen vollständig gerecht werden kann. Die Überwachung des Netzverkehrs bzw. von Systemparametern ermöglicht nicht die erforderliche Überwachung nutzerorientierter Parameter. Die *Client*-seitige Überwachung liefert zwar nutzerorientierte Parameter, kann aber den Detaillierungsgrad, der für eine zügige und erfolgreiche Fehlerdiagnose unabdingbar ist, nicht zur Verfügung stellen. Einzig durch Überwachung der Gesamtanwendung ist es möglich, die geforderten Parameter zu ermitteln. Im Bereich der Überwachung der Gesamtanwendung zeigt sich, daß eine Anwendungsbeschreibung meist ebenfalls keine ausreichende Information liefern kann und auf Anwendungsinstrumentierung zurückgegriffen werden muß. Dies ist allerdings bereits zum Zeitpunkt der Anwendungserstellung mit erheblichem Mehraufwand für den Anwendungsentwickler verbunden und ist zu einem späteren Zeitpunkt nahezu unmöglich.

Die Untersuchung erstreckte sich über die unterschiedlichen Standardisierungsgremien, Ansätze aus dem Bereich der Forschung sowie am Markt verfügbare Produkte unterschiedlicher Hersteller. Existierende Standards beschäftigen sich vorwiegend mit der Definition der erforderlichen Information, machen aber keinerlei Aussage darüber, wie diese Information zu ermitteln ist. Auch die Standards im Bereich der Instrumentierungstechniken sind hier wenig hilfreich: Sie definieren zwar Schnittstellen, mit denen die benötigte Information von der zu überwachenden Anwendung zur Verfügung gestellt werden kann, geben aber keine Hilfestellung bezüglich einer Methodik für die Instrumentierung oder gar hinsichtlich einer Automatisierung. Dies stellt heute den wesentlichen Einflußfaktor für die geringe Verbreitung von Instrumentierungstechniken dar.

Auch im Bereich der Forschung finden sich nur wenige Arbeiten, die sich mit der Automation des Anwendungsmanagements beschäftigen. Im Bereich der Automation der Instrumentierung von Anwendungen konnten keine Arbeiten gefunden werden.

Die Hersteller von Managementwerkzeugen haben die Problematik der Anwendungsüberwachung erkannt und versuchen, geeignet darauf zu reagieren. Die meisten bieten mittlerweile kombinierte Managementlösungen an, die unterschiedliche Konzepte integrieren und versuchen somit, die Nachteile der einzelnen Konzepte auszugleichen. Die

Abkehr von Methoden der Netz- und Systemüberwachung ist deutlich zu erkennen. Diese kommen in modernen Werkzeugen zur Überwachung von Anwendungsdiensten bestenfalls zur Unterstützung nutzerorientierter Verfahren zum Einsatz.

Es läßt sich also feststellen, daß das einzige der vorgestellten Konzepte, das in der Lage ist, alle geforderten Informationen zu liefern, die Anwendungsinstrumentierung ist; diese wird aber aufgrund des mit ihr verbundenen hohen Aufwands derzeit nur in geringem Umfang eingesetzt. Die meisten der aktuellen Arbeiten fokussieren auf eine Verbesserung der Alternativen zur Anwendungsinstrumentierung, z.B. durch Kombination unterschiedlicher Konzepte, um somit ähnlich aussagekräftige und detaillierte Informationen erbringen zu können. Arbeiten, die versuchen, den mit der Managementinstrumentierung einer Anwendung einhergehenden Aufwand zu verringern, sind zum heutigen Zeitpunkt nicht bekannt. Die speziellen Bedürfnisse bausteinbasierter Anwendungen werden nur von wenigen Arbeiten betrachtet: Das CAMI der DMTF beschreibt Managementanforderungen einzelner Bausteine, ohne dabei aber auf die Verknüpfung mehrerer Bausteine zu einer Anwendung einzugehen; Baggiolini und Harms stellen eine Lösung vor, die die Bausteinstruktur von Anwendungen berücksichtigt, geben dabei allerdings keinerlei Hilfestellung bezüglich der erforderlichen Instrumentierung der einzelnen Bausteine.

Der im nachfolgenden Kapitel vorgeschlagene Ansatz basiert auf der Instrumentierung von Anwendungen und ist insbesondere an die ARM API angelehnt. Darüber hinaus werden aber auch Aspekte des Produktes *ETEWatch* der Firma *Candle* mit in die Architektur aufgenommen, um die Identifikation der eigentlichen Benutzertransaktionen mit geringem Aufwand durchführen zu können. Die wesentlichen Punkte der im Folgenden vorgeschlagenen Architektur, das automatische Einfügen von Meßpunkten in die Anwendung aufgrund der Bausteinstruktur sowie die Zuordnung der einzelnen Meßwerte zu übergeordneten Transaktionen aufgrund von Kontrollflüssen, sind in keinem der aktuell existierenden Ansätze in ähnlicher Form bereits vorhanden.

Kapitel 5

Überwachung bausteinbasierter Anwendungen

Das folgende Kapitel beschreibt eine Architektur für die Überwachung bausteinbasierter Anwendungen, die die in Kapitel 3 ermittelten Anforderungen erfüllt. Grundsätzliches Ziel ist es, eine Managementschnittstelle sowohl für einzelne Bausteine als auch für daraus zusammengesetzte Anwendungen zu definieren. Die Zuordnung der Managementinformation der einzelnen Bausteine zur Managementinformation der Gesamtanwendung soll hierbei automatisch erfolgen.

Zunächst werden in Abschnitt 5.1 die wesentlichen Designentscheidungen, die bei der Entwicklung der Lösung zu treffen waren, dargestellt; Abschnitt 5.2 stellt dann einen naheliegend erscheinenden ersten Ansatz dar, der aber viele der gestellten Anforderungen nur unzureichend erfüllen kann. Ein zweiter Ansatz, der viele der Probleme des ersten Ansatzes nicht aufweist, wird in Abschnitt 5.3 vorgestellt. Auch wenn dieser Ansatz einen Großteil der Anforderungen bereits erfüllen kann, verbleiben dennoch nicht erfüllbare Anforderungen. Daher basiert die in Abschnitt 5.4 letztlich vorgeschlagene Architektur auf einer geeigneten Kombination der beiden Ansätze und kann somit den gestellten Anforderungen umfassend gerecht werden.

5.1 Lösungsansätze

Aufgrund der in Kapitel 3 ermittelten Anforderungen ergibt sich, daß eine Lösung folgende Grundvoraussetzungen erfüllen muß:

- Es muß möglich sein, die erforderlichen Informationen einzelner Basisbausteine zu ermitteln.
- Es muß möglich sein, die erforderliche Information zusammengesetzter Bausteine bzw. kompletter Anwendungen zu ermitteln.
- Die Zuordnung der Information eines Bausteins zur Information der ihn zusammensetzenden Bausteine (und umgekehrt) muß möglich sein.

Diese Anforderungen geben noch keine Lösung fest vor, sondern es ergeben sich Freiheitsgrade, die bei der Konzeption einer Lösung zu berücksichtigen sind. In diesem Abschnitt werden daher die wesentlichen Designentscheidungen dargestellt, die zu treffen waren, sowie ein kurzer Überblick über die näher untersuchten Ansätze gegeben.

5.1.1 Wesentliche Designentscheidungen

Bei der Entwicklung und Auswahl einer geeigneten Lösung waren zunächst die folgenden drei Designentscheidungen zu treffen:

- Woher stammt die erforderliche Managementinformation?
- Wie werden die Abhängigkeiten innerhalb der Anwendung beschrieben?
- Wie und wo erfolgt die Korrelation der Managementinformation?

Die unterschiedlichen Varianten, wie diese Fragestellungen zu beantworten sind, werden im folgenden kurz dargestellt.

Informationsquelle
In Abschnitt 3.2.1 wurde ein Modell bausteinbasierter Anwendungsdienste angegeben (siehe Abbildung 3.2). Man erkennt, daß jede bausteinbasierte Anwendung, die aus mehr als einem einzelnen Basisbaustein besteht, aus Bausteinen und Anwendungslogik zusammengesetzt ist. Dies liefert auch die möglichen Punkte für die Managementinstrumentierung. Die Managementinstrumentierung kann somit entweder

- innerhalb der Bausteine oder
- in der Anwendungslogik

erfolgen.

Beschreibung von Abhängigkeiten innerhalb der Anwendung
Um die Informationen einzelner Bausteine mit Informationen über die daraus zusammengesetzten Anwendungen korrelieren zu können, ist es erforderlich, die Zusammenhänge und Abhängigkeiten der einzelnen Bausteine innerhalb der Anwendung beschreiben zu können. Hierfür sind zwei grundsätzlich unterschiedliche Varianten vorstellbar:

- Statische Beschreibung zum Zeitpunkt der Anwendungserstellung.
- Dynamische Ermittlung zur Laufzeit der Anwendung.

Während der Anwendungserstellung werden die Verknüpfungen der einzelnen Bausteine untereinander festgelegt. Somit sind zu diesem Zeitpunkt auch ihre Abhängigkeiten bekannt. Darüber hinaus können vom Anwendungsentwickler noch weitere Informationen hinzugefügt werden. Es ist somit denkbar, einen Abhängigkeitsgraphen zu erstellen, der die zu diesem Zeitpunkt vorliegende Information beschreibt und somit für spätere Managementaufgaben zugänglich macht.

Die Alternative hierzu ist es, auf die Erstellung der Abhängigkeitsinformation zum Zeitpunkt der Entwicklung zu verzichten und diese Information stattdessen dynamisch zur Laufzeit aus den tatsächlich auftretenden Abläufen innerhalb der Anwendung zu ermitteln.

Informationskorrelation
Unabhängig davon, wie die Abhängigkeiten innerhalb der Anwendung ermittelt und beschrieben werden, ist zu entscheiden, wie und wo die Korrelation der Managementinformation erfolgt. Wiederum sind zwei unterschiedliche Möglichkeiten denkbar:

- Korrelation im Baustein.
- Korrelation im Managementsystem.

Findet die Korrelation bereits im jeweiligen Baustein statt, dann wird dem Managementsystem bereits vorverarbeitete Information über die zugrundeliegenden Bausteine übermittelt. Im Gegensatz dazu steht die Variante, bei der die vollständige Information an ein Managementsystem übergeben wird und erst dort aufgrund der oben beschriebenen Abhängigkeitsbeziehungen miteinander korreliert wird.

5.1.2 Untersuchte Ansätze

Im Rahmen der Arbeit wurden zwei unterschiedliche Lösungsvarianten untersucht und bewertet. Der zunächst naheliegend erscheinende Ansatz der *Komposition von Managementschnittstellen instrumentierter Bausteine* basiert auf der geforderten identischen Struktur der Managementschnittstellen von Basisbausteinen und zusammengesetzten Bausteinen. Durch Instrumentierung der Basisbausteine und Identifikation von Zusammenhängen und Abhängigkeiten zwischen diesen Bausteinen ist es möglich, die Managementinformation der zusammengesetzten Bausteine aus der Managementinformation der zugrundeliegenden Basisbausteine zu bestimmen. Dieser Ansatz wird in Abschnitt 5.2 vorgestellt und diskutiert. Aufgrund sich ergebender erheblicher Nachteile des Ansatzes wird aber von einer detaillierteren Betrachtung abgesehen und stattdessen ein zweiter Ansatz eingeführt, die *Automation der Managementinstrumentierung bausteinbasierter Anwendungen*. Dieser Ansatz ist in der Lage, durch automatisches Einfügen von Managementcode an den Schnittstellen zwischen den Bausteinen sowie durch dynamische Ermittlung der Abhängigkeiten zur Laufzeit, den in Kapitel 3 ermittelten Anforderungen an die Überwachung bausteinbasierter Anwendungsdienste weitgehend gerecht zu werden. Die letztlich gewählte Architektur basiert jedoch auf einer Kombination der beiden gewählten Ansätze.

5.2 Erster Ansatz: Komposition von Managementschnittstellen instrumentierter Bausteine

Dieser Abschnitt stellt den zunächst naheliegend erscheinenden Ansatz der Komposition von Managementschnittstellen einzelner Bausteine zu einer Managementschnittstelle des daraus zusammengesetzten Bausteins dar. Bei der Bewertung dieses Ansatzes wird sich allerdings zeigen, daß dieser einige der gestellten Anforderungen nur unzureichend erfüllen kann.

5.2.1 Idee

Die Idee des Ansatzes besteht darin, jeden Basisbaustein so zu instrumentieren, daß er die managementrelevante Information an einer – vom eigentlichen Dienstzugangspunkt getrennten – Managementschnittstelle zur Verfügung stellt. Die Managementschnittstelle eines zusammengesetzten Bausteins muß, wie in Abschnitt 3.2.2.1 dargestellt, dieselbe Struktur aufweisen wie die der darunterliegenden Bausteine. Die dort verfügbare Managementinformation soll nun *nicht* durch Instrumentierung des zusammengesetzten Bausteins ermittelt werden, sondern durch geeignete Verknüpfung der Managementinformation der darunterliegenden Bausteine. Aufgrund der Wiederverwendung der Basisbausteine ließe sich somit also eine erhebliche Verringerung des gesamten Instrumentierungsaufwandes herbeiführen.

Abbildung 5.1 zeigt hierfür ein einfaches Beispiel. Baustein A ist zusammengesetzt aus den beiden Basisbausteinen B und C. Mit Hilfe der Anwendungslogik werden die Nutzungsschnittstellen der Bausteine B und C so miteinander verknüpft, daß die gewünschte Funktionalität an der Nutzungsschnittstelle von Baustein A zur Verfügung steht. Die Bausteine B und C verfügen neben ihren Nutzungsschnittstellen über Managementschnittstellen, an denen durch geeignete Instrumentierung des jeweiligen Bausteins Managementinformation abgerufen werden kann. Aus der an diesen Managementschnittstellen verfügbaren Information soll nun die Managementinformation von Baustein A ermittelt und an dessen Managementschnittstelle angeboten werden. Es soll also – analog zur Anwendungslogik – eine Managementlogik angegeben werden, die die Managementinformationen der darunterliegenden Bausteine auf die gewünschte Managementinformation des zusammengesetzten Bausteins abbildet. Die darunterliegenden Bausteine B und C müssen dabei nicht unbedingt Basisbausteine sein, sondern können ihrerseits wiederum zusammengesetzt sein; ebenso ist die Zusammensetzung eines Bausteins aus mehr als zwei Bausteinen möglich.

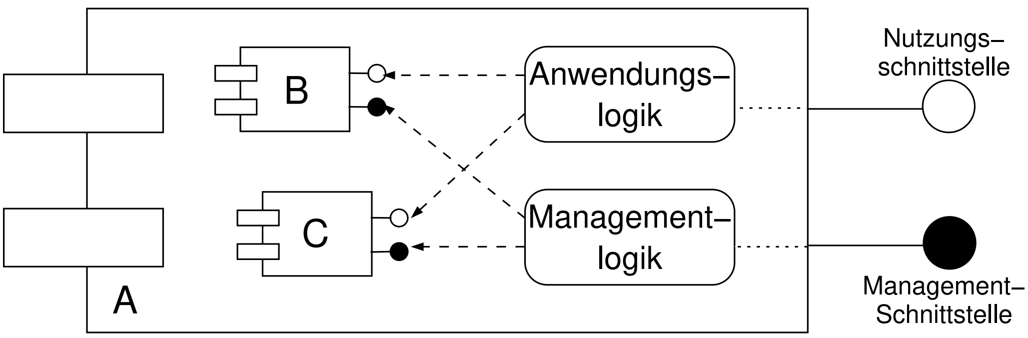

Abbildung 5.1: Komposition von Managementschnittstellen mittels Managementlogik

5.2.2 Verknüpfung von Managementinformation

Die Verknüpfung von Managementinformation (durch die Managementlogik) kann nicht für beliebige Informationen und für beliebige Anwendungen identisch erfolgen. Vielmehr muß für jede Art von Information ein Algorithmus angegeben werden, wie die Verknüpfung für diese Informationsart zu erfolgen hat. So kann es beispielsweise sinnvoll sein, bestimmte Managementinformation der darunterliegenden Bausteine aufzuaddieren, Maximal- oder Durchschnittswerte zu bestimmen oder die Vereinigungsmenge der Einzelinformationen zu bilden.

Ebenso kann die Art der Verknüpfung der Bausteine in den Algorithmus mit eingehen. Ruft ein Baustein eine Operation eines anderen Bausteins beispielsweise synchron auf, wird somit also bis zur Lieferung eines Ergebnisses blockiert, so kann dies eine andere Verknüpfung der zugehörigen Managementinformation bedingen als im Falle eines asynchronen Aufrufs. Der genaue Algorithmus der Managementlogik wird also von den folgenden zwei Faktoren wesentlich beeinflußt:

- Art der Managementinformation
- Art der Verknüpfung der Bausteine

Dies wird im folgenden noch einmal anhand eines einfachen Beispiels verdeutlicht: Gegeben sei (wie bereits im obigen Beispiel) ein Baustein A, der aus zwei Bausteinen B und C zusammengesetzt ist. Bei Aufruf einer Transaktion des Bausteins A sorgt die Anwendungslogik zunächst für den Aufruf des Bausteins B und dann für einen Aufruf des Bausteins C. Beide Bausteine führen beliebige Berechnungen aus. Die Aufrufe können sowohl synchron als auch asynchron ausgeführt werden. Abbildung 5.2 stellt zwei mögliche Sequenzdiagramme gegenüber. Links im Bild befindet sich der Ablauf des übergeordneten Bausteins A. Der mittlere Teil stellt synchrone Aufrufe von B und C dar, während der rechte Teil des Bildes asynchrone Aufrufe illustriert. Weiterhin sind auch beliebige Kombinationen von synchronen und asynchronen Aufrufen sowie Aufrufe von mehr als zwei

Bausteinen denkbar. Jeder der aufgerufenen Bausteine kann seinerseits wiederum Bausteine aufrufen. Dies kann ebenfalls synchron oder asynchron geschehen.

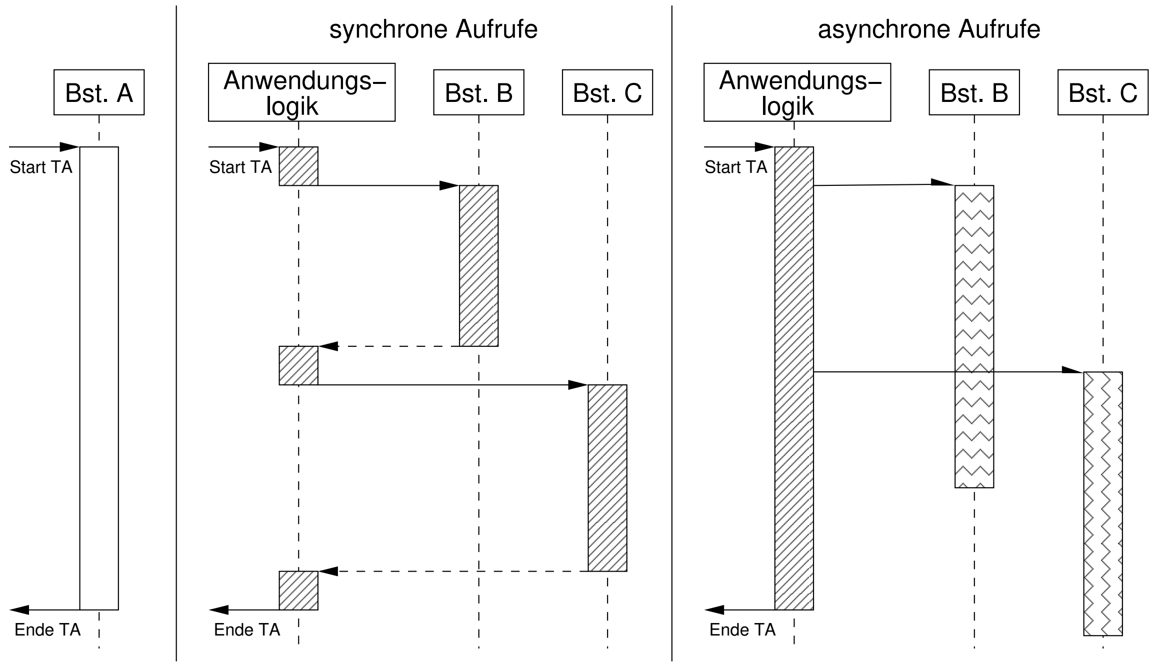

Abbildung 5.2: Sequenzdiagramme für synchrone und asynchrone Aufrufe von Bausteinen

Um die Verknüpfung der Managementinformation zu verdeutlichen, wird in folgenden angegeben, wie die in Kapitel 3 geforderte transaktionsbasierte Überwachung möglich wäre. Es wird also gezeigt, wie die Antwortzeit, die Transaktionsdauer sowie die Verfügbarkeit einzelner Bausteine aus den Informationen der darunterliegenden Bausteine zu errechnen ist.

- Antwortzeit
  Die Antwortzeit beschreibt die wahrnehmbare Dauer einer Transaktion (vgl. Abschnitt 2.2.2). Bei Benutzertransaktionen kann diese unter Umständen durch den Zeitpunkt der Ergebnispräsentation bestimmt werden. Handelt es sich aber um eine Transaktion ohne explizite Präsentation eines Ergebnisses, so ist die Antwortzeit bestimmt durch den Zeitpunkt der Rückkehr des Kontrollflusses zum Aufrufer.

  Nachdem im allgemeinen Fall keine explizite Präsentation eines Ergebnisses vorliegt, wird im folgenden ein Algorithmus angegeben, wie die Zeit bis zur Rückkehr des Kontrollflusses zum Aufrufer aus den Zeiten der darunterliegenden Bausteine errechnet werden kann. Abbildung 5.3 skizziert den resultierenden Algorithmus.

```
Antwortzeit(Baustein X) {
  IF (X instrumentiert)
     antwortzeit := MesseAntwortzeit();
  ELSE {
     antwortzeit := 0;
     FORALL(Y aus {synchron von X aufgerufene Bausteine})
          antwortzeit := antwortzeit + Antwortzeit(Y);
  }
  RETURN antwortzeit;
}
```

Abbildung 5.3: Algorithmus für die Berechnung der Antwortzeit

Handelt es sich um einen instrumentierten Baustein (einen Basisbaustein), so kann die Antwortzeit mit Hilfe der Instrumentierung einfach gemessen werden. Ist dies nicht der Fall, so kann die Antwortzeit durch Berechnung der Summe der Antwortzeiten aller synchron von diesem Baustein aufgerufenen Bausteine ermittelt werden. Asynchron aufgerufene Bausteine werden bei dieser Berechnung nicht berücksichtigt, da die im asynchron aufgerufenen Baustein durchgeführten Berechnungen nicht mit in die Antwortzeit des übergeordneten Bausteins eingehen. Die Zeit für den asynchronen Aufruf selbst muß bei dieser Vorgehensweise vernachlässigt werden.

- Transaktionsdauer

  Im Gegensatz zur Antwortzeit wird unter der Transaktionsdauer die Zeit bis zum vollständigen Ende aller aus einer Transaktion resultierenden Aktivitäten verstanden. Diese kann also aus der Differenz zwischen dem spätesten Ende einer resultierenden Subtransaktion und dem Beginn der Transaktion berechnet werden. Hierzu kann der in Abbildung 5.4 dargestellte Algorithmus verwendet werden.

  Mit Hilfe der Funktion `Start(Baustein X)` wird der Beginn der Transaktion ermittelt. Dieser kann entweder durch die Instrumentierung direkt ermittelt werden, oder wird durch den Startpunkt des ersten aufgerufenen Bausteins bestimmt. Um das Ende der Transaktion zu ermitteln, wird zunächst wiederum analog vorgegangen: Handelt es sich um einen instrumentierten Baustein, so kann das Ende einfach ermittelt werden. Ist dies nicht der Fall, so wird das Ende des zuletzt synchron aufgerufenen Bausteins bestimmt.

  Daraufhin wird für alle asynchron aufgerufenen Bausteine der Zeitpunkt des Endes der entsprechenden Subtransaktion bestimmt. Handelt es sich hierbei um einen späteren Zeitpunkt als den bereits ermittelten, so wird dieser als das Ende der Transaktion betrachtet. Die eigentliche Transaktionsdauer wird dann aus der Differenz zwischen ermitteltem Endzeitpunkt und ermitteltem Startzeitpunkt errechnet.

```
Transaktionsdauer(Baustein X) {
  RETURN (Ende(X) - Start(X));
}

Start(Baustein X) {
  IF (X instrumentiert)
    start := messeStart();
  ELSE
    start := Start(erster von X aufgerufener Baustein);
  RETURN start;
}

Ende(Baustein X) {
  IF (X instrumentiert)
    ende := messeEnde();
  ELSE {
    ende := Ende(letzter synchron von X aufgerufener Bst);
    FORALL (Y aus {asynchron von X aufgerufene Bste}) {
        IF (Ende(Y) > ende)
              ende := Ende(Y);
  }
  RETURN ende;
}
```

Abbildung 5.4: Algorithmus für die Berechnung der Transaktionsdauer

- Verfügbarkeit
  Die Bestimmung der Verfügbarkeit eines zusammengesetzten Bausteins läßt sich nicht vollständig automatisieren. Man kann davon ausgehen, daß ein Baustein nur dann verfügbar ist, wenn alle von ihm synchron aufgerufenen Bausteine ebenfalls verfügbar sind. Andernfalls würde der zusammengesetzte Baustein bei Aufruf des nicht verfügbaren Bausteins blockieren. Im Falle asynchroner Aufrufe kann diese Entscheidung allerdings nicht automatisch getroffen werden. In diesem Fall muß der Anwendungsentwickler bei der Entwicklung der Anwendung angeben, ob ein asynchron aufgerufener Bausteine in die Berechnung der Verfügbarkeit des übergeordneten Bausteins eingehen soll oder nicht. Der zugehörige Algorithmus ist in Abbildung 5.5 dargestellt.

  Wiederum kann die Verfügbarkeit direkt bestimmt werden, wenn es sich um einen instrumentierten Baustein handelt. Andernfalls wird die Verfügbarkeit zunächst auf

```
Verfügbarkeit (Baustein X) {
  IF (X instrumentiert)
    verfügbarkeit := messeVerfügbarkeit();
  ELSE
    verfügbarkeit := TRUE;
  FORALL (Y aus {synchron von X aufgerufene Bausteine})
    verfügbarkeit := verfügbarkeit AND Verfügbarkeit(Y);
  FORALL (Y aus {asynchron von X aufgerufene Bausteine})
    IF (Y von Anwendungsentwickler ausgewählt)
      verfügbarkeit := verfügbarkeit AND Verfügbarkeit(Y);
  RETURN verfügbarkeit;
}
```

Abbildung 5.5: Algorithmus für die Berechnung der Verfügbarkeit

`TRUE` gesetzt. Daran anschließend wird die Verfügbarkeit aller synchron aufgerufenen Bausteine überprüft, die mit Hilfe einer `AND`-Verknüpfung in die Berechnung eingeht. Die Verfügbarkeit asynchron aufgerufener Bausteine geht nur dann in die Berechnung mit ein, wenn dies vom Anwendungsentwickler zum Zeitpunkt der Anwendungserstellung so bestimmt wurde.

5.2.3 Erstellung der Managementlogik

Die oben dargestellten Algorithmen beschreiben die Managementlogik bausteinbasierter Anwendungen allgemein, sind jedoch alleine noch nicht ausreichend, um die Managementlogik einer konkreten Anwendung zu beschreiben. Desweiteren ist Information über die Struktur der zu überwachenden Anwendung erforderlich. So ist es beispielsweise für die oben beschriebenen Algorithmen erforderlich, für jeden Baustein der Anwendung die von ihm aufgerufenen Bausteine sowie die Reihenfolge dieser Aufrufe anzugeben. Weiterhin ist Information darüber erforderlich, ob es sich um synchrone oder asynchrone Aufrufe handelt. Für jeden asynchron aufgerufenen Baustein bedarf es der Information, ob dessen Verfügbarkeit in die Berechnung der Verfügbarkeit des übergeordneten Bausteins eingehen soll oder nicht.

Die Beschreibung dieser Information kann beispielsweise mit Hilfe eines Graphen erfolgen. Abbildung 5.6 stellt einen möglichen Strukturgraphen für eine einfache Beispielanwendung dar. Man erkennt, daß Baustein A zunächst Baustein B und dann Baustein C aufruft. Anhand der Attributierung der Kanten erkennt man, daß Baustein B asynchron aufgerufen wird, während der Aufruf

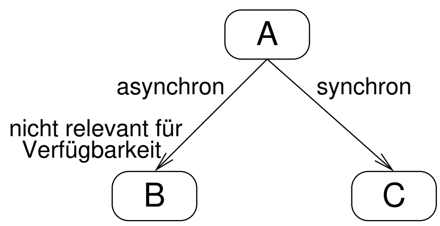

Abbildung 5.6: Strukturgraph einer einfachen Beispielanwendung

von Baustein C synchron erfolgt. Bei dem asynchronen Aufruf von Baustein B ist weiterhin zu erkennen, daß dessen Verfügbarkeit nicht in die Berechnung der Verfügbarkeit von Baustein A eingehen soll.

Die Erstellung dieses Strukturgraphen muß parallel zur Erstellung der Anwendung erfolgen. Fügt der Anwendungsentwickler eine Verknüpfung zweier Bausteine in die Anwendung ein, so muß der zugehörige Strukturgraph entsprechend erweitert werden. Dies kann in weiten Teilen automatisiert erfolgen. Lediglich bei Einfügen von asynchronen Aufrufen, muß der Anwendungsentwickler angeben, ob die Verfügbarkeit des aufgerufenen Bausteines in die Berechnung der Verfügbarkeit des übergeordneten Bausteins eingehen soll oder nicht.

5.2.4 Bewertung

Der vorgeschlagene Ansatz basiert auf einer Managementinstrumentierung der Basisbausteine. Die Abhängigkeiten zwischen den einzelnen Bausteinen eines zusammengesetzten Bausteins werden zum Zeitpunkt der Anwendungserstellung in einem Strukturgraphen statisch festgelegt. Die Korrelation der Informationen der darunterliegenden Bausteine erfolgt bereits im Baustein und nicht in einer speziellen Managementanwendung. Dieser Ansatz hat sich im Laufe der durchgeführten Untersuchungen als nur bedingt geeignet für die Überwachung bausteinbasierter Anwendungsdienste erwiesen. Dies liegt insbesondere an der statischen Beschreibung der Abhängigkeiten innerhalb der Anwendung bereits zum Zeitpunkt der Anwendungserstellung, aber auch an dem immer noch erheblichen Aufwand für Baustein- und Anwendungsentwickler und der eingeschränkten Flexibilität der Lösung. Im folgenden wird eine detaillierte Bewertung des vorgestellten ersten Ansatzes vorgenommen:

- Beschreibung der Abhängigkeiten
  Die Beschreibung der Abhängigkeiten und Zusammenhänge innerhalb der zu überwachenden Anwendung erfolgt bei diesem Ansatz statisch zum Zeitpunkt der Anwendungserstellung. Dies erweist sich allerdings als schwierig. Anwendungen durchlaufen z.B. je nach Benutzereingabe unterschiedliche Zweige und aus Gründen der Lastverteilung und Redundanz können mehrere Instanzen desselben Bausteins existieren. Es ist somit zum Zeitpunkt der Anwendungserstellung im allgemeinen nicht möglich, einen Strukturgraphen anzugeben, der die tatsächlichen Abläufe innerhalb der Anwendung hinreichend beschreibt. Diese Information kann erst zur Laufzeit der Anwendung ermittelt werden. Selbst wenn ein Graph angegeben werden kann, der alle Möglichkeiten beschreibt, wie die Anwendung durchlaufen werden könnte, ist es zur Laufzeit der Anwendung dennoch nicht möglich, den für eine konkrete Instanz einer Transaktion relevanten Weg durch den Graphen ohne Instrumentierung der Anwendungslogik zu bestimmen.

- Aufwand
  Durch die Verlagerung des Instrumentierungsaufwandes vom Anwendungsentwickler auf den Bausteinentwickler kann eine erhebliche Reduzierung des Gesamtaufwandes erreicht werden. Dies begründet sich mit der Möglichkeit der mehrfachen Verwendung von Bausteinen in unterschiedlichen Anwendungen, der Verringerung der Komplexität der einzelnen Instrumentierungsaufgaben und der teilweise automatisierbaren und werkzeuggestützten Erstellung der Managementlogik.

  Die Instrumentierung jedes einzelnen Basisbausteins stellt aber dennoch einen erheblichen zusätzlichen Aufwand für den Bausteinentwickler dar. Das Problem des hohen Instrumentierungsaufwandes, das eines der Haupthindernisse für eine rasche Verbreitung von Instrumentierungstechniken im Anwendungsmanagement darstellt, wird somit nicht gelöst, sondern nur verringert und vom Anwendungsentwickler auf den Bausteinentwickler verlagert.

  Auch die schwierige Aufgabe, die Abhängigkeiten der einzelnen Bausteine untereinander zu bestimmen, kann nicht vollständig automatisiert werden. Auch wenn in vielen Fällen die Zusammenhänge automatisch erkannt werden können, müssen dennoch in einer Vielzahl der Fälle Entscheidungen weiterhin vom Anwendungsentwickler getroffen werden.

- Verfügbare Information
  Durch die Instrumentierung der Bausteine ist es möglich, die geforderte Information vollständig zu ermitteln. Da Bausteine allerdings mehrere unterschiedliche Transaktionen erbringen können, wäre ein Ausweitung des Ansatzes von der Beschreibung der Verknüpfung von Bausteinen auf die Verknüpfung von Transaktionen eines Bausteins erforderlich.

- Zuordnung der Teilinformationen
  Sind zu einem bestimmten Zeitpunkt mehrere Transaktionen gleichzeitig aktiv, so ist bei einer Beschreibung der Abhängigkeiten zum Zeitpunkt der Anwendungserstellung nicht festzustellen, zu welcher der Transaktionen ein Baustein aktuell beiträgt. Somit kann die beschriebene Berechnung nur durchgeführt werden, wenn zu jedem Zeitpunkt nur eine Transaktion ausgeführt wird und keine parallelen Transaktionen existieren. Dies ist in der Praxis in den meisten Fällen so nicht gegeben.

  In vielen Fällen sind Bausteine nicht einer bestimmten Anwendung fest zugewiesen, sondern können von mehreren unterschiedlichen Anwendungen genutzt werden. Die Zuordnung von Managementinformation zu einer übergeordneten Anwendung erweist sich in diesem Fall als besonders schwierig. Die Bausteine müßten in der Lage sein, je nachdem, welcher der übergeordneten Bausteine eine Anfrage durchführt, die jeweils korrekte Information zu liefern. Dies würde zu einer erheblichen Erhöhung der Komplexität der Bausteine führen. Zustandslose Bausteine, die in vielen Bausteinarchitekturen explizit vorgesehen sind, könnten diese Art der Information ohnehin nicht liefern.

- Integration von Legacy-Bausteinen
  Eine Lösung, die eine spezielle Managementinstrumentierung aller Bausteine erfordert, eignet sich nicht für die Integration von *Legacy*-Bausteinen, also Bausteinen, die nicht über eine derartige Instrumentierung verfügen. Bausteine ohne entsprechende Schnittstelle können zwar weiterhin zum Einsatz kommen, die Einbeziehung dieser Bausteine in die Überwachung ist dann aber ausgeschlossen. Da in der Praxis nicht davon auszugehen ist, daß sämtliche eingesetzten Bausteine über eine derartige Schnittstelle verfügen werden, ist eine Lösung, die auch die (zumindest eingeschränkte) Überwachung nicht instrumentierter Bausteine gestattet, vorzuziehen.

- Verläßlichkeit der Information
  Insbesondere vor dem Hintergrund der Anwendungserstellung aus Bausteinen unterschiedlicher – evtl. sogar anonymer – Herkunft ist die Verläßlichkeit der gelieferten Information ein wesentlicher Aspekt. Die gewonnenen Daten werden unter anderem eingesetzt für Zwecke der Abrechnung, aber auch um diejenigen Bausteine zu ermitteln, die ursächlich für eine Fehlfunktion oder den Ausfall eines Dienstes oder einer bestimmten Transaktion sind. Es ist somit nicht auszuschließen, daß vom Baustein übermittelte Informationen z.B. eine häufigere Nutzung des Bausteines vortäuschen, um somit beispielsweise höhere Nutzungsgebühren fordern zu können. Ebenso ist es denkbar, daß eine bessere Performanz angegeben wird, um das Risiko zu verringern, daß der Baustein gegen einen leistungsfähigeren Baustein z.B. eines anderen Herstellers ausgetauscht wird.

- Flexibilität
  Eine Verdichtung der Managementinformation bereits im Baustein schränkt die Flexibilität der Lösung für die Managementsysteme ein. Einer Managementanwendung können somit evtl. nicht mehr die eigentlich benötigten Informationen zur Verfügung gestellt werden, obwohl diese ursprünglich von den darunterliegenden Bausteinen angeboten wurden.

- Präzision der Information
  Durch die Berechnung der Managementinformation aus den Informationen der darunterliegenden Bausteine macht man bei gewissen Informationen zwangsläufig Fehler. Dies begründet sich damit, daß die Anwendungslogik, die die einzelnen Bausteine miteinander verknüpft, nicht in die Berechnung mit eingeht, allerdings einen teilweise nicht unerheblichen Prozentsatz z.B. der Gesamtdauer einer Transaktion oder der verbrauchten Prozessorzeit einnimmt. So kann in dem oben beschriebenen Algorithmus für die Berechnung der Antwortzeit beispielsweise der asynchrone Aufruf eines Bausteins nicht berücksichtigt werden, auch wenn der Aufruf selbstverständlich eine

gewisse Zeit in Anspruch nimmt. Hier wäre es erforderlich, zusätzlich zur Instrumentierung der Bausteine, eine Instrumentierung der Anwendungslogik vorzunehmen, so daß die dort anfallenden Informationen ebenfalls mit in die Berechnung einbezogen werden können.

5.3 Zweiter Ansatz: Automation der Managementinstrumentierung bausteinbasierter Anwendungen

Die im ersten Ansatz vorgestellte Lösung kann zwar den Gesamtaufwand für die Managementinstrumentierung einer Anwendung verringern, insbesondere für die Bausteinentwickler entsteht aber dennoch ein unverhältnismäßig hoher Aufwand. Ebenso erschwert bzw. verhindert die statische Beschreibung der Abhängigkeiten bei diesem Ansatz eine Zuordnung, z.B. von Subtransaktionen zu ihren übergeordneten Transaktionen. Aus diesen Gründen wird im folgenden ein zweiter Ansatz beschrieben, der eine weitere erhebliche Reduktion des Aufwandes aller Beteiligten mit sich führt. Darüber hinaus erleichtert diese Variante die Integration von *Legacy*-Bausteinen, liefert verläßlichere und präzisere Informationen und bietet größtmögliche Flexibilität für Managementanwendungen bezüglich der Nutzung der gewonnenen Information. Die Bewertung des Ansatzes wird allerdings zeigen, daß nicht alle erforderlichen Informationen auf diese Weise zu ermitteln und nicht alle Arten von Bausteinen zu integrieren sind, weshalb letztlich eine Kombination der beiden Ansätze erforderlich ist.

5.3.1 Idee

Die grundsätzliche Idee des im folgenden vorgestellten Ansatzes besteht darin, auf eine Instrumentierung einzelner Bausteine vollständig zu verzichten und stattdessen eine automatische Instrumentierung der Anwendungslogik vorzunehmen. Somit müssen die Bausteine bei diesem Ansatz neben ihrer Nutzungschnittstelle keine spezielle Managementschnittstelle zur Verfügung stellen. Weiterhin entfällt die Erfordernis, eine explizite Managementlogik zu generieren. Diese wird stattdessen in die Anwendungslogik integriert. Der Strukturgraph der Anwendung wird somit jeweils zur Laufzeit einer Transaktion dynamisch aus den tatsächlich auftretenden Abläufen ermittelt.

Wie bereits dargestellt, erfolgt der Aufruf eines Bausteins nie direkt aus einem anderen Baustein heraus, sondern immer mittelbar über die Anwendungslogik. Es ist somit möglich, durch Einfügen von Meßpunkten in die Anwendungslogik, Informationen über

Aufrufe einzelner Bausteine zu ermitteln, ohne daß hierzu eine Instrumentierung der Bausteine selbst erforderlich wäre.

Geht man davon aus, daß die Anwendungslogik von einem Anwendungsentwickler mit Hilfe einer speziellen Entwicklungsumgebung erzeugt und teilweise von dieser generiert wird, ist es möglich, die Meßpunkte automatisch in die Anwendungslogik einzubringen. Hierzu ist es erforderlich, die Entwicklungsumgebung geeignet zu erweitern, so daß sie zusätzlich zum *Code*, der die eigentliche Verknüpfung zweier Bausteine realisiert, Meßpunkte in die Anwendungslogik einfügt.

Im Gegensatz zur vorher vorgestellten Lösung werden bei dieser Variante die Benutzertransaktionen vom Anwendungsentwickler explizit definiert und können somit präzise gemessen werden. Darüber hinaus wird, wie bereits erwähnt, Information über jeden Aufruf eines Bausteins gemessen. Durch geeignete Korrelation der beteiligten Kontrollflüsse läßt sich dann die Zuordnung einer Benutzertransaktion zu den an ihrer Erbringung beteiligten Bausteinen herstellen. Zu beachten ist, daß nicht (wie im vorher vorgestellten Ansatz) eine explizite Managementlogik erstellt wird, sondern daß Meßpunkte in die Anwendungslogik eingefügt werden, deren Zusammenhang zur Laufzeit dynamisch ermittelt werden kann.

Abbildung 5.7 veranschaulicht dies nochmals anhand eines einfachen Beispiels. A und B seien beliebige Bausteine, die über keinerlei spezielle Managementinstrumentierung verfügen müssen. Ein Anwendungsentwickler erstellt mit Hilfe einer Entwicklungsumgebung und unter Verwendung der Bausteine A und B den Baustein C, dessen Anwendungslogik bei Aufruf z.B. für eine sequentielle Ausführung der beiden darunterliegenden Bausteine sorgt. Die Entwicklungsumgebung erstellt die Anwendungslogik und verknüpft die Bausteine entsprechend. Außerdem fügt sie automatisch Meßpunkte in die Anwendungslogik ein, die jeweils vor dem Aufruf eines Bausteins bzw. nach Rückkehr aus einem Baustein Managementinformationen liefern können. Es ergibt sich somit eine instrumentierte Anwendung C, deren Instrumentierung ohne manuelles Eingreifen des Baustein- bzw. Anwendungsentwicklers vorgenommen werden konnte. Die Instrumentierung erlaubt es beispielsweise, die Aufrufhäufigkeiten der einzelnen Bausteine, die Zeitdauer eines Aufrufs oder das Verhältnis erfolgreicher zu nicht erfolgreichen Aufrufen zu bestimmen.

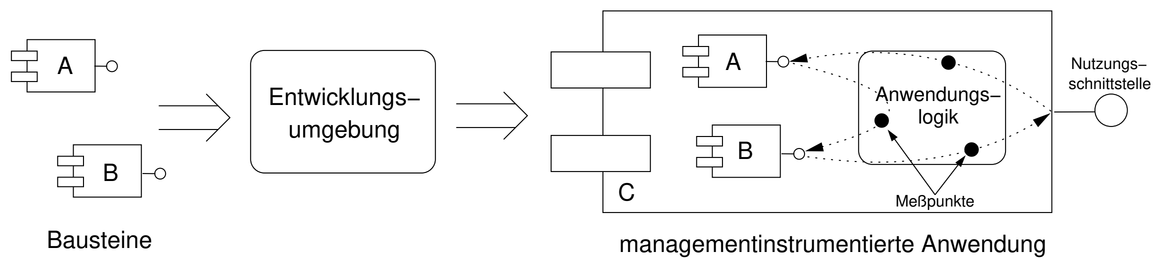

Abbildung 5.7: Automation der Managementinstrumentierung bausteinbasierter Anwendungen

Im folgenden Abschnitt wird dargestellt, wie die Identifikation und Generierung geeigneter Meßpunkte mit möglichst geringem Aufwand erfolgen kann. In Abschnitt 5.3.3 wird dann gezeigt, wie durch Korrelation von Kontrollflüssen eine eindeutige Zuordnung von Meßwerten zu Benutzertransaktionen möglich ist. Der daran anschließende Abschnitt gibt eine detaillierte Bewertung der vorgeschlagenen Lösung und geht insbesondere auf verbleibende Schwachpunkte ein.

5.3.2 Identifikation von Meßpunkten

Die Lösung basiert auf der Überwachung von Transaktionen. Es werden sowohl die vom Benutzer als solche wahrnehmbaren Transaktionen, die sogenannten Benutzertransaktionen überwacht als auch die Transaktionen der einzelnen darunterliegenden Bausteine, die Subtransaktionen. Es soll jeweils die Antwortzeit und die Transaktionsdauer sowie der Erfolg bzw. Mißerfolg der einzelnen Transaktionen gemessen werden können.

Dieser Abschnitt stellt dar, wie die Identifikation geeigneter Meßpunkte innerhalb einer bausteinbasierten Anwendung für die Überwachung von Transaktionen mit möglichst geringem Aufwand erfolgen kann. Zunächst wird angegeben, wie der Beginn und das Ende von Benutzertransaktionen identifiziert werden kann; im Anschluß daran wird gezeigt, wie die Generierung von Meßpunkten für die Überwachung der einzelnen Subtransaktionen vollständig automatisiert erfolgen kann.

5.3.2.1 Überwachung von Benutzertransaktionen

Unter einer Benutzertransaktion (BTA) wird eine am Dienstzugangspunkt erbrachte, in sich abgeschlossene und vom Dienstnutzer als solche wahrnehmbare Aufgabe verstanden (vgl. Abschnitt 2.2.2). Im Falle der Erbringung von BTAs unter Verwendung bausteinbasierter Anwendungen stellt sich die Situation wie in Abbildung 5.8 skizziert dar. Die Entgegennahme der BI und damit der Anstoß der BTA kann ausschließlich über Eingabebausteine erfolgen. Hier ist es möglich, daß unterschiedliche BIs bzw. Eingabebausteine existieren, die dieselbe BTA anstoßen können. Die eigentliche Aktivität besteht aus einer beliebigen Verknüpfung weiterer (*Server*)-Bausteine. Diese können über mehrere Systeme verteilt sein. Zu einem beliebigen Zeitpunkt während der Erbringung der Aktivität erfolgt – wiederum durch eine BI – die Präsentation des Ergebnisses mit Hilfe eines Präsentationsbausteins. Wiederum ist vorstellbar, daß mehrere unterschiedliche Präsentationsbausteine existieren, die das Ende einer BTA darstellen können, abhängig vom Verlauf der BTA, von bestimmten Benutzerwünschen oder sonstigen Gegebenheiten. Das tatsächliche Ende der ausgeführten Aktivität ist unabhängig vom Zeitpunkt der Ergebnispräsentation.

Um BTAs überwachen zu können, ist es erforderlich, Meßpunkte in die Anwendung einzubringen, die den Beginn und das Ende jeder BTA an das Managementsystem übermitteln. Die Identifikation von Benutzertransaktionen kann nicht vollständig automatisiert

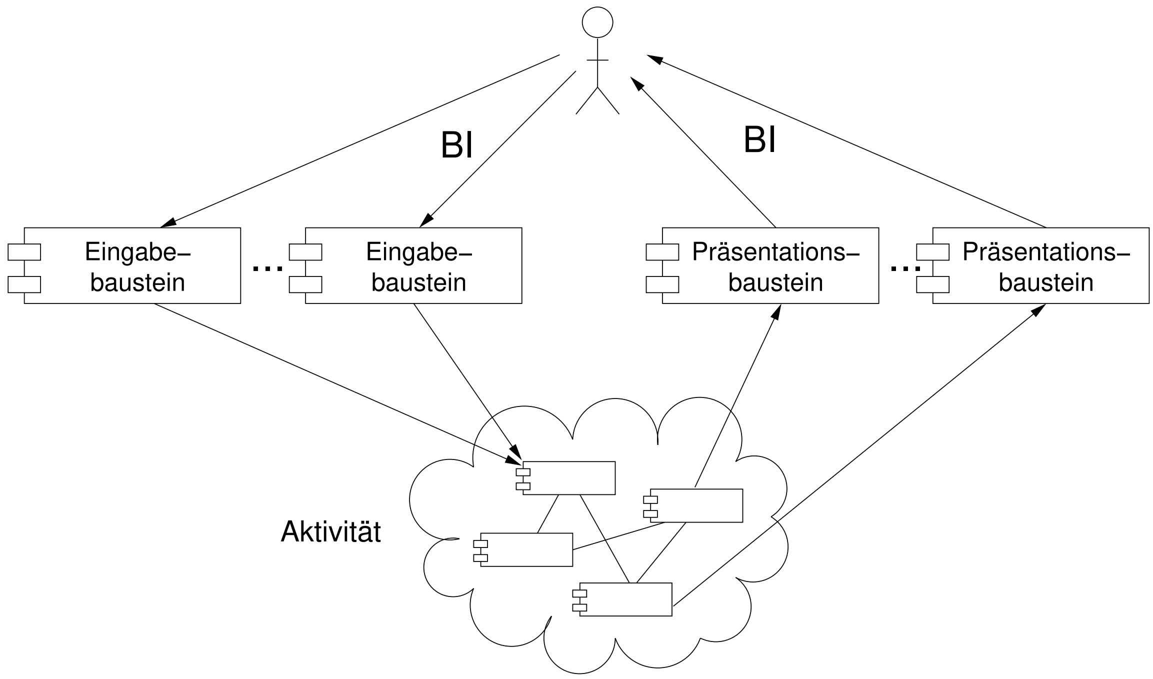

Abbildung 5.8: BTA eines bausteinorientierten Anwendungsdienstes

erfolgen, da jede BI eines Eingabebausteins theoretisch beliebige BTAs anstoßen kann. Die Zuordnung von BIs zu bestimmten BTAs kann daher erst vom Anwendungsentwickler zum Zeitpunkt der Anwendungserstellung angegeben werden. Die folgenden Abschnitte beschreiben, wie diese Zuordnung mit möglichst geringem Aufwand für den Anwendungsentwickler stattfinden kann.

Beginn einer Benutzertransaktion

Eine vom Benutzer initiierte BI kann nur auf den Eingabebausteinen der Anwendung getätigt werden. Beispiele für derartige BIs sind die Betätigung der `RETURN`-Taste nach der Eingabe von Information oder das Anklicken eines bestimmten *Buttons* auf der graphischen Benutzeroberfläche einer Anwendung.

Zum Zeitpunkt der Anwendungserstellung sind dem Anwendungsentwickler diejenigen BIs bekannt, die als Beginn zu überwachender BTAs betrachtet werden sollen. Aufgabe des Anwendungsentwicklers ist es nun, diese BIs zu identifizieren und die zugehörige BTA (innerhalb der Anwendung) eindeutig zu benennen.

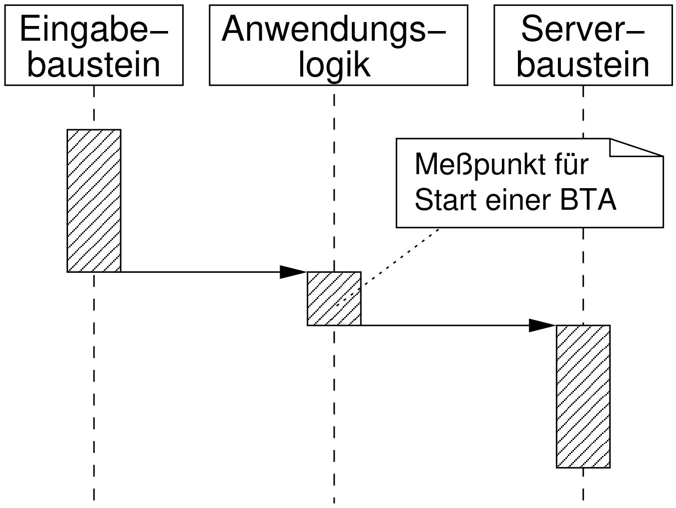

Abbildung 5.9: Identifikation des Meßpunktes für den Beginn einer BTA

Geht man davon aus, daß ein Eingabebaustein neben der Entgegennahme der BI keine nennenswerten eigenen Berechnungen ausführt, sondern unmittelbar einen *Server*-Baustein aufruft, dann kann der Aufruf dieses *Server*-Bausteins als der Beginn der BTA betrachtet werden, ohne hierdurch die Meßgenauigkeit zu stark einzuschränken. Die Entwicklungsumgebung muß also die Möglichkeit bieten, beim Einfügen einer Verknüpfung eines Eingabebausteins mit einem weiteren Baustein die erstellte Verknüpfung als den Beginn einer BTA zu definieren. Sie muß daraufhin einen entsprechenden Meßpunkt in die generierte Anwendungslogik einfügen. Abbildung 5.9 illustriert dies nochmals anhand eines Sequenzdiagramms.

Ende einer Benutzertransaktion

Es ist erforderlich, sowohl den Zeitpunkt der Ergebnispräsentation (zur Berechnung der Antwortzeit) als auch das tatsächliche Ende der Bearbeitung der BTA (zur Berechnung der Transaktionsdauer) zu ermitteln. Dies muß allerdings auf unterschiedliche Weise erfolgen:

- Bestimmung der Antwortzeit

  Analog zum oben dargestellten Verfahren, den Beginn einer BTA zu identifizieren, kann in der umgekehrten Richtung vorgegangen werden, um den Zeitpunkt der Präsentation des Ergebnisses zu ermitteln: Nur wenige Bausteine einer Anwendung, die Präsentationsbausteine, geben Ergebnisse auf dem Bildschirm aus. Dies erfolgt z.B. durch den Aufruf einer Systemroutine, die ein Neuzeichnen des Bildschirm-/Fensterinhalts veranlaßt.

  Die Präsentation des Ergebnisses einer BTA wird im allgemeinen in einem anderen Baustein stattfinden als ihr Beginn; es ist aber davon auszugehen, daß diese Bausteine innerhalb desselben System ablaufen, da ein Benutzer die Reaktion auf eine Eingabe üblicherweise auf dem System erwartet, auf dem er die Eingabe vorgenommen hat. Somit ist die lokale Systemzeit ausreichend, um die Antwortzeit der BTA zu bestimmen.

  Eine einfache Instrumentierung der Systemroutine, die den neuen Bildschirmaufbau durchführt, ist allerdings nicht ausreichend, da diese nicht nur bei Abschluß einer

BTA, sondern auch zu jedem beliebigen anderen Zeitpunkt während der Ausführung der Transaktion aufgerufen werden kann. So können beispielsweise Zwischenergebnisse präsentiert werden oder der Benutzer kann manuell einen neuen Bildschirmaufbau veranlassen. Dies hat zur Folge, daß die Instrumentierung vor Aufruf der entsprechenden Routine erfolgen muß, also zu einem Zeitpunkt, zu dem definitiv bekannt ist, ob das endgültige Ergebnis der BTA präsentiert wird oder nicht. Somit kann also nicht der exakte Zeitpunkt gemessen werden, zu dem das Ergebnis dem Benutzer auf dem Bildschirm präsentiert wird; Geht man aber davon aus, daß die Zeit, die bis zum tatsächlichen Neuaufbau des Bildschirms verstreicht, im Vergleich zur Gesamtdauer der BTA vernachlässigbar klein ist, so kann diese Einschränkung in Kauf genommen werden (eine Lösung, die diese Voraussetzung nicht erfordert, wird in Abschnitt 5.4.1.2 vorgestellt).

Der Zeitpunkt der Präsentation des Ergebnisses einer BTA in einer konkreten Anwendung kann also ebenfalls nur durch den Anwendungsentwickler festgelegt werden. Wie bereits erwähnt ist es möglich, daß mehrere unterschiedliche BIs als Ergebnis derselben BTA in Frage kommen, beispielsweise bei Abbruch der BTA durch den Benutzer oder bei nicht erfolgreicher Ausführung der Transaktion.

Zur Bestimmung der Antwortzeit der BTA kann der Aufruf des Präsentationsbausteins verwendet werden. Geht man auch hier davon aus, daß im eigentlichen Präsentationsbaustein keine wesentlichen Berechnungen ausgeführt werden, sondern ausschließlich die Präsentation des Ergebnisses erfolgt, so kann der Zeitpunkt des Aufrufs des Präsentationsbausteins als guter Näherungswert für den Zeitpunkt der tatsächlichen Präsentation des Ergebnisses betrachtet werden. Dies wird in Abbildung 5.10 nochmals dargestellt.

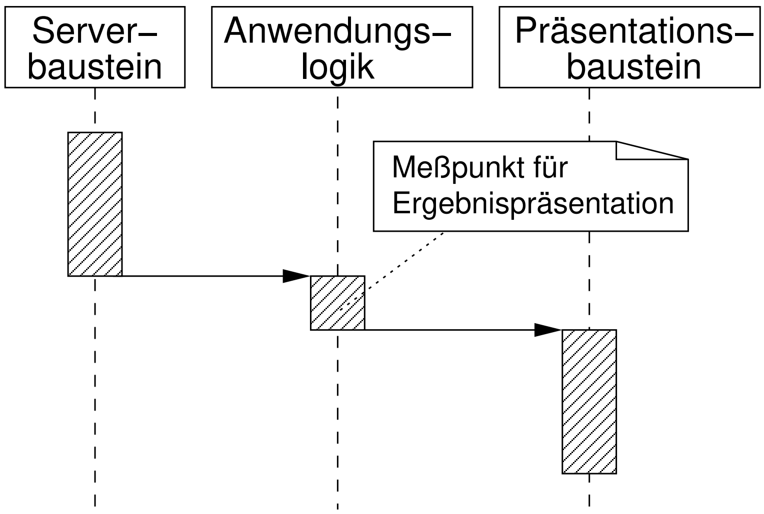

Abbildung 5.10: Identifikation des Meßpunktes für die Ergebnispräsentation

Unter Umständen ist es dem Anwendungsentwickler nicht möglich, anzugeben, welche BI das abschließende Ergebnis der BTA präsentiert. Dies kann z.B. der Fall sein, wenn kein eindeutiges Ergebnis der BTA existiert, sondern stattdessen parallel laufende Subtransaktionen jeweils Teilergebnisse präsentieren. In diesem Fall muß der späteste Zeitpunkt der Präsentation eines Teilergebnisses zur Bestimmung der Antwortzeit verwendet werden. Da zum Zeitpunkt der Anwendungserstellung die Reihenfolge der Teilergebnisse nicht immer sicher vorhergesagt werden kann, muß der Zeitpunkt sämtlicher, in Frage kommender Teilergebnisse, übermittelt werden. Nachdem wiederum davon auszugehen ist, daß alle Teilergebnisse auf demselben System präsentiert werden (dem System, auf dem die BTA angestoßen wurde), ist die Bestimmung des spätesten Zeitpunktes eines Teilergebnisses wiederum problemlos mit Hilfe der lokalen Systemzeit möglich.

Die Entwicklungsumgebung muß also weiterhin die Möglichkeit anbieten, Aufrufe von Präsentationsbausteinen als Zeitpunkt der Präsentation eines Ergebnisses einer BTA identifizieren zu können. In die generierte Anwendungslogik muß wiederum ein entsprechender Meßpunkt eingefügt werden. Um den Erfolg bzw. Mißerfolg der BTA erkennen zu können, muß der Anwendungsentwickler über die Möglichkeit verfügen zu definieren, ob der entsprechende Aufruf eines Präsentationsbausteins ein erfolgreiches oder nicht erfolgreiches Ende der BTA darstellt.

- Bestimmung der Transaktionsdauer

  Das tatsächliche Ende einer BTA zur Bestimmung der Transaktionsdauer kann im Gegensatz zum Zeitpunkt der Präsentation des Ergebnisses ohne explizites Eingreifen des Anwendungsentwicklers bestimmt werden: Eine BTA ist dann endgültig beendet, wenn alle an ihrer Erbringung beteiligten Kontrollflüsse beendet wurden oder die Erbringung der BTA anderweitig verlassen haben.

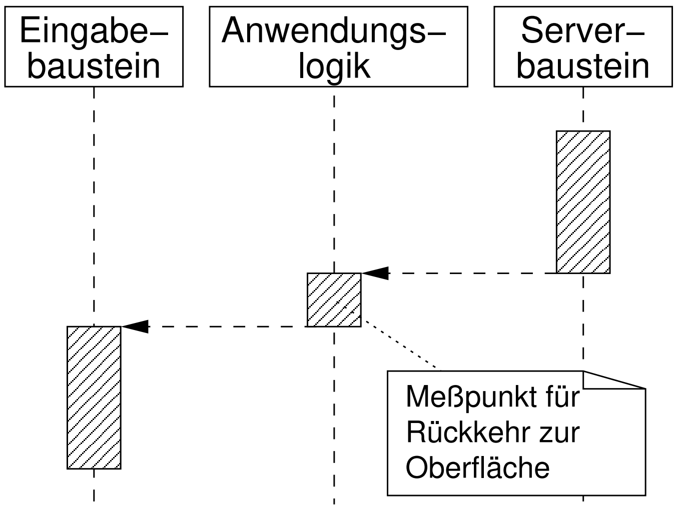

Abbildung 5.11: Identifikation des Meßpunktes für die Rückkehr zur Oberfläche

Es muß also ein Meßpunkt in die Anwendungslogik eingefügt werden, der die Rückkehr des Kontrollflusses zur Oberfläche kennzeichnet. Ähnlich wie bei der Ermittlung des Beginns einer BTA kann hier die Rückkehr des Kontrollflusses zum Eingabebaustein als guter Näherungswert für die Rückkehr zur Oberfläche betrachtet werden. Dies wird durch den Ausschnitt eines Sequenzdiagramms in Abbildung 5.11 nochmals veranschaulicht.

Wurden in die Bearbeitung der BTA weitere Kontrollflüsse mit einbezogen, so muß durch geeignete Instrumentierung der relevanten Bibliotheken des Systems (siehe Abschnitt 5.4.2.1) das Managementsystem über die jeweils aktuell beteiligten Kontrollflüsse informiert werden. Somit kann entschieden werden, wann der letzte Kontrollfluß die Bearbeitung der BTA verlassen hat.

5.3.2.2 Überwachung von Subtransaktionen

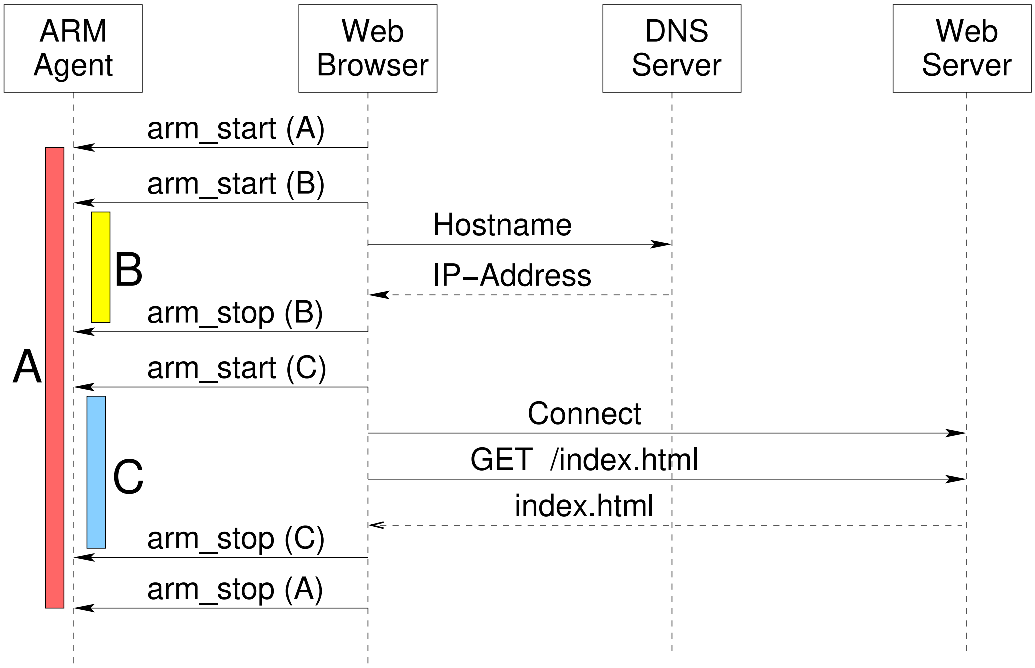

Abbildung 5.12: ARM-Instrumentierung eines Webbrowsers (nach [HaRe 00])

Um dem Diensterbringer die Möglichkeit zu geben, die Ursache fehlerhafter oder (zu) langsamer BTAs zu bestimmen, ist es erforderlich, diese in Subtransaktionen aufzuteilen. Die Subtransaktionen müssen dann ihren jeweiligen BTAs zugeordnet werden. Dies wird in Abbildung 5.12 nochmals anhand des Beispiels einer einfachen ARM-Instrumentierung eines *Webbrowsers* verdeutlicht [HaRe 00]. Transaktion A stellt die gesamte BTA des Herunterladens einer Webseite dar. Diese wird weiter in zwei Subtransaktionen B und C unterteilt, die die Auflösung der IP-Adresse des *Webservers* bei einem *DNS Server* bzw. den tatsächlichen *Download* der Seite vom *Webserver* umfassen. Somit ist es möglich, bei Auftreten eines Problems die Ursache schnell einzugrenzen und die Fehlerbehebung zu beschleunigen.

Wie in Abschnitt 3.2 bereits dargestellt, ist es im Falle bausteinbasierter Anwendungen sinnvoll und erforderlich, die von den jeweiligen Bausteinen ausgeführten Operationen als Subtransaktionen zu betrachten und zu überwachen. Somit ergeben sich die erforderlichen Meßpunkte folgendermaßen: Die Entwicklungsumgebung muß jeweils unmittelbar vor dem Aufruf eines Bausteins und unmittelbar nach Beendigung der Bearbeitung eines Aufrufs durch einen Baustein einen entsprechenden Meßpunkt in die Anwendungslogik einfügen.

Im Fall synchroner Aufrufe von Bausteinen ist dies vollständig automatisierbar. Jeweils unmittelbar vor dem Aufruf und unmittelbar nach dem Aufruf eines Bausteins muß ein Meßpunkt in den generierten *Code* eingefügt werden (vgl. Abbildung 5.13). Die erfolgreiche bzw. nicht erfolgreiche Bearbeitung des Aufrufes kann von der Anwendungslogik durch das Abfangen evtl. auftretender *Exceptions* erkannt werden. Tritt eine *Exception* auf, so ist von einer nicht ordnungsgemäßen Bearbeitung des Aufrufes auszugehen. Um den Programmablauf nicht zu stören muß die *Exception* anschließend an den ursprünglich aufrufenden Baustein übergeben werden.

Werden Bausteine asynchron aufgerufen, so sind wiederum Meßpunkte unmittelbar vor dem Aufruf und unmittelbar nach dem Verlassen des Bausteins erforderlich. Um in diesem Fall Fehlersituationen erkennen zu können, muß vom Anwendungsentwickler beim Einfügen einer Verknüpfung allerdings explizit angegeben werden, ob der gewählte Ausgang eine erfolgreiche oder eine fehlerhafte Bearbeitung des Aufrufs repräsentiert. Die Entwicklungsumgebung muß in diesem Fall also die Möglichkeit vorsehen, beim Einfügen beliebiger Verknüpfungen einen entsprechenden Parameter angeben zu können.

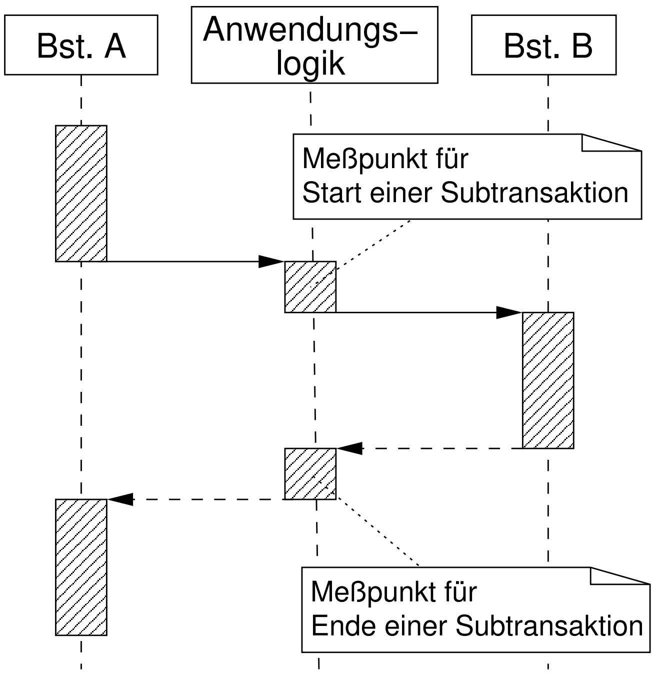

Abbildung 5.13: Identifikation der Meßpunktes für Subtransaktionen

5.3.3 Zuordnung der Teilinformationen

Im vorangegangenen Abschnitt wurde dargestellt, wie die erforderlichen Meßpunkte innerhalb einer Anwendung mit geringem Aufwand für den Anwendungsentwickler größtenteils automatisiert in die Anwendungslogik eingebracht werden können. Durch Ausnutzung der – ohnehin vorhandenen – Information über die beteiligten Kontrollflüsse wird eine eindeutige Zuordnung der einzelnen Meßwerte zu den jeweiligen Instanzen von BTAs möglich. Die vorgestellte Lösung basiert auf der Beobachtung, daß die Bearbeitung einer Instanz einer BTA innerhalb eines einzelnen Kontrollflusses geschieht bzw. eine Hinzunahme weiterer Kontrollflüsse sicher festgestellt werden kann. Hierfür ist kein Eingreifen des Anwendungsentwicklers erforderlich.

5.3.3.1 Zuordnung innerhalb eines Kontrollflusses

Im einfachsten Fall findet die komplette Bearbeitung einer BTA innerhalb eines Kontrollflusses statt. In diesem Fall muß bei Aufruf eines Bausteins der aufrufende Baustein blockieren, da der aufgerufene Baustein im Kontrollfluß des Aufrufers ausgeführt wird (siehe Abbildung 5.14). Es sind somit also ausschließlich synchrone Aufrufe möglich. In diesem Fall läßt sich die Zuordnung der einzelnen Meßwerte zur jeweiligen Transaktion problemlos wie in den beiden folgenden Absätzen beschrieben erreichen.

Zuordnung von Meßwerten zu Benutzertransaktionen

Die Zuordnung eines Meßwertes zu einer Instanz einer BTA kann mit Hilfe des Identifikators des Kontrollflusses geschehen. Dieser muß beim Start einer BTA bestimmt werden. Alle folgenden Meßwerte desselben Kontrollflusses sind dann derselben Instanz der BTA zuzuordnen. Das Ende der Instanz der BTA läßt sich wie bereits beschrieben durch Rückkehr des Kontrollflusses zur Oberfläche oder Beendigung des Kontrollflusses ermitteln. Auf diese Weise ist es insbesondere auch möglich, alle Subtransaktionen ihrer zugehörigen BTA zuzuordnen.

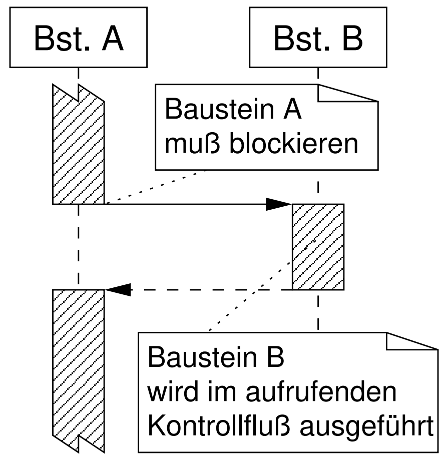

Abbildung 5.14: Aufruf innerhalb eines Kontrollflusses

Zuordnung von Meßwerten zu Subtransaktionen

Die Zuordnung von Meßwerten zu Subtransaktionen kann anhand der Reihenfolge der Meßwerte erfolgen. Jeder empfangene Meßwert kann vom Managementsystem der zuletzt gestarteten (und noch nicht beendeten) Subtransaktion zugeordnet werden. Dies liegt daran, daß innerhalb eines Kontrollflusses (aufgrund der Unmöglichkeit asynchroner Aufrufe) die Reihenfolge der Übermittlung der Meßwerte an das Managementsystem zugesichert werden kann. Es kann also beispielsweise ausgeschlossen werden, daß sich ein empfangener Meßwert auf eine bereits abgeschlossene Subtransaktion bezieht.

5.3.3.2 Zuordnung bei Einbeziehung mehrerer Kontrollflüsse

Die Zuordnung der Teilinformation bei Einbeziehung mehrerer Kontrollflüsse wird am Beispiel der Zuordnung von Subtransaktionen zu BTAs dargestellt. Die Zuordnung von Meßwerten zu BTAs bzw. zu Subtransaktionen kann analog erfolgen. Die vorgeschlagene Lösung erlaubt insbesondere die Zuordnung der Meßwerte von Subtransaktionen für alle in Abschnitt 3.2.2.3 geforderten Varianten der Verknüpfung zweier Bausteine (synchrone und asynchrone Operationsaufrufe, *Events*). Der folgende Abschnitt stellt zunächst nochmals die Problematik der Zuordnung von Subtransaktionen dar und zeigt, warum die Einbeziehung der Information über Kontrollflüsse für die sinnvolle Zuordnung erforderlich ist.

Problematik der Zuordnung von Subtransaktionen

Ziel bei der Zuordnung von Subtransaktionen ist es, jede Subtransaktion derjenigen Instanz einer BTA zuzuordnen, zu der die jeweilige Subtransaktion beiträgt. Im folgenden Abschnitt soll die daraus resultierende Problematik verdeutlicht werden, indem zunächst die aufwendige Herangehensweise der *ARM API* dargestellt wird. Anschließend wird gezeigt, daß auch die im vorangegangenen Abschnitt verwendete Beschreibung der statischen Verknüpfungsstruktur einer Anwendung nicht ausreichend wäre, um die gewünschte Zuordnung herzustellen.

Bei Einsatz der *ARM API* (vgl. Abschnitt 4.2.1.3) wird die Zuordnung von Subtransaktionen zu BTAs folgendermaßen erreicht (vgl. Abbildung 5.15): Nach Start einer Transak-

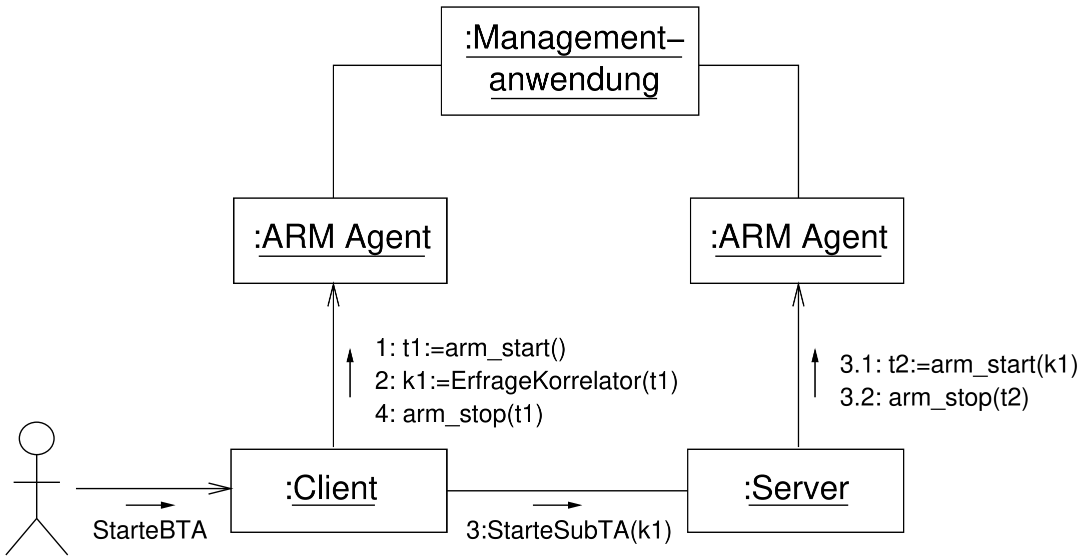

Abbildung 5.15: Zuordnung von Subtransaktionen bei Verwendung der ARM API (nach [John 97])

tion `t1` wird vom ARM Agent ein sogenannter Korrelator generiert und kann dort erfragt werden(`k1`), der die Instanz der Transaktion eindeutig identifiziert. Dieser muß bei einer nachfolgenden Subtransaktion (`t2`) wieder an den jeweiligen ARM Agenten übergeben werden und gestattet somit die Zuordnung der Transaktionen in einer angeschlossenen Managementanwendung. Problem hierbei ist aber wiederum der erhebliche zusätzliche Aufwand für den Anwendungsentwickler, da der Korrelator der Transaktionsinstanz innerhalb der Anwendung – im Beispiel sogar über Systemgrenzen hinweg – propagiert werden muß (z.B. als zusätzlicher Aufrufparameter). Dies verhindert meist eine nachträgliche Instrumentierung von Anwendungen, da hierfür erhebliche Eingriffe in den *Source Code* erforderlich wären. Im Falle bausteinbasierter Anwendungserstellung ist eine derartige Vorgehensweise gänzlich ausgeschlossen, da eine Propagierung dieser Information über Bausteingrenzen hinweg nicht ohne Veränderung der Schnittstellen aller eingesetzter Bausteine möglich wäre.

Auch aufgrund der Kenntnis der Verknüpfungsstruktur einer bausteinbasierten Anwendung läßt sich die Zuordnung nicht ermitteln. Dies wird an zwei einfachen Beispielen deutlich: Abbildung 5.16 zeigt vereinfacht eine Anwendung, die aus vier Bausteinen besteht. Bausteine A und B sind Eingabebausteine, mit denen `BTA 1` bzw. `BTA 2` gestartet werden. Beide rufen daraufhin Baustein C auf, der seinerseits Baustein D aktiviert. In diesem einfachen Beispiel ist es bereits nicht mehr möglich, anhand der Verknüpfungsstruktur der Anwendung die von Baustein D erbrachte Subtransaktion einer der beiden BTAs korrekt zuzuordnen.

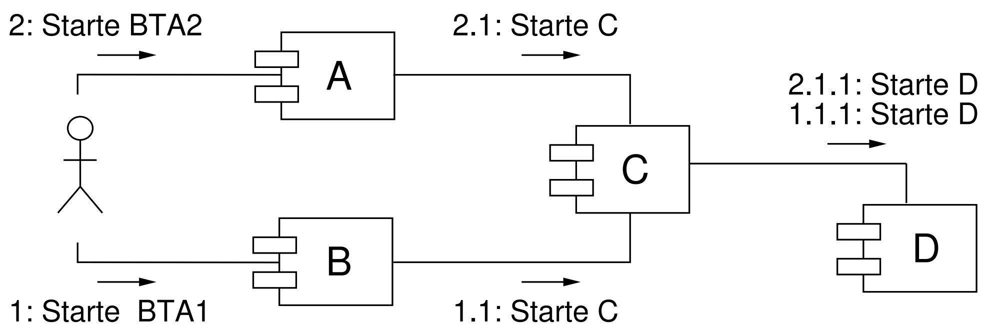

Abbildung 5.16: Problematik der Zuordnung von Subtransaktionen zu BTAs

Eine ähnliche Situation ergibt sich bei zeitlich verschränkt ablaufenden Instanzen derselben BTA. `BTA 1a` und `BTA 1b` seien unterschiedliche Instanzen derselben BTA. Beide rufen daher denselben Baustein B auf. Da die Reihenfolge der Bearbeitung durch Baustein B nicht garantiert werden kann, ist es auch in diesem Beispiel nicht möglich, aufgrund der Verknüpfungsstruktur der Anwendung die beiden Instanzen der von Baustein B erbrachten Subtransaktion korrekt der zugehörigen Instanz der BTA zuzuordnen.

Zuordnung anhand von Kontrollflüssen

Der oben dargestellten Problematik kann wiederum durch Zuordnung anhand von Kontrollflüssen begegnet werden. Jede BTA beginnt sicher innerhalb eines einzelnen Kontrollflusses. Dies ist derjenige Kontrollfluß, in dem die Oberfläche der Anwendung ausgeführt wird. Je nach Art und Umfang der gestarteten BTA kann die komplette Bearbeitung innerhalb dieses einen Kontrollflusses geschehen, es können aber auch weitere Kontrollflüsse in die Bearbeitung einbezogen werden.

Im einfachsten Fall, der Erbringung der kompletten BTA innerhalb eines einzigen Kontrollflusses ist es – wie bereits beschrieben – problemlos möglich, die Zuordnung zwischen Subtransaktionen und der zugehörigen BTA anhand des Identifikators des Kontrollflusses herzustellen. Dies liegt an der Tatsache, daß innerhalb eines Kontrollflusses zu jedem Zeitpunkt nur eine BTA ausgeführt werden kann.

Wird die BTA unter Verwendung mehrerer Kontrollflüsse erbracht, so ist ebenfalls eine Zuordnung möglich. Hierfür muß dem Managementsystem lediglich die Hinzunahme jedes weiteren Kontrollflusses in die Bearbeitung einer BTA bekannt gemacht werden. Hierbei ist aber die korrekte Reihenfolge der Bearbeitung einzelner Aufrufe entscheidend. Ein andernfalls möglicherweise auftretendes Problem ist in Abbildung 5.17 dargestellt: Ein Benutzer stoße eine BTA an (`StartBTA1`), die mit der Ausführung von Baustein A im Kontrollfluß der Benutzeroberfläche beginnt. Baustein A startet asynchron einen weiteren Kontrollfluß, in dem Baustein B zur Ausführung kommt, während der ursprüngliche Kontrollfluß zur Benutzeroberfläche zurückkehrt und somit die Bearbeitung von `BTA 1` wieder verläßt. Wird nun eine zweite BTA angestoßen (`StartBTA2`), so wird diese zunächst wiederum im Kontrollfluß der Oberfläche ausgeführt. Würde erst jetzt die Hinzunahme des Baustein B ausführenden Kontrollflusses in die Bearbeitung von BTA 1 an das Managementsystem übermittelt, würde dieser fälschlicherweise zu BTA 2 zugeordnet, da diese zu diesem Zeitpunkt bereits in Kontrollfluß der Oberfläche ausgeführt wird.

Daraus ergibt sich die Forderung, daß die Hinzunahme eines weiteren Kontrollflusses vom Managementsystem einer Instanz einer BTA zugeordnet werden muß, bevor eine darauf folgende Mitteilung des bereits existierenden Kontrollflusses verarbeitet werden darf. Andernfalls ist die eindeutige Zuordnung mit Hilfe von Kontrollflüssen nicht mehr gewährleistet. In den meisten Fällen läßt sich diese Einhaltung der Reihenfolge problemlos garantieren: Die Aufrufe des Managementsystems werden im bereits existierenden Kontrollfluß durchgeführt; durch Verwendung synchroner Aufrufe kann zugesichert werden, daß eine Rückkehr zur Anwendung erst stattfindet, nachdem eine Zuordnung zu einer Instanz einer BTA erfolgte. In den Fällen, in denen die Mitteilung über die Hinzunahme eines weiteren Kontrollflusses allerdings erst in dem neu hinzugenommenen Kontrollfluß erfolgen kann (z.B. Aufruf aktiver Bausteine oder verteilte Erbringung einer BTA, siehe unten), ist es nicht mehr möglich, die Einhaltung dieser Reihenfolge zu garantieren. In diesem Fall muß ein expliziter Identifikator generiert werden, der eine spätere Zuordnung gestattet (vgl. Abschnitt 5.4).

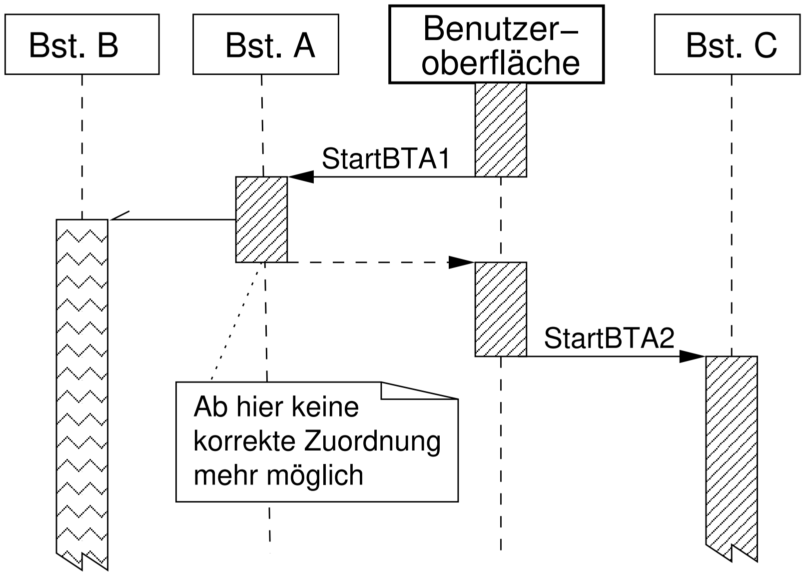

Abbildung 5.17: Korrelation anhand von Kontrollflüssen: Mögliche Probleme

Varianten der Erbringung von BTAs

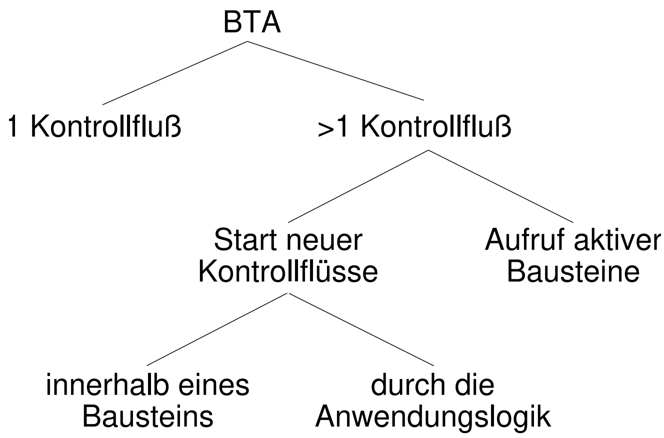

Abbildung 5.18: Varianten der Erbringung von BTAs

Wie bereits gesehen, existieren unterschiedliche Varianten, wie BTAs erbracht werden können und wie zusätzliche Kontrollflüsse in die Bearbeitung einer BTA aufgenommen werden können. Für alle diese Varianten ist es erforderlich, detailliert zu untersuchen, ob und wie die benötigte Korrelationsinformation ermittelt und an das Managementsystem übertragen werden kann. Im folgenden wird daher zunächst eine Klassifikation der verschiedenen Varianten der Erbringung von BTAs angegeben (siehe Abbildung 5.18):

Häufig, insbesondere im Falle von BTAs mit typischerweise kurzer Transaktionsdauer, ist der Fall anzutreffen, daß die komplette BTA innerhalb eines einzigen Kontrollflusses erbracht wird. Dies hat aber z.B. zur Folge, daß während der Bearbeitung der BTA die Oberfläche der Anwendung blockiert ist und somit kein Aufruf weiterer BTAs möglich ist. Aus diesem Grund und zur Erhöhung der Effizienz der Bearbeitung ist häufig der Fall anzutreffen, daß weitere Kontrollflüsse in die Bearbeitung der BTA mit aufgenommen werden. Dies kann einerseits durch den Start neuer Kontrollflüsse geschehen, andererseits aber auch durch den Aufruf aktiver Bausteine, d.h. von Bausteinen, die über einen eigenen Kontrollfluß verfügen. Im Falle des Starts neuer Kontrollflüsse läßt sich weiterhin unterscheiden, ob dieser Start innerhalb eines Bausteins erfolgt oder aber in der Anwendungslogik bei der Verknüpfung zweier Bausteine stattfindet.

Im folgenden werden die oben angegebenen Fälle eingehend untersucht und es wird angegeben, wie in den unterschiedlichen Fällen die Zuordnung der Subtransaktionen erfolgen kann. Ein Sonderfall mehrerer Kontrollflüsse, die über mehrere Prozesse (oder Systeme) verteilte Erbringung einer BTA, wird im Anschluß daran betrachet.

Korrelation mehrerer Kontrollflüsse
Sind mehrere Kontrollflüsse an der Erbringung einer BTA beteiligt, so müssen dem Managementsystem die Identifikatoren aller beteiligten Kontrollflüsse bekannt sein. Da zu einem bestimmten Zeitpunkt jeder Kontrollfluß nur an der Bearbeitung einer einzigen Instanz einer BTA beteiligt sein kann, ist dies bereits ausreichend, um alle beteiligten Subtransaktionen eindeutig ihrer jeweiligen BTA zuordnen zu können.

Entlang der in Abbildung 5.18 angegebenen Varianten der Erbringung von BTAs wird im folgenden untersucht, wie die Zuordnung der Subtransaktionen im Falle mehrerer beteiligter Kontrollflüsse erfolgen kann.

- Start neuer Kontrollflüsse
  Der Start neuer Kontrollflüsse erfolgt durch einen Systemmechanismus. Dieser wird gewöhnlich von einer speziellen Bibliothek des Systems aufgerufen, die den eigentlichen Systemaufruf vor der Anwendung verschattet. Der Aufruf der Bibliothek erfolgt innerhalb des aktuellen Kontrollflusses. Somit ist an dieser Stelle sowohl der Identifikator des alten als auch der Identifikator des neuen Kontrollflusses bekannt (siehe Abbildung 5.19). Übergibt man diese Informationen an das Managementsystem, so wird es in die Lage versetzt, den neuen Kontrollfluß ebenfalls der entsprechenden BTA zuzuordnen. Es ist also eine Instrumentierung der Bibliothek für den Start neuer

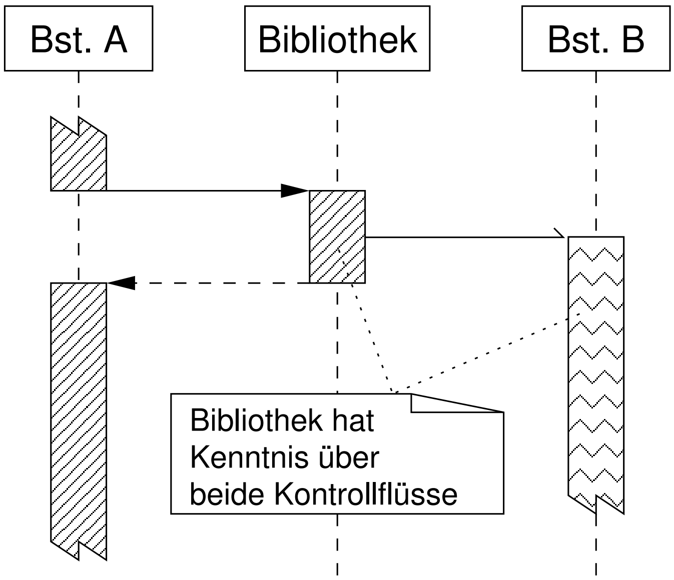

Abbildung 5.19: Start neuer Kontrollflüsse mittels Bibliothek

Kontrollflüsse erforderlich, durch die die gewünschten Informationen an die Managementsysteme übermittelt werden.

Findet der Start des neuen Kontrollflusses nicht innerhalb eines Bausteins, sondern in der Anwendungslogik statt, so ist zu untersuchen, inwiefern die oben beschriebene Technik ausreicht, um die Korrelation der Kontrollflüsse zu gewährleisten: Die Anwendungslogik wird zunächst sicher im Kontrollfluß des aufrufenden Bausteins ausgeführt, unabhängig davon, ob es sich z.B. tatsächlich um einen asynchronen Aufruf eines Bausteines oder einen *Event*-basierten Aufrufmechanismus handelt. Der Start des neuen Kontrollflusses findet somit ebenfalls sicher im Kontrollfluß des aufrufenden Bausteins statt; die oben beschriebene Instrumentierung der Bibliothek für die Erstellung neuer Kontrollflüsse ist somit auch in diesem Falle geeignet, um die Zuordnung herzustellen.

Ebenso wie der Start eines Kontrollflusses kann auch dessen Ende durch Erweiterung des entsprechenden Mechanismus ermittelt und an das Managementsystem übermittelt werden. Start und Stop von Kontrollflüssen sind gleichzeitig als Start und Stop einer Subtransaktion zu betrachten.

- Aufruf aktiver Bausteine

  Neben dem Start neuer Kontrollflüsse ist der Aufruf aktiver Bausteine die zweite Variante, wie weitere Kontrollflüsse an der Erbringung einer BTA beteiligt werden können. Ein aktiver Baustein ist hierbei ein Baustein, der über einen eigenen Kontrollfluß verfügt und der z.B. in regelmäßigen Abständen überprüft, ob ein Auftrag zur Bearbeitung vorliegt. Das Erteilen eines Auftrages erfolgt durch den Aufruf einer entsprechenden Methode des aktiven Bausteins, die beispielsweise die zu bearbeitenden Daten in eine Warteschlange einreiht. Andere Formen der Kommunikation, z.B. über gemeinsamen Speicher sind aufgrund der getroffenen Voraussetzungen über die Kommunikation zwischen Bausteinen nicht möglich.

  Abbildung 5.20 veranschaulicht die Abläufe beim Aufruf eines aktiven Bausteins. Aus darstellungstechnischen Gründen wurde eine explizite Warteschlange für die Aufträge eingeführt; diese kann aber ebenso Teil des aktiven Bausteins sein. Um einen Auftrag an einen aktiven Baustein zu erteilen, ruft ein *Client*-Baustein eine entsprechende Methode der Warteschlange auf, die somit im Kontrollfluß des *Clients* ausgeführt wird. Zu beliebigen, nicht vorherbestimmbaren Zeitpunkten überprüft der aktive Baustein, ob Aufträge in der Warteschlange vorliegen. Ist dies der Fall, so beginnt er mit der Bearbeitung des jeweiligen Auftrags.

  Die Problematik, die sich hierbei ergibt, ist die vollständige Entkoppelung der beteiligten Kontrollflüsse. Zum Zeitpunkt der Auftragsbearbeitung durch den aktiven Baustein ist es nicht mehr ohne Instrumentierung des Bausteins möglich zu entscheiden, welcher BTA die Subtransaktion zuzuordnen ist, da sich in der Warteschlange Daten unterschiedlicher BTAs zur Bearbeitung befinden können (Eine Lösung für diese Problematik wird in Abschnitt 5.4.1 angegeben).

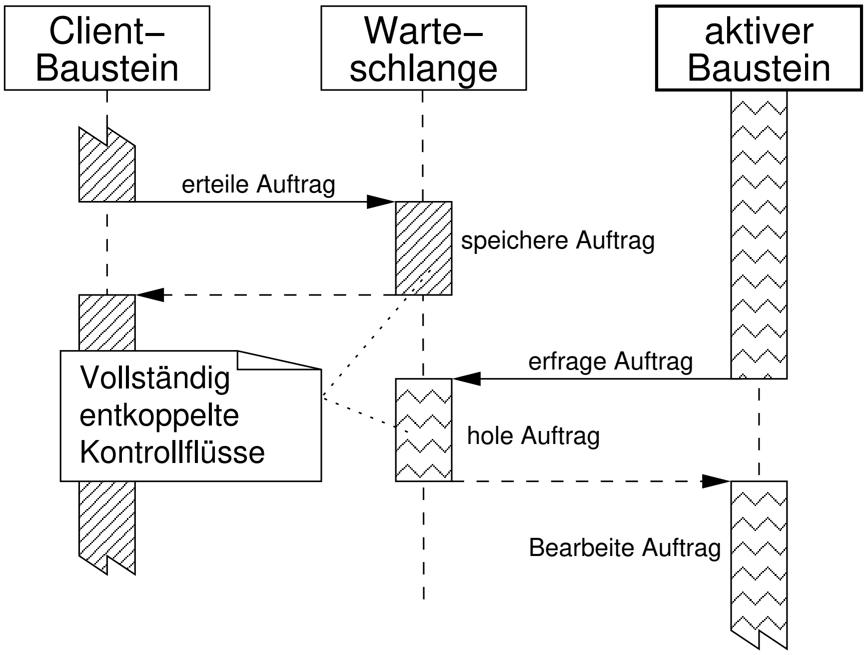

Abbildung 5.20: Vollständige Entkoppelung der Kontrollflüsse im Falle aktiver Bausteine

Verteilte Erbringung einer BTA
Ein häufig auftretender Fall ist die über mehrere Prozesse oder sogar mehrere Systeme verteilte Erbringung einer BTA. Auch hier ist es erforderlich, die unterschiedlichen Subtransaktionen der entsprechenden BTA zuordnen zu können. Das Managementsystem braucht also wiederum Information darüber, welche Kontrollflüsse des entfernten Prozesses mit welcher Instanz einer BTA zu korrelieren sind.

Die erforderliche Information kann mit Hilfe einer Instrumentierung der darunterliegenden Kommunikationsmechanismen erbracht werden: Die Kommunikation zwischen den Prozessen erfolgt im Normalfall über einen Mechanismus, der die Details der Kommunikation vor den Bausteinen verschattet. Erweitert man diesen Mechanismus dahingehend, daß er einen Identifikator der aktuellen BTA an das entfernte System überträgt, so kann die spätere Zuordnung erfolgen.

Da aber nicht davon auszugehen ist, daß die beteiligten Systeme über untereinander synchronisierte Uhren verfügen, läßt sich die Reihenfolge der Subtransaktionen in diesem Fall nicht mehr ohne weiteres anhand der gemessenen Zeitpunkte rekonstruieren. Stattdessen ist es erforderlich, daß auf dem aufrufenden System der Zeitpunkt des Aufrufes festgehalten und mit einem eindeutigen Identifikator versehen wird, der an das entfernte System übertragen werden kann. Anhand dieses Identifikators ist dann eine spätere problemlose Zuordnung der Subtransaktionen sowie deren Reihenfolge möglich. Zusätzlich muß im aufrufenden System gespeichert werden, ob es sich um einen synchronen oder asynchronen Aufruf handelt.

5.3.4 Anforderungen an die Implementierung der Bausteinarchitektur

Die vorangegangene Untersuchung hat gezeigt, daß gewisse Anforderungen an die darunterliegende Bausteinarchitektur zu stellen sind, um die vorgestellte Lösung zu ermöglichen. Diese sollen hier noch einmal kurz zusammengefaßt dargestellt werden:

- Instrumentierbare Anwendungslogik
  Die Kommunikation zwischen zwei Bausteinen darf nicht direkt erfolgen, sondern muß mittelbar über die Anwendungslogik erfolgen. Es muß möglich sein, Aufrufe einer Managementschnittstelle in die Anwendungslogik zu integrieren. Die Entwicklungsumgebung muß es gestatten, bei Einfügen einer Verknüpfung Parameter anzugeben, die zur Laufzeit an das Managementsystem übergeben werden.

- Eindeutig identifizierbare Kontrollflüsse
  Unterschiedliche Kontrollflüsse müssen anhand eines Identifikators innerhalb desselben Systems eindeutig unterscheidbar sein. Innerhalb eines Kontrollflusses muß es möglich sein, dessen Identifikator zu ermitteln und weiterzugeben.

- Asynchrone Kommunikation mittels synchroner Mechanismen
  Asynchrone Kommunikation muß auf synchrone Kommunikation zurückführbar sein. D.h., um asynchrone Kommunikation zu erreichen müssen Kommunikationsmechanismen verwendet werden, die synchron aufgerufen werden und erst dann durch die Aktivierung weiterer Kontrollflüsse für Asynchronität sorgen. Dies ist bei den hier vorausgesetzten Mechanismen zur Kommunikation zwischen Bausteinen (z.B. RPC, Ereigniskanäle) sicher erfüllt.

- Instrumentierbare Bibliotheken des Systems
  Es muß die Möglichkeit bestehen, gewisse Bibliotheken des Systems, wie z.B. die Bibliotheken zum Aktivieren und Stoppen von Kontrollflüssen so zu instrumentieren, daß diese die benötigte Korrelationsinformation an das Managementsystem melden.

- Erkennbare Fehlerzustände
  Die fehlerhafte Bearbeitung eines Aufrufs muß erkennbar sein. Im Falle synchron aufgerufener Bausteine bedeutet dies, daß Fehler nicht als normale Rückgabewerte, sondern als *Exception* übermittelt werden. Asynchron aufzurufende Bausteine müssen über dedizierte Fehlerausgänge verfügen.

5.3.5 Bewertung

Der zweite vorgeschlagene Ansatz verzichtet völlig auf die Instrumentierung von Bausteinen und ermittelt die benötigte Information durch Instrumentierung der Anwendungslogik. Da diese Instrumentierung nahezu vollständig automatisiert erfolgen kann (vgl. Abschnitt 5.4.3) und eine Zuordnung von Subtransaktionen zu ihren zugehörigen BTAs automatisch anhand von Kontrollflüssen erfolgen kann, kann der Gesamtaufwand im Vergleich zum ersten Ansatz nochmals erheblich reduziert werden. Auch hinsichtlich der Integration von *Legacy*-Bausteinen, der Verläßlichkeit und Präzision der Information oder der Flexibilität der Lösung stellt sich der zweite Ansatz als günstiger heraus. Es zeigt sich allerdings, daß nicht in allen Fällen vollständig auf die Instrumentierung von Bausteinen verzichtet werden kann. So bedingen aktive Bausteine z.B. eine Instrumentierung, um in die Überwachung integriert werden zu können; Oberflächenbausteine können zwar überwacht werden, die Präzision der Messung könnte durch deren Instrumentierung allerdings weiterhin verbessert werden. Daher basiert die im nächsten Abschnitt vorgestellte Architektur auf einer Kombination der beiden vorgestellten Ansätze, um die Vorteile beider Lösungen zu vereinen. Im folgenden wird eine detailliertere Bewertung des zweiten vorgestellten Ansatzes vorgenommen:

- Aufwand
  Da keinerlei Instrumentierung der Bausteine vorausgesetzt wird, ist auch keinerlei zusätzlicher Aufwand für den Bausteinentwickler zu erwarten. Einzig der Anwendungsentwickler muß die Identifikation derjenigen BIs vornehmen, die als der Beginn einer BTA betrachtet werden sollen und muß diese eindeutig benennen. Ebenso muß er die BIs identifizieren, mit denen das Ergebnis der BTA dem Benutzer übermittelt wird. Im Fall von asynchroner Kommunikation zwischen den Bausteinen muß er darüber hinaus angeben, welche der Ausgänge eines Bausteins eine fehlerhafte Bearbeitung des erfolgten Aufrufs repräsentieren.

  Darüber hinaus wird auch vom Anwendungsentwickler kein weiterer Aufwand gefordert; der zusätzliche Aufwand für Anwendungs- bzw. Bausteinentwickler, der aus der Verwendung dieser Lösung resultiert, kann also als äußerst gering bezeichnet werden.

- Verfügbare Information
  Der vorgestellte Ansatz gestattet die nahezu vollständige Ermittlung der geforderten Information. Einzig bezüglich der Erkennung von Fehlerzuständen kann durch Instrumentierung der Bausteine zusätzliche, wertvolle Information gewonnen werden. Sowohl die Erkennung interner Fehlerzustände einzelner Bausteine als auch die Erkennung des Scheiterns der gesamten BTA würde sich durch Instrumentierung der Bausteine vereinfachen.

- Zuordnung der Teilinformationen
  Durch die vollständig dynamische Beschreibung der Abhängigkeiten der einzelnen

Bausteine einer Anwendung wird es möglich, weitaus realitätsnähere Aussagen z.B. über die Auswirkungen eines Bausteinausfalls zu treffen. Insbesondere gestattet die dynamische Zuordnung von Subtransaktionen korrekte Ergebnisse auch bei paralleler Ausführung mehrerer Instanzen von BTAs.

Im Rahmen der Untersuchungen hat sich allerdings gezeigt, daß eine korrekte Beschreibung der Abhängigkeiten bei Verwendung aktiver Bausteine nicht möglich ist. Hierfür ist zwingend eine Instrumentierung des aktiven Bausteins erforderlich.

- Integration von Legacy-Bausteinen
  Die vorgestellte Lösung eignet sich hervorragend für die Integration von *Legacy*-Bausteinen, da keine besonderen Anforderungen an die Bausteine gestellt werden. Jeder Baustein (mit Ausnahme der eben beschriebenen aktiven Bausteine) kann ohne jegliche Veränderung in die Überwachung aufgenommen werden.

- Verläßlichkeit der Information
  Dadurch, daß nicht der Bausteinentwickler für die erbrachte Information verantwortlich ist, ist davon auszugehen, daß verläßlichere Informationen geliefert werden können. Dies bezieht sich sowohl auf die verringerte Fehleranfälligkeit durch erhebliche Automatisierung als auch auf den Ausschluß bewußter Täuschung durch den Bausteinentwickler.

- Flexibilität
  Da bei der vorgestellten Lösung keinerlei Vorverarbeitung der Daten im Baustein stattfindet, kann größtmögliche Flexibilität der Managementsysteme erreicht werden. Selbstverständlich muß dennoch auf eine geeignete Vorverarbeitung der Information auf dem System, auf dem die Information anfällt, geachtet werden, um die Performanz des Managemensystems nicht zu beeinträchtigen.

- Präzision der Information
  Da der in diesem Abschnitt vorgestellte Ansatz die Information einer BTA explizit mißt und nicht aus Informationen der Subtransaktionen berechnet, lassen sich präzisere Werte erzielen. Auch die in der Anwendungslogik verbrauchte Zeit kann bei dieser Vorgehensweise problemlos mit überwacht werden.

  Problematisch stellt sich allerdings dar, daß sowohl der Startzeitpunkt einer BTA als auch der Zeitpunkt der Präsentation eines Ergebnisses nur näherungsweise bestimmt werden können. Dies liegt daran, daß Meßpunkte bei diesem Ansatz nur zwischen Bausteinen platziert werden können, keinesfalls aber innerhalb eines Bausteins. Eine Instrumentierung der Oberflächenbausteine würde hier eine wesentlich präzisere Messung ermöglichen.

5.4 Architektur für die Überwachung bausteinbasierter Anwendungen

Der folgende Abschnitt stellt eine Architektur für die Überwachung bausteinbasierter Anwendungen dar. Diese basiert zu großen Teilen auf dem eben vorgestellten Ansatz der *Automation der Managementinstrumentierung bausteinbasierter Anwendungen* (Abschnitt 5.3). Dies gestattet das weitgehend automatische Einfügen von Meßpunkten in die erstellte Anwendungslogik sowie die automatische Zuordnung der einzelnen Meßwerte zu ihrer übergeordneten BTA. Da dieser Ansatz alleine aber nicht in der Lage ist, die gestellten Anforderungen vollständig zu erfüllen, werden darüber hinaus auch Aspekte des zuvor beschriebenen Ansatzes der *Komposition von Managementschnittstellen instrumentierter Bausteine* (Abschnitt 5.2) einbezogen. Dies bezieht sich insbesondere auf die Instrumentierung von Bausteinen, die sich an einigen Stellen als unvermeidbar erwiesen hat. Durch die Möglichkeit der optionalen Instrumentierung aller Bausteine kann insbesondere im Fehlerfall wesentlich detailliertere Information ermittelt werden.

Abschnitt 5.4.1 stellt dar, wie durch die Instrumentierung von Bausteinen die Nachteile des im vorangegangenen Abschnitts dargestellten Ansatzes behoben werden können. Die eigentliche Architektur läßt sich anhand des Lebenszyklus einer Anwendung in zwei Teile untergliedern: Einen Teil, der für die Erstellung einer managementinstrumentierten Anwendung verantwortlich ist, und einen Teil, der zur Laufzeit der Anwendung die Ermittlung der Managementinformation durchführt. In Abschnitt 5.4.2 wird zunächst eine detaillierte Beschreibung der *Architektur für die Ermittlung der Managementinformation* zur Laufzeit inklusive ihrer Schnittstellen angegeben. Abschnitt 5.4.3 beschreibt daraufhin die *Architektur für die Automation der Managementinstrumentierung* von Anwendungen und legt Methodiken für Anwendungs- sowie Bausteinentwickler fest. Ein abschließender Vergleich mit dem derzeit am Markt am stärksten verbreiteten Ansatz für die Managementinstrumentierung von Anwendungen, der ARM API, stellt in Abschnitt 5.5.1 die Vorteile der konzipierten Lösung nochmals heraus.

5.4.1 Instrumentierung von Bausteinen

Bei der Untersuchung der *Automation der Managementinstrumentierung bausteinbasierter Anwendungen* hat sich herausgestellt, daß eine Integration aktiver Bausteine ohne Instrumentierung dieser Bausteine nicht möglich ist. Ebenso hat sich gezeigt, daß die Präzision der gemessenen Information durch eine Instrumentierung der Oberflächenbausteine wesentlich verbessert werden könnte. Das Scheitern einer Transaktion eines Bausteines kann zwar anhand des Rückgabewertes bzw. Ausgangs erkannt werden, detaillierte Informationen über Art und Ursache des Fehlers können allerdings wiederum nur durch eine Instrumentierung des Bausteins ermittelt werden.

Im weiteren Verlauf wird dargestellt, wie eine Instrumentierung der entsprechenden Bausteintypen zu erfolgen hat, um die Schwachstellen des im vorangegangenen Abschnitt beschriebenen Ansatzes zu beheben. Eine Beschreibung der Methodik, nach der ein Bausteinentwickler bei der Instrumentierung vorgehen sollte, findet sich in Abschnitt 5.4.3.2.

5.4.1.1 Integration aktiver Bausteine

Wie bereits dargestellt (vgl. Abbildung 5.20), ist das Problem bei der Integration aktiver Bausteine in die Überwachung die vollständige Entkoppelung der beteiligten Kontrollflüsse. Anders als beim Start neuer Kontrollflüsse, z.B. durch einen Systemmechanismus, liegt hier zu keinem Zeitpunkt die Information über aufrufenden und aufgerufenen Kontrollfluß gemeinsam vor. Das Einfügen eines Auftrags in die Warteschlange des aktiven Bausteins erfolgt innerhalb des Kontrollflusses des aufrufenden (*Client-*) Bausteins. Zu einem späteren Zeitpunkt wird dieser Auftrag im Kontrollfluß des aktiven Bausteins bearbeitet. Im aufrufenden Kontrollfluß ist der Identifikator des ausführenden Kontrollflusses ebenso wenig bekannt wie umgekehrt der Identifikator des aufrufenden Kontrollfluß im aufgerufenen.

Durch geeignete Instrumentierung des aktiven Bausteins ist es aber auch in diesem Fall möglich, die Zuordnung zu gewährleisten. Ähnlich wie im Falle der verteilten Erbringung von BTAs muß der aktive Baustein (bzw. im vorliegenden Fall die Warteschlange) bei Entgegennahme eines Auftrages einen eindeutigen Identifikator beim Managementsystem erfragen, der gemeinsam mit der Information über den Auftrag gespeichert wird. Zu Beginn der eigentlichen Bearbeitung des Auftrags kann anhand dieses Identifikators die Zuordnung zur anstoßenden Transaktion hergestellt werden. Die zusätzlich einzubringende Funktionalität ist in Abbildung 5.21 durch kräftigeren Druck gekennzeichnet.

5.4.1.2 Instrumentierung von Oberflächenbausteinen

Beim Ansatz der *Automation der Instrumentierung bausteinbasierter Anwendungen* mußte davon ausgegangen werden, daß innerhalb der Oberflächenbausteine neben der Realisierung der BI keine nennenswerten Berechnungen ausgeführt werden. Unter dieser Voraussetzung war es möglich, den Beginn sowie den Zeitpunkt der Ergebnispräsentation einer BTA aufgrund des Zeitpunktes der Verknüpfung dieser Bausteine zu ermitteln. Ist diese Voraussetzung allerdings nicht erfüllt, liefert ein dementsprechendes Vorgehen nur unzureichende Präzision. Günstiger wäre es in diesem Fall, die Oberflächenbausteine so zu instrumentieren, daß der genaue Zeitpunkt der BI gemessen werden kann. Die möglichen Stellen innerhalb der Oberflächenbausteine, an denen BIs stattfinden können, sind bereits vor der Erstellung der eigentlichen Anwendung bei der Entwicklung der Bausteine bekannt.

Zum Zeitpunkt der Anwendungserstellung sind dem Anwendungsentwickler diejenigen BIs bekannt, die als Beginn bzw. Ergebnispräsentation zu überwachender BTAs betrachtet

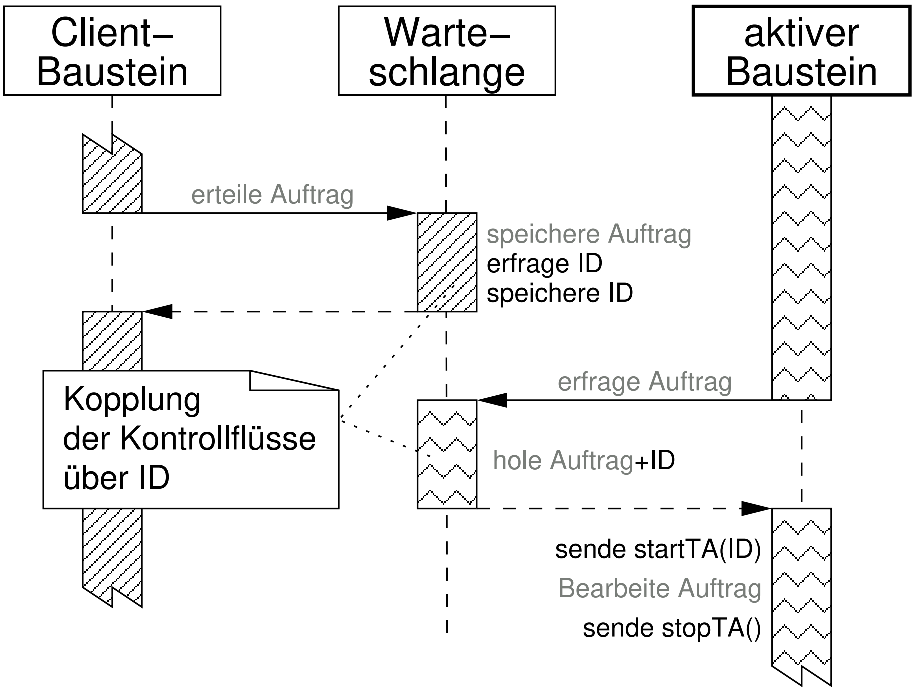

Abbildung 5.21: Instrumentierung aktiver Bausteine

werden sollen. Aufgabe des Anwendungsentwicklers ist es nun, diese BIs zu identifizieren und die zugehörige BTA (innerhalb der Anwendung) eindeutig zu benennen. Da der Anwendungsentwickler keinen Zugriff auf den Code der eingesetzten Bausteine hat, muß dies im Rahmen des *Customizing* der Oberflächenbausteine erfolgen.

Somit muß vom Entwickler von Eingabebausteinen der Baustein-Code dahingehend erweitert werden, daß an allen Stellen, die BIs erlauben, die Möglichkeit besteht, durch *Customizing* des Bausteins die jeweilige Stelle als den Beginn einer BTA zu benennen und ihr einen eindeutigen Namen zu geben (siehe Abschnitt 5.4.3.2). Dies bedeutet insbesondere, daß von dem entsprechenden Baustein bei Eintreten der BI über die Meßschnittstelle der Beginn der BTA sowie ihr Name an das Managementsystem übermittelt werden muß.

Ebenso müssen die Präsentationsbausteine so instrumentiert werden, daß jede BI, durch die einem Benutzer Ergebnisse präsentiert werden können, im Rahmen des *Customizing* als tatsächliches Ende einer BTA festgelegt werden kann. Darüber hinaus muß eine Möglichkeit vorhanden sein, den Erfolg oder Mißerfolg der BTA angeben zu können.

5.4.1.3 Identifikation interner Fehlerzustände eines Bausteins

Über das Abfangen von *Exceptions* bzw. die Identifikation von Fehlerausgängen ist es möglich, die fehlerhafte Bearbeitung von Aufrufen einzelner Bausteine zu erkennen. Um aber eine einfache und schnelle Behebung von Fehlern zu ermöglichen, sollte ein Baustein neben der Information über das Auftreten eines Fehlers auch Information über die (mögliche) Fehlerursache liefern.

Diese Information kann nur durch entsprechende Instrumentierung des Bausteins ermittelt werden. Es ist nur sehr eingeschränkt möglich, generische Vorgaben über die hierbei zu übermittelnde Information zu machen. Das CAMI des *TMForum* (vgl. Abschnitt 4.2.1.4) stellt einen ersten Ansatz dar, der sich aber auf die Angabe einiger weniger Problemklassen beschränkt. Diese dienen der näheren Beschreibung der Ursache des Problems. Beispiele hierfür sind ein internes Problem des Bausteins, ein Problem mit einem *Client* oder *Server* oder ein Netzproblem.

Wesentlich ist letztlich, einem Baustein eine Möglichkeit anzubieten, die Information an das Managementsystem zu übermitteln. Durch Ausnutzung der Kenntnis über den Kontrollfluß, in dem das Problem auftauchte, wird es gleichzeitig möglich, den aufgetretenen Fehlerzustand einer Instanz einer BTA zuzuordnen.

5.4.2 Architektur für die Ermittlung der Managementinformation

Der folgende Abschnitt stellt die Architektur für die Ermittlung der erforderlichen Managementinformation zur Laufzeit der Anwendung dar. Nach einem kurzen Überblick über die einzelnen Komponenten werden die Kommunikationsschnittstellen der Architektur detailliert beschrieben.

5.4.2.1 Architekturübersicht

Die Architektur für die Ermittlung der Managementinformation besteht aus einer instrumentierten Anwendung, einem Meßobjekt pro Prozeß, instrumentierten Bibliotheken sowie Managementagenten und einem Managementsystem. Abbildung 5.22 zeigt diese Architektur. Sowohl die instrumentierte Anwendung als auch die von dieser verwendeten instrumentierten Bibliotheken rufen Methoden des in den lokalen Prozeß integrierten Meßobjekts auf, um so Informationen über BTAs und Subtransaktionen zu übergeben. Das Meßobjekt nimmt die Information entgegen und übergibt sie an einen Managementagenten auf dem lokalen System, der diese beliebig verarbeiten oder an ein übergeordnetes Managementsystem weiterleiten kann. Anwendungen können sich hierbei über mehrere Prozesse erstrecken, die sich sowohl innerhalb eines Systems befinden können als auch über mehrere Systeme verteilt sein können. In den folgenden Abschnitten werden die einzelnen Komponenten der Architektur detailliert vorgestellt.

Instrumentierte Anwendung

Die instrumentierte Anwendung liefert Informationen über den Beginn und das Ende von

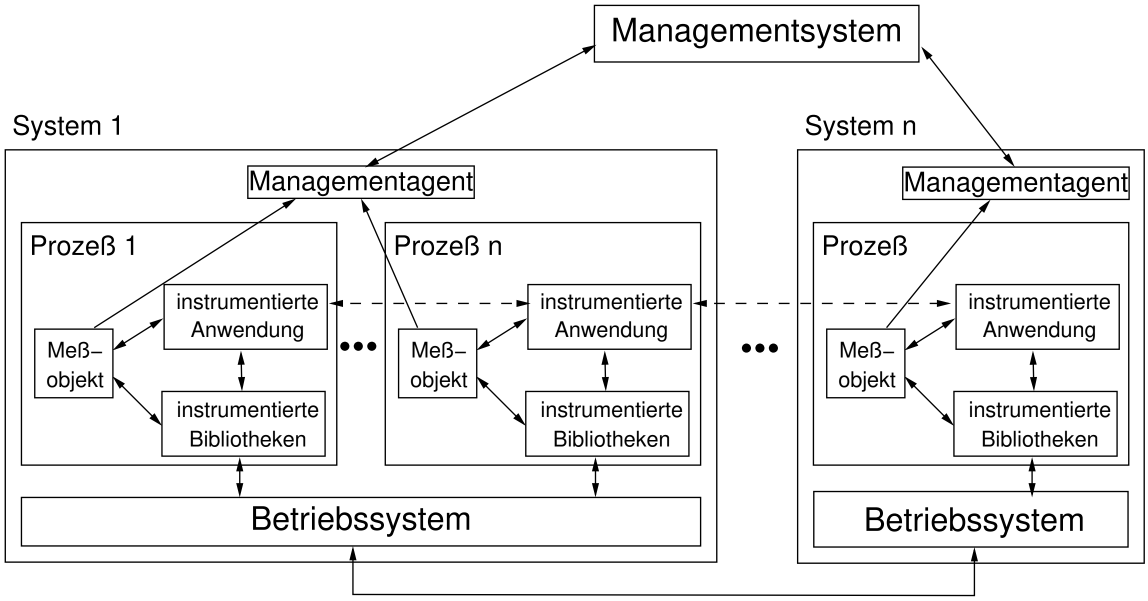

Abbildung 5.22: Architektur für die Ermittlung der Managementinformation

BTAs und ihrer Subtransaktionen an das Meßobjekt. Dies erfolgt über die Meßschnittstelle (siehe Abschnitt 5.4.2.2). Die Erstellung einer instrumentierten Anwendung kann mit Hilfe der in Abschnitt 5.4.3 dargestellten Architektur mit sehr geringem Aufwand für Anwendungs- und Bausteinentwickler erfolgen.

Instrumentierte Bibliotheken

Um BTAs, die mehrere Kontrollflüsse umfassen, für Anwendungs- und Bausteinentwickler transparent überwachen zu können, ist die Instrumentierung einiger weniger Systembibliotheken erforderlich. Insbesondere müssen der Mechanismus für den Start und die Beendigung neuer Kontrollflüsse und die Mechanismen für die Kommunikation zwischen Prozessen instrumentiert werden.

- Start und Stop von Kontrollflüssen

  Bei jedem Start eines Kontrollflusses muß der Identifikator des neu gestarteten Kontrollflusses an das Meßobjekt übermittelt werden. Dies geschieht mit Hilfe des Aufrufs `addControlFlow`. Wird ein Kontrollfluß beendet (oder verläßt die Bearbeitung der BTA anderweitig), muß dies dem Meßobjekt mittels `removeControlFlow` angezeigt werden.

- Interprozeßkommunikation

  Ebenso müssen die Bibliotheken für die Kommunikation zwischen Prozessen dahingehend erweitert werden, daß eine Korrelation der Subtransaktionen möglich ist. Dies bedeutet, daß der lokale Teil des Kommunikationsmechanismus mit Hilfe des Aufrufs `initiatedTA` dem Meßobjekt mitteilt, daß eine Transaktion angestoßen wurde. Als Rückgabewert wird ein vom Meßobjekt generierter eindeutiger Identifikator geliefert, der an den entfernten Teil des Kommunikationsmechanismus übergeben wird. Dort wird durch einen Aufruf von `startTA` (mit diesem Identifikator als Parameter) dem entfernten Meßobjekt der Beginn der neuen Subtransaktion übermittelt.

Mit Ausnahme dieser Bereiche sind für die vorgestellte Architektur keinerlei Anpassungen der darunterliegenden Systeme erforderlich.

Meßobjekt

Das Meßobjekt wird wie bereits gesehen sowohl von der instrumentierten Anwendung als auch von den instrumentierten Bibliotheken des Systems aufgerufen. Es stellt hierzu die sogenannte Meßschnittstelle zur Verfügung (siehe Abschnitt 5.4.2.2). Aus Performanzgründen wird das Meßobjekt innerhalb des Prozesses der zu überwachenden Anwendung ausgeführt, es existiert also ein Meßobjekt pro Prozeß. Die gewonnene Information wird über die Managementschnittstelle (siehe Abschnitt 5.4.2.2) des Meßobjektes an den Managementagenten des jeweiligen Systems zur weiteren Verarbeitung übergeben.

Das Meßobjekt nimmt die einzelnen Aufrufe entgegen und ordnet Subtransaktionen den entsprechenden BTAs zu. Die Zuordnung erfolgt anhand des Kontrollflusses, in dem der aufrufende Baustein ausgeführt wird. Da Aufrufe der Meßschnittstelle im Kontrollfluß des Aufrufers ausgeführt werden, ist die Bestimmung des Identifikators des aktuellen Kontrollflusses innerhalb des Meßobjekts möglich und muß nicht von der Anwendung übergeben werden. Weiterhin muß die aktuelle Systemzeit zum Zeitpunkt des Aufrufs ermittelt werden.

Wird ein neuer Kontrollfluß in die Bearbeitung einer BTA aufgenommen oder verläßt ein Kontrollfluß die Bearbeitung einer BTA, so muß das Meßobjekt dies in einer entsprechenden Datenstruktur eintragen, um später die korrekte Korrelation von Transaktionen gewährleisten zu können. Hierzu muß für jede Instanz einer BTA ein eindeutiger Identifikator generiert werden, anhand dessen die Zuordnung erfolgen kann. Die Zuordnung eines Aufrufs zur Instanz einer BTA muß innerhalb des synchronen Aufrufs des Meßobjekts erfolgen, um eine korrekte Korrelation zu gewährleisten. Die weitere Verarbeitung und Weiterleitung an den Managementagenten kann hingegen asynchron erfolgen. Die genaue Spezifikation der vom Meßobjekt angebotenen Schnittstellen sowie deren Semantik wird in Abschnitt 5.4.2.2 ausführlich beschrieben.

Managementagent

Auf jedem System läuft ein Managementagent, der die Managementinformation der einzelnen Meßobjekte des Systems sammelt, evtl. vorverarbeitet und an Managementsysteme zur weiteren Verarbeitung verteilt.

Managementsysteme
In den angeschlossenen Managementsystemen kann eine beliebige Weiterverarbeitung der gewonnenen Information erfolgen. Insbesondere muß dort im Falle verteilter Erbringung von BTAs eine geeignete Zusammenführung der Teilinformation der einzelnen Managementagenten stattfinden. Durch die Generierung eindeutiger Identifikatoren sowie die Verwendung des `initiatedTA`-Aufrufs läßt sich diese Zusammenführung aber problemlos erreichen.

In Abschnitt 5.6 wird anhand der Bestimmung der Verfügbarkeit von Anwendungsdiensten ein Beispiel dafür gegeben, welche Funktionalität von einem derartigen Managementsystem erbracht werden könnte. Auf eine weiterführende Diskussion der Managementsysteme wird an dieser Stelle aber verzichtet, da dies den Rahmen der vorliegenden Arbeit erheblich übersteigen würde. In Abschnitt 6.2.4 wird eine einfache prototypische Managementanwendung zur Visualisierung der ermittelten Informationen vorgestellt.

5.4.2.2 Definition von Schnittstellen

Aufgrund der im vorigen Abschnitt eingeführten Architektur ergeben sich die drei in Abbildung 5.23 zu erkennenden Kommunikationsschnittstellen. Zum einen ist die Schnittstelle zwischen instrumentierter Anwendung bzw. instrumentierten Bibilotheken und dem Meßobjekt, die sogenannte Meßschnittstelle, zu spezifizieren. Außerdem ist es erforderlich, die Schnittstelle zwischen dem Meßobjekt und dem zugehörigen Managementagenten festzulegen, die im folgenden als Managementschnittstelle der Anwendung bezeichnet wird. Die Schnittstelle zwischen den Managementagenten und den angeschlossenen Managementsystemen ist vom jeweils angeschlossenen Managementsystem abhängig und kann daher im Rahmen der vorliegenden Arbeit nicht betrachtet werden. Je nach Managementar-

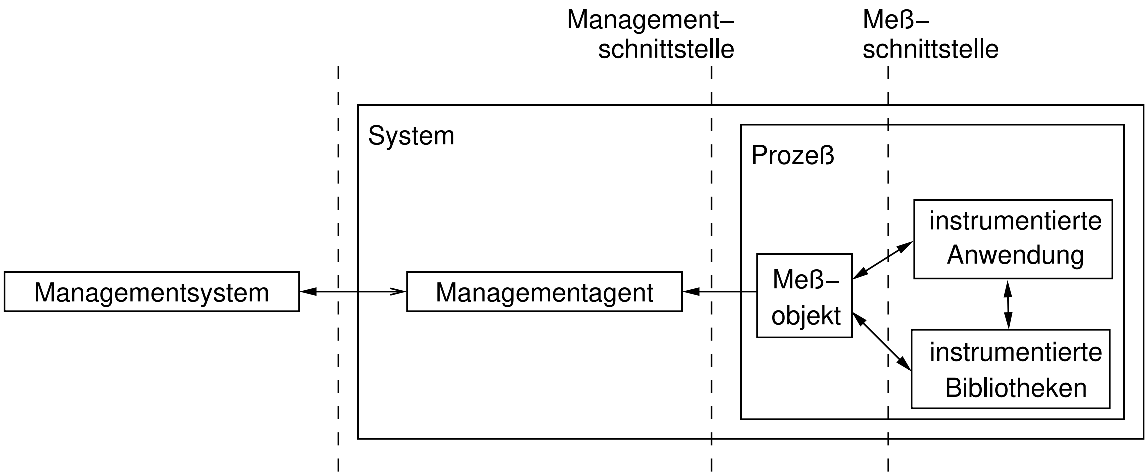

Abbildung 5.23: Kommunikationsschnittstellen der Architektur

chitektur kann für eine Realisierung dieser Schnittstelle beispielsweise SNMP [CFSD 90], CMIP [ISO 9596-1] oder CORBA [CORBA 2.2] eingesetzt werden.

Meßschnittstelle

An der Meßschnittstelle stellt ein Meßobjekt den instrumentierten Anwendungen acht Operationen zur Verfügung, über die z.B. Informationen über den Beginn und das Ende von Transaktionen übermittelt werden können. Es handelt sich um Klassenoperationen, um den einfachen Zugriff aus der gesamten Anwendung heraus zu gestatten. Die Meßschnittstelle ist abhängig von der jeweils verwendeten Programmiersprache. Abbildung 5.24 faßt die angebotenen Operationen programmiersprachenunabhängig zusammen.

Der folgende Abschnitt gibt eine detaillierte Beschreibung der einzelnen Operationen und ihrer Parameter:

- `startBTA(btaName: String, userName: String, bausteinName: String, bausteinID: OID)`
  Aufruf, der den Beginn einer BTA anzeigt. `StartBTA` wird unmittelbar nach einer Benutzerinteraktion aufgerufen, die den Beginn einer BTA darstellt.

  Parameter:

  - `btaName: String`
    Eindeutiger Name des Typs der BTA.
    Der Name der BTA wird durch den Anwendungsentwickler beim *Customizing* des Eingabebausteins vergeben.
  - `userName: String`
    Name des Benutzers, der die BTA angestoßen hat.

<<interface>> **Meßschnittstelle**
startBTA(btaName: String, userName: String, bausteinName: String, bausteinID: OID) *stopBTA(status: String, info: String)* *startTA(bausteinName: String, bausteinID: OID, parentTA: UUID=0)* *stopTA(status: String, info: String)* *addControlFlow(newControlFlow: ThreadID, target: OID)* *removeControlFlow(status: String, reason: String)* *initiatedTA(isAsynchronous: Boolean):UUID* *logInfo(info: String)*

Abbildung 5.24: Spezifikation der Meßschnittstelle

- `bausteinName: String`
  Name des Eingabebausteins, in dem die Benutzerinteraktion stattfand. Der Name des Bausteins wird vom Bausteinentwickler beim Entwurf des Bausteins festgelegt.
- `bausteinID: OID`
  Identifikator der ausführenden Instanz des Bausteins. Dieser kann erst zur Laufzeit dynamisch bestimmt werden.

Bei Entgegennahme eines `startBTA`-Aufrufs generiert das Meßobjekt einen eindeutigen Identifikator für die BTA und bestimmt die aktuelle Zeit sowie den aktuellen Kontrollfluß. Die Zuordnung von BTA zu Kontrollfluß wird festgehalten. Die ermittelten sowie die übergebenen Informationen werden gemeinsam abgespeichert.

- `stopBTA(status: String, info: String)`
  Aufruf, der den Zeitpunkt der Präsentation des Ergebnisses einer BTA anzeigt. `StopBTA` wird unmittelbar vor der Benutzerinteraktion aufgerufen, die das Ergebnis der BTA dem Benutzer präsentiert.

  Parameter:

  - `status: String`
    Gibt an, ob die BTA erfolgreich oder nicht erfolgreich abgeschlossen wurde. Erfolgreiche Bearbeitung wird durch `Success`, nicht erfolgreiche durch `Failure` repräsentiert. Läßt sich der Erfolg der Bearbeitung nicht angegeben, so ist `unknown` zu übergeben.
  - `info: String`
    Zur Übergabe beliebiger weiterer Information z.B. hinsichtlich Fehlerursache etc.

  Das Meßobjekt muß bei Aufruf von `stopBTA` anhand des Kontrollflusses die Zuordnung zu einer BTA herstellen. Der aktuelle Zeitpunkt sowie der übergebene Wert für den Erfolg der BTA wird gespeichert. Im Falle mehrerer Aufrufe von `stopBTA` innerhalb derselben BTA ist der letzte Aufruf als der korrekte Zeitpunkt der Ergebnispräsentation zu betrachten. Dies ist bereits im Meßobjekt möglich, da, wie in Abschnitt 5.3.2.1 beschrieben, davon auszugehen ist, daß alle Aufrufe von `stopBTA` auf demselben System erfolgen.

- `startTA(bausteinName: String, bausteinID: OID, parentTA: UUID=0)`
  Aufruf, um den Beginn einer Subtransaktion anzuzeigen. Wird unmittelbar vor Aufruf einer Operation eines Bausteins aufgerufen.

Parameter:

- `bausteinName: String`
  Name des aufgerufenen Bausteins.
  Der Name des Bausteins wird vom Bausteinentwickler beim Entwurf des Bausteins festgelegt. Zum Zeitpunkt der Anwendungserstellung ist der Name des aufgerufenen Bausteins bereits bekannt und kann automatisch von der Entwicklungsumgebung eingefügt werden.
- `bausteinID: OID`
  Identifikator der ausführenden Instanz des aufgerufenen Bausteins. Dieser kann erst zur Laufzeit dynamisch bestimmt werden.
- `parentTA: UUID=0`
  Optionaler Parameter, der einen eindeutigen Identifikator der übergeordneten Transaktion enthält. Im Falle aktiver Bausteine sowie Kommunikation über Prozeßgrenzen in Verbindung mit dem `initiatedTA`-Aufruf verwendet, um eine korrekte Zuordnung zu gestatten.

Beim Aufruf von `startTA` wird vom Meßobjekt anhand des aktuellen Kontrollflusses die Zuordnung zu einer BTA hergestellt. Die aktuelle Zeit sowie die übergebenen Informationen werden abgespeichert.

- `stopTA(status: String, info: String)`
  Aufruf, um das Ende einer Subtransaktion anzuzeigen. Wird unmittelbar nach Rückkehr des Aufrufs einer Operation eines Bausteins aufgerufen.

  Parameter:

  - `status: String`
    Gibt an, ob die Subtransaktion erfolgreich beendet wurde oder nicht.
    Erfolgreiche Bearbeitung wird durch `Success`, nicht erfolgreiche durch `Failure` repräsentiert. Läßt sich der Erfolg der Bearbeitung nicht angegeben, so ist `unknown` zu übergeben.
  - `info: String`
    Zur Übergabe beliebiger weiterer Information z.B. hinsichtlich Fehlerursache etc.

  Bei Aufruf von `stopTA` wird vom Meßobjekt anhand des aktuellen Kontrollflusses die Zuordnung zu einer BTA hergestellt. Der aktuelle Zeitpunkt sowie die Information über Erfolg bzw. Mißerfolg der Subtransaktion werden gespeichert.

- `addControlFlow(newControlFlow: ThreadID, target: OID)`
  Aufruf, um dem Meßobjekt anzuzeigen, daß ein weiterer Kontrollfluß an der Erbringung einer BTA beteiligt ist.

Parameter:

  - `newControlFlow: ThreadID`
    Identifikator des neuen Kontrollflusses, der an der Erbringung der BTA beteiligt wird.
  - `target: OID`
    Identifikator des Bausteins, der im zu starteten Kontrollfluß ausgeführt werden soll.

Bei Aufruf von `addControlFlow` stellt das Meßobjekt die Zuordnung des neuen Kontrollflusses zur im aktuellen Kontrollfluß ausgeführten BTA her. Der aktuelle Zeitpunkt wird als der Beginn einer neuen Subtransaktion gespeichert. Dies kann (muß aber nicht) durch einen internen Aufruf von `startTA` erfolgen.

- `removeControlFlow(status: String, reason: String)`

  Aufruf, um dem Meßobjekt anzuzeigen, daß ein Kontrollfluß nicht weiter an der Erbringung einer BTA beteiligt ist.

  Parameter:

  - `status: String`
    Gibt an, ob das Verlassen der Bearbeitung ein erfolgreiches oder nicht erfolgreiches Ende einer Subtransaktion darstellt.
    Erfolgreiche Bearbeitung wird durch `Success`, nicht erfolgreiche durch `Failure` repräsentiert. Läßt sich der Erfolg der Bearbeitung nicht angegeben, so ist `unknown` zu übergeben.
  - `reason: String`
    Grund für das Verlassen der Bearbeitung der BTA (beliebiger Text). Beispiele hierfür sind `Ende des Kontrollflusses` oder `Rückkehr zur Oberfläche`.

  Bei Aufruf von `removeControlFlow` löst das Meßobjekt die Zuordnung des aktuellen Kontrollflusses zur aktuellen BTA. Der aktuelle Zeitpunkt wird als das Ende einer Subtransaktion gespeichert. Dies kann (muß aber nicht) durch einen internen Aufruf von `stopTA` erfolgen.

- `initiatedTA(isAsynchronous: Boolean):UUID`
  Aufruf, um dem Meßobjekt anzuzeigen, daß eine Transaktion angestoßen wurde, die in einem nicht automatisch korrelierbaren Kontrollfluß ausgeführt wird. Wird beispielsweise bei der Instrumentierung von aktiven Bausteinen oder bei der Kommunikation über Prozeß- und Systemgrenzen verwendet.

Parameter:

- `isAsynchronous: Boolean`
  Gibt an, ob die Transaktion synchron oder asynchron angestoßen wurde.

Rückgabewert:

- `UUID`
  Ein global eindeutiger Identifikator.

Bei Aufruf von `initiatedTA` wird ein global eindeutiger Identifikator generiert und zurückgegeben. Dieser wird gemeinsam mit dem aktuellen Zeitpunkt sowie der Information über synchronen oder asynchronen Anstoß der Transaktion gespeichert, um eine spätere Zuordnung von `startTA`-Aufrufen mit demselben Identifikator zu ermöglichen.

- `logInfo(info: String)`
  Aufruf zur Übergabe beliebiger Log-Information eines Bausteins an das Managementsystem.

  Parameter:

  - `info`
    Freier Text, der vom Bausteinentwickler festgelegt werden kann.

  Beim Aufruf von `logInfo` bestimmt das Meßobjekt anhand des aktuellen Kontrollflusses die aktuelle BTA und speichert den übergebenen *String* gemeinsam mit dem aktuellen Zeitpunkt ab.

Managementschnittstelle
Über die Managementschnittstelle werden zwischen dem Meßobjekt und dem zugehörigen Managementagenten Informationen über laufende bzw. abgeschlossene Transaktionen ausgetauscht. Um die Managementinformation implementierungsunabhängig übergeben zu können, wurde eine XML-Repräsentation der Information gewählt. Die übergebene Managementinformation orientiert sich an der im Rahmen der Arbeiten der *DAP/Metrics WG* der DMTF (vgl. Abschnitt 4.2.1.2) spezifizierten Information. Da das *DAP/Metrics* Schema der DMTF allerdings keine Möglichkeit vorsieht, zwischen synchron angestoßenen und asynchron angestoßenen Subtransaktionen einer Transaktion zu unterscheiden und auch keine explizite Unterscheidung zwischen Benutzertransaktionen und ihren Subtransaktionen möglich ist, wurde im Rahmen der vorliegenden Arbeit eine *Document Type Definition (DTD)* entwickelt, die diese Möglichkeiten vorsieht. Die DTD ist in Abbildung 5.25 dargestellt.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!ELEMENT bta (userName?, btaName, time, duration, status,
    completed, info?, syncTA)>
<!ELEMENT taInfo (componentClass, componentInstanceID,
    taName, duration, status, info?, parentID?, logInfo*)>
<!ELEMENT syncTA (taInfo, ta*)>
<!ELEMENT asyncTA (taInfo, ta*)>
<!ELEMENT ta (syncTA|asyncTA|taID)>
<!ELEMENT userName (#PCDATA)>
<!ELEMENT btaName (#PCDATA)>
<!ELEMENT time (#PCDATA)>
<!ELEMENT status (#PCDATA)>
<!ELEMENT duration (#PCDATA)>
<!ELEMENT completed (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT componentClass (#PCDATA)>
<!ELEMENT componentInstanceID (#PCDATA)>
<!ELEMENT taName (#PCDATA)>
<!ELEMENT parentID (#PCDATA)>
<!ELEMENT logInfo (time, info)>
<!ELEMENT taID (#PCDATA)>
     <!ATTLIST taID isAsynchronous (true|false) #REQUIRED>
```

Abbildung 5.25: DTD der an der Managementschnittstelle übergebenen Information

Jede BTA enthält neben den Informationen über den anstoßenden Benutzer, dem Namen der BTA, der Startzeit und der Antwortzeit, dem Status und optionaler Zusatzinformation eine (synchrone) Subtransaktion, die die auslösende BI beschreibt. Diese enthält detaillierte Information über den ausführenden Baustein und den Namen der angestoßenen Subtransaktion sowie wiederum die Dauer der Subtransaktion, den Status und optionale Zusatzinformation. Darüber hinaus kann diese nun aber aus beliebigen weiteren Transaktionen zusammengesetzt sein, die sowohl synchron als auch asynchron angestoßen sein können. Wurde eine Subtransaktion auf einem entfernten System angestoßen, so ist ein Platzhalter vorgesehen, der die spätere Zuordnung der entfernten Transaktion anhand eines eindeutigen Identifikators gestattet. In Abschnitt 6.2.3 wird ein zu dieser DTD konformes Beispiel für ein XML-Dokument angegeben.

5.4.3 Architektur für die Automation der Managementinstrumentierung

Dieser Abschnitt stellt die Architektur für die Automation der Managementinstrumentierung dar. Wiederum wird zunächst ein Überblick über die Komponenten der Architektur gegeben. Im Anschluß daran werden Methodiken vorgestellt, nach denen Anwendungs- bzw. Bausteinentwickler bei der Erstellung von Anwendungen bzw. Bausteinen zu verfahren haben, um mit geringem Aufwand instrumentierte Anwendungen erzeugen zu können.

5.4.3.1 Architekturübersicht

Die Architektur für die Automation der Managementinstrumentierung bausteinbasierter Anwendungen besteht im wesentlichen aus Bausteinen und einer erweiterten Entwicklungsumgebung. Der Großteil der Bausteine erfordert hierbei keinerlei spezielle Anpassung, gewisse Bausteine müssen allerdings vom Bausteinentwickler instrumentiert werden. Das grundsätzliche Prinzip wurde bereits in Abbildung 5.7 dargestellt: Eine Entwicklungsumgebung wird verwendet, um vorgefertigte Bausteine zu einer Anwendung zusammenzufügen. Dabei werden automatisch Meßpunkte in die Anwendung eingefügt.

Instrumentierte Bausteine
Einen wesentlichen Teil der Lösung stellen instrumentierte Bausteine dar. Auch wenn die Instrumentierung von Bausteinen für den jeweiligen Bausteinentwickler mit zusätzlichen Aufwand verbunden ist, läßt sie sich an einigen Stellen nicht vollständig vermeiden. Dies sind zum einen die Oberflächenbausteine, die mittels Benutzerinteraktionen (BIs) mit dem (menschlichen) Benutzer des Anwendungsdienstes in Kontakt treten, zum anderen die aktiven Bausteine, die ohne Instrumentierung keine Korrelation der von ihnen erbrachten Subtransaktion zur zugehörigen BTA gestatten würden. Die in Abschnitt 5.4.3.2 vorgestellte Methodik für den Bausteinentwickler beschreibt im Detail, wie eine Instrumentierung dieser Bausteine zu erfolgen hat.

Alle weiteren Bausteine können ohne jegliche Instrumentierung in die Anwendung aufgenommen werden und werden aufgrund der automatischen Instrumentierung der Anwendungslogik ohne manuelles Eingreifen des Anwendungs- bzw. Bausteinentwicklers in die Überwachung mit einbezogen. Um die Fehlerdiagnose zu erleichtern, können sie allerdings instrumentiert werden, dem Managementsystem zusätzliche Information über mögliche Fehlerursachen zu übergeben.

Entwicklungsumgebung
Von herausragender Bedeutung für die Automation der Managementinstrumentierung ist eine spezielle, erweiterte Entwicklungsumgebung, die während der Anwendungserstellung Managementaufrufe in die generierte Anwendungslogik einfügt. Von den in Ab-

schnitt 2.1.1 vorgestellten Bausteinarchitekturen kommen demnach insbesondere *JavaBeans* sowie *AcitveX* in Frage, da diese Architekturen die Generierung von Adaptern durch Entwicklungsumgebungen vorsehen.

Die Entwicklungsumgebung verwendet sowohl instrumentierte als auch nicht instrumentierte Bausteine. Die Instrumentierung eines Bausteins ist dabei für die Entwicklungsumgebung jedoch transparent, d.h. es ist keine Anpassung der Entwicklungsumgebung für den Einsatz instrumentierter Bausteine erforderlich. Sowohl aktive Bausteine als auch Bausteine, die zur Identifikation interner Fehlerzustände instrumentiert wurden, können wie herkömmliche Bausteine in die Anwendung integriert werden. Bei Verwendung von Oberflächenbausteinen muß vom Anwendungsentwickler die BI identifiziert und benannt werden, die den Start einer BTA darstellt. Dies geschieht transparent für die Entwicklungsumgebung im Rahmen des *Customizing* des Bausteins und wird im Rahmen der Vorstellung der Methodik für den Anwendungsentwickler in Abschnitt 5.4.3.2 eingehend vorgestellt.

Die wesentliche Erweiterung der Entwicklungsumgebung bezieht sich auf das automatische Einfügen von Meßpunkten zur Messung von Subtransaktionen. Jeweils vor Aufruf einer Methode eines Bausteins und nach Rückkehr von der Bearbeitung dieser Methode muß ein Meßpunkt eingefügt werden. Vor dem Aufruf einer Methode muß ein Aufruf von `startTA` eingefügt werden, der dem Meßobjekt den Beginn einer neuen Subtransaktion anzeigt. Bei der Rückkehr aus dem Methodenaufruf muß überprüft werden, ob eine erfolgreiche Bearbeitung erfolgte oder ob eine *Exception* ausgelöst wurde. Diese Information wird im Rahmen des anschließenden `stopTA`-Aufrufes ebenfalls an das Meßobjekt übermittelt. Das in Abbildung 5.26 dargestellte Beispiel zeigt den in Abschnitt 2.1.1.1 bereits vorgestellten Adapter zur Verknüpfung zweier *JavaBeans*, wobei zu dessen Generierung diesmal eine erweiterte Entwicklungsumgebung verwendet wurde. Der zusätzlich zum Standardadapter generierte *Code* ist durch fetteren Druck hervorgehoben. Wie zu erkennen ist, wurde vor dem Aufruf der Methode des Zielbausteins ein `startTA`-Aufruf eingefügt. Somit kann der Beginn der neuen Subtransaktion gemessen werden. Der eigentliche Aufruf wurde in eine `try/catch`-Umgebung verlegt, um evtl. auftretende *Exceptions* abfangen zu können. Im Falle einer *Exception* wird dem Meßobjekt das fehlerhafte Ende der Subtransaktion durch Übergabe von `Failure` als Parameter des `stopTA`-Aufrufes übermittelt und die *Exception* erneut geworfen, um den Programmablauf nicht zu stören. Kehrt der Aufruf der Methode fehlerfrei zurück, so wird dem Meßobjekt mit Hilfe eines `stopTA(Success)`-Aufrufes die fehlerfreie Beendigung der Subtransaktion übermittelt.

Generiert eine Entwicklungumgebung die entsprechenden Adapter nicht vollständig, wie z.B. das in Abschnitt 2.1.1.1 vorgestellte *Visual Basic*, so kann das dem Anwendungsentwickler zur Verfügung gestellte Adaptergerüst um die entsprechenden Zeilen erweitert werden. Der Entwickler muß dann lediglich den gewünschten Aufruf des Zielbausteins an der geeigneten Stelle einfügen.

Wird die Anwendungslogik nicht von einer Entwicklungsumgebung generiert, sondern

```
// Automatically generated event hookup file.

package tmp.sunw.beanbox;
import sunw.demo.molecule.Molecule;
import java.awt.event.MouseListener;
import java.awt.event.MouseEvent;

public class ___Hookup_16d4439d80 implements ja-
va.awt.event.MouseListener,
        java.io.Serializable {

     public void setTarget(sunw.demo.molecule.Molecule t) {
         target = t;
     }

     public void mouseClicked(java.awt.event.MouseEvent arg0) {
         startTA();
         try{
              target.rotateOnX();
         }
         catch(exception e) {
              stopTA(Failure);
              throw(e);
         }
         stopTA(Success);
     }

     ...

     private sunw.demo.molecule.Molecule target;
}
```

Abbildung 5.26: Beispiel für einen automatisch instrumentierten Adapter vom Anwendungsentwickler (beispielsweise im *Client*) von Hand erstellt, so muß der entsprechende *Code* ebenfalls manuell in die erstellte Anwendung eingebracht werden. Selbst dann ist bei Verwendung der in Abschnitt 5.4.3.2 vorgestellten Methodik aber eine wesentliche Vereinfachung zu erreichen.

5.4.3.2 Methodiken

Der folgende Abschnitt soll Methodiken für den Baustein- und Anwendungsentwickler vorgeben, die die einfache Erstellung instrumentierter Anwendungen garantieren. Die Einfachheit der vorgestellten Methodiken zeigt, wie stark die Managementinstrumentierung bausteinbasierter Anwendungen durch Verwendung der oben vorgestellten Architektur vereinfacht werden kann.

Methodik für den Bausteinentwickler
Jeder Bausteinentwickler sollte bei der Erstellung von Bausteinen nach der folgenden Methodik verfahren, um Bausteine zu erzeugen, die die oben beschriebene Managementlösung unterstützen:

Zunächst ist zu entscheiden, ob der zu erstellende Baustein überhaupt eine spezielle Instrumentierung benötigt. Dies ist nur für Oberflächen- und aktive Bausteine der Fall. Andere Bausteine als die oben angegeben erfordern keinerlei spezielle Managementinstrumentierung, können aber optional instrumentiert werden. Je nach Art des Bausteins sind dann die folgenden Schritte durchzuführen:

- Eingabebaustein
  - Identifikation aller BIs des Bausteins, die den Beginn einer BTA darstellen können.
  - Für jede identifizierte BI:
    - Einfügen einer Variablen (und geeigneter Methoden zum Zugriff auf die Variable), die zum Zeitpunkt der Anwendungserstellung im Rahmen des *Customizing* des Bausteins mit dem Namen der zugehörigen BTA belegt werden kann.
    - Einfügen von `startBTA` unmittelbar nach Entgegennahme der BI und `removeControlFlow` unmittelbar vor einer Rückkehr des Kontrollflusses zur Oberfläche. Diese Aufrufe dürfen nur dann erfolgen, wenn vom Anwendungsentwickler die zugehörige Variable gesetzt worden ist, die BI also tatsächlich den Beginn einer zu überwachenden BTA darstellt. Falls möglich, soll der Aufruf von `removeControlFlow` Information über die erfolgreiche bzw. nicht erfolgreiche Ausführung der zugehörigen Subtransaktion enthalten.

```
private String mouseReleasedBTA;

String getMouseReleasedBTA(){
       return mouseReleasedBTA;
}

public void setMouseReleasedBTA(String newBTA) {
       mouseReleasedBTA = newBTA;
}

public void mouseReleased(MouseEvent evt) {
       if(mouseReleasedBTA){
       Messung.startBTA(mouseReleasedBTA);
       }
       try{
          fireAction();
       }
       catch(Exception e){
          if(mouseReleasedBTA){
             Messung.removeControlFlow("Failure");
       }
       if(mouseReleasedBTA){
          Messung.removeControlFlow("Success");
       }
       repaint();
}
```

Abbildung 5.27: Beispiel für die Instrumentierung eines Eingabebausteins

Das in Abbildung 5.27 dargestellte *Code*-Segment zeigt am Beispiel eines *JavaBeans Buttons*, welche Anpassungen erforderlich sind. Die einzufügenden Zeilen sind durch fetteren Druck hervorgehoben.

Zunächst muß eine private Variable angelegt werden, die den Namen der gestarteten BTA enthalten kann. In diesem Fall soll die BI `mouseReleased` des *Buttons* instrumentiert werden, also das Loslassen des *Buttons* mit der Maus. Dies stellt typischerweise die BI dar, die bei einem *Button* zum Start einer BTA führt. Es wird also eine Variable `mouseReleasedBTA` vom Typ `String` angelegt, sowie (nach *JavaBeans*-Konvention) die beiden Methoden `getMouseReleasedBTA` und `setMouseReleasedBTA` zum Auslesen und Setzen der Variable durch die Entwicklungsumgebung.

Innerhalb der Methode `mouseReleased` wird mit Hilfe des Aufrufs von `fireAction` das eigentliche *Event* ausgelöst. Direkt vor Aufruf von `fireAction` wird folglich `startBTA` eingefügt und der Name der gestarteten BTA als Parameter übergeben. Der Aufruf von `fireAction` findet in einer `try/catch`-Umgebung statt. Nach Rückkehr von diesem Aufruf hat der Kontrollfluß die Bearbeitung der BTA sicher verlassen und es kann `removeControlFlow` aufgerufen werden. Je nachdem, ob eine *Exception* auftrat oder nicht, wird `Success` oder `Failure` übergeben. Diese Aufrufe finden nur statt, wenn vom Anwendungsentwickler für `mouseReleasedBTA` eine Belegung definiert wurde, die Überwachung für diese BI also gewissermaßen aktiviert wurde.

- Präsentationsbaustein
  - Identifikation aller BIs des Bausteins, die das Ende einer BTA darstellen könnten
  - Für jede identifizierte BI:
    - Einfügen einer Variablen vom Typ `boolean` (und geeigneter Methoden zum Zugriff auf die Variable), die zum Zeitpunkt der Anwendungserstellung im Rahmen des *Customizing* des Bausteins mit dem Wert `true` belegt werden können.
    - Einfügen von `stopBTA` unmittelbar vor Beginn der jeweiligen BI. Dieser Aufruf darf nur dann erfolgen, wenn die zugehörige Variable auf `true` gesetzt wurde.

  Die Instrumentierung von Präsentationsbausteinen erfolgt also analog zur Instrumentierung von Eingabebausteinen.

- aktiver Baustein
  - Erweiterung aller Methoden, über die ein Auftrag an einen aktiven Baustein erteilt werden kann. Folgende Schritte sind innerhalb dieser Methoden durchzuführen:
    - Aufruf von `initiatedTA` um dem Meßobjekt die Auftragserteilung bekanntzugeben. Im Falle aktiver Bausteine handelt es sich um einen asynchronen Anstoß einer Subtransaktion, was mit Hilfe des Parameterwertes `async` übermittelt wird.
    - Speicherung des als Rückgabewert erhaltenen Identifikators gemeinsam mit der Information über den zu erledigenden Auftrag.
  - Erweiterung der Auftragsbearbeitung:
    - Aufruf von `startTA` zu Beginn der Auftragsbearbeitung. Als Parameter muß der zuvor gespeicherte Identifikator der aufrufenden Transaktion übergeben werden.

```
public void erteileAuftrag(Auftrag newAuftrag){
       private int Id;
       Id = Messung.Initiated_TA(true);

       // Stelle Auftrag newAuftrag in Warteschlange

       // Stelle Id in Warteschlange

       }

private void bearbeiteAuftrag(){

       // Hole Auftrag newAuftrag aus Warteschlange

       // Hole Id aus Warteschlange

       Messung.startTA(bausteinName, bausteinID, Id);

       // Bearbeite Auftrag newAuftrag

       Messung.removeControlFlow("Success");
       }
```

Abbildung 5.28: Beispiel für die Instrumentierung eines aktiven Bausteins

∗ Aufruf von `removeControlFlow` zum Ende der Bearbeitung des Auftrags um anzuzeigen, daß der Kontrollfluß die Bearbeitung der BTA wieder verlassen hat und die Subtransaktion beendet wurde. Wiederum ist die Angabe des Erfolgs bzw. Mißerfolgs der Bearbeitung möglich.

Der einzufügende *Code* wird wiederum durch die schwarz gedruckten Zeilen des in Abbildung 5.28 dargestellten *JavaBeans*-Beispiels verdeutlicht. In diesem Beispiel wird davon ausgegangen, daß der aktive Baustein Aufträge als Parameter des Aufrufs `erteileAuftrag` entgegennimmt. In dieser Methode ist ein Aufruf von `initiatedTA` einzufügen und der Rückgabewert zu speichern. Neben dem Abspeichern des Auftrages in der Warteschlange muß nun dieser Rückgabewert ebenfalls in der Warteschlange gespeichert werden.

Die Methode `bearbeiteAuftrag` läuft in einem anderen Kontrollfluß und wird in regelmäßigen Abständen aktiviert, um die Warteschlange auf Aufträge zu überprüfen. Ist ein Auftrag vorhanden, so muß zusätzlich dessen zuvor gespeicherter Identifikator ausgelesen werden. Dann muß vor der eigentlichen Bearbeitung des Auftrags ein `startTA`-Aufruf und nach Beendigung der Bearbeitung ein `removeControlFlow`-Aufruf (im Beispiel mit Parameter „Success“) eingefügt werden. Der `startTA`-Aufruf erhält dabei (neben dem üblichen Bausteinnamen sowie einem Identifikator der Bausteininstanz) den Identifikator der Transaktion als Parameter.

Darüber hinaus wird kein zusätzlicher Aufwand vom Bausteinentwickler gefordert. Selbstverständlich sind auch beliebige Kombinationen der beschriebenen Bausteinklassen denkbar, also z.B. aktive Präsentationsbausteine ober kombinierte Eingabe- und Präsentationsbausteine.

Durch Einfügen von `logInfo`-Aufrufen in beliebige Bausteine ist es optional möglich, weitere Information (z.B. interne Fehlerzustände) zu übermitteln. Diese können beliebig eingefügt werden und müssen nicht explizit mit einer Transaktion korreliert werden, da dies von der vorgestellten Architektur vollständig transparent erbracht wird.

Methodik für den Anwendungsentwickler

Ein Anwendungsentwickler sollte bei der Erstellung von bausteinbasierten Anwendungen nach der folgenden Methodik verfahren, um korrekt managementinstrumentierte Anwendungen zu erhalten.

- Bestimmung der Antwortzeit

  Um die Antwortzeiten einer Anwendung bestimmen zu können, ist es erforderlich, jeweils den Beginn sowie den Zeitpunkt der Ergebnispräsentation aller zu überwachenden BTAs der Anwendung zu identifizieren. Dies geschieht folgendermaßen:

  – Identifikation der zu überwachenden BTAs aufgrund der Dienstvereinbarung/-beschreibung
  – Identifikation der Bausteine und BIs, die den Start bzw. mögliche Enden der BTAs darstellen
  – Identifikation von Startpunkt und Zeitpunkt der Ergebnispräsentation einer BTA durch:
    * Eindeutige Benennung der BTAs durch *Customizing* der jeweiligen Eingabebausteine.
    * *Customizing* der Präsentationsbausteine (falls vorhanden), um mögliche Zeitpunkte der Ergebnispräsentation festzulegen. Falls möglich, Definition von erfolgreicher bzw. nicht erfolgreicher Bearbeitung der BTA.

- Überwachung der Subtransaktionen

  Die Methodik für die Instrumentierung zur Überwachung der Subtransaktionen hängt stark von der Art der Verknüpfung der Bausteine in der eingesetzten Bausteinarchitektur ab:

- Verknüpfung durch generierte Adapter

  In diesem Fall ist kein weiteres Eingreifen des Anwendungsentwicklers erforderlich, die Instrumentierung kann vollständig automatisiert durch die in Abschnitt 5.4.3.1 vorgestellte erweiterte Entwicklungsumgebung erfolgen.
- Verknüpfung durch teilweise generierte Adapter

  In diesem Fall wird dem Anwendungsentwickler das Gerüst eines Adapters inklusive der entsprechenden Aufrufe des Meßobjektes zur Verfügung gestellt. Es verbleibt lediglich das Einfügen des Aufrufs eines geeigneten Bausteins zwischen der `startTA`- und der `stopTA`-Anweisung. Will der Entwickler mehr als einen Aufruf innerhalb eines Adapters realisieren, so ist analog der unter *Verknüpfung im Client* beschriebenen Methodik (siehe unten) vorzugehen.
- Verknüpfung im *Client*

  In diesem Fall kann keine Werkzeugunterstützung angeboten werden. Der Anwendungsentwickler muß jeweils vor Aufruf eines Bausteins eine `startTA`-Anweisung sowie nach Rückkehr von einem Aufruf eine `stopTA`-Anweisung einfügen. Die `stopTA`-Anweisung muß als Parameter den Erfolg bzw. Mißerfolg der ausgeführten Methode des Bausteins enthalten (z.B. durch Abfangen von *Exceptions*).

Darüber hinaus wird kein zusätzlicher Aufwand vom Anwendungsentwickler gefordert; insbesondere die Zuordnung der einzelnen Meßwerte zu den jeweiligen Transaktionen kann vollständig automatisiert erfolgen.

5.5 Bewertung

5.5.1 Vergleich mit der ARM API

In der folgenden Aufzählung wird die vorgestellte Architektur der ARM API (vgl. Abschnitt 4.2.1.3) gegenübergestellt, die im Bereich der Anwendungsinstrumentierung aktuell den größten Verbreitungsgrad besitzt. Somit werden die wesentlichen Weiterentwicklungen der vorgeschlagenen Lösung nochmals herausgestellt:

- Methodik für die Instrumentierung

  Die ARM API liefert lediglich eine Schnittstelle, die vom Anwendungsentwickler aufgerufen werden kann. Sie gibt keinerlei Methodik an, wie bei der Erstellung von Anwendungen vorzugehen ist, um instrumentierte Anwendungen zu erhalten. Insbesondere die Identifikation der relevanten Meßpunkte muß vom Anwendungsentwickler ohne jegliche Vorgaben erfolgen.

Im Gegensatz dazu werden in der vorliegenden Arbeit Methodiken angegeben, die sowohl Baustein- als auch Anwendungsentwickler in der Erstellung instrumentierter Anwendungen wesentlich unterstützen. Bei Anwendung dieser Methodiken kann die Identifikation der relevanten Meßpunkte durch den Entwickler mit äußerst geringem Aufwand erfolgen.

- Automatisierung und Werkzeugunterstützung

  Im Rahmen der ARM API ist keinerlei Automatisierung oder Werkzeugunterstützung vorgesehen. Ein Entwickler muß die jeweiligen Managementanweisungen vollständig manuell in den erstellten *Code* einbringen.

  Die vorgeschlagene Lösung ermöglicht es jedoch, große Teile der Managementinstrumentierung vollständig automatisiert zu erbringen. Der nicht zu vermeidende Rest an manuellem Instrumentierungsaufwand wurde vollständig vom Anwendungs- auf den Bausteinentwickler übertragen, was zu einer weiteren Verringerung des Aufwands durch Mehrfachnutzung der instrumentierten Bausteine führt. Für den Anwendungsentwickler verbleibt lediglich die Konfiguration der instrumentierten Bausteine mit Hilfe grafischer Entwicklungsumgebungen.

- Korrelation von Subtransaktionen

  Ein wesentlicher Nachteil der ARM API ist die aufwendige Korrelation von Subtransaktionen. Diese erfolgt durch Generierung eines eindeutigen Identifikators, der als Parameter durch die zu überwachenden Anwendung geschleust werden muß. Dies erschwert neben der nachträglichen Instrumentierung von Anwendungen insbesondere auch die Instrumentierung bausteinbasierter Anwendungen erheblich, da es die Anpassung der Schnittstellen aller Bausteine erfordern würde.

  Im vorliegenden Ansatz wird die Korrelation der Subtransaktionen mit Hilfe der darunterliegenden Kontrollflüsse automatisch und für den Entwickler vollständig transparent erreicht. Dies stellt eine wesentliche Erleichterung für den Anwendungsentwickler – insbesondere im Falle der Erstellung bausteinbasierter Anwendungen – dar. Einzig bei Verwendung aktiver Bausteine muß eine ähnliche Technik wie bei Verwendung der ARM API eingesetzt werden. Hier muß der generierte Identifikator allerdings nicht über Bausteingrenzen propagiert werden, sondern muß lediglich innerhalb des zu überwachenden aktiven Bausteins übergeben werden. Somit stellt auch dies keine wesentliche Einschränkung dar.

5.5.2 Leistungsbewertung

Naturgemäß wird die Leistung eines Systems durch das Einfügen einer Managementinstrumentierung negativ beeinflußt und die Meßergebnisse somit verfälscht. Im Vergleich zu herkömmlichen Methoden für die Managementinstrumentierung von Anwendungen bedingt die vorgeschlagene Lösung sogar eine geringfügig stärkere Beeinflussung der überwachten Anwendung. Dies begründet sich mit der automatischen Korrelation von Messungen, die nicht nur bei Beginn und Ende jeder Subtransaktion, sondern auch bei Erzeugung und Beendigung von Kontrollflüssen, Managementaufrufe erfordert.

Im folgenden wird anhand von einfachen Beispielanwendungen, die mit Hilfe des in Kapitel 6 beschriebenen, *JavaBeans*-basierten Prototypen erstellt wurden, eine Abschätzung dieser Leistungseinbußen getroffen. Hierbei zeigt sich, daß selbst bei Verwendung eines nicht optimierten Prototypen die zusätzliche Beeinflussung der Anwendung gering ist und im Vergleich zur erheblichen Aufwandsverringerung bei der Anwendungserstellung problemlos toleriert werden kann. Die Messungen erfolgten auf einem *AMD Athlon* System mit einer Taktfrequenz von 1.1GHz unter dem Betriebssystem *Linux*. Um unabhängig von der aktuellen Last des Meßsystems zu sein wurden keine absoluten Ausführungszeiten sondern CPU-Nutzungszeiten gemessen.

Der Meßaufwand für eine typische BTA setzt sich aus folgenden Komponenten zusammen:

- Aufruf von `startBTA`
- Aufruf von `startTA` sowie `stopTA` für jede Subtransaktion der BTA
- Aufruf von `addControlFlow` sowie `removeControlFlow` für jeden in die Bearbeitung hinzugenommenen Kontrollfluß
- Aufruf von `stopBTA`

Die folgenden Abschnitte beschreiben den zusätzlichen Aufwand für die Messung von Subtransaktionen (`startTA`/`stopTA`) bzw. bei Erzeugung und Beendigung von Kontrollflüssen (`addControlFlow`/`removeControlFlow`). Da `startBTA` und `stopBTA` nur jeweils einmal innerhalb einer BTA aufgerufen werden, wird auf eine explizite Betrachtung dieser Aufrufe verzichtet. Der Aufwand hierfür ist in etwa mit dem für die Hinzunahme eines Kontrollflusses vergleichbar.

5.5.2.1 Start und Stop von Subtransaktionen

Für die Ermittlung des zusätzlichen Aufwands für die Ermittlung der Laufzeit von Subtransaktionen wurde folgende einfache Beispielanwendung erstellt: Ein Baustein wird (von einem Eingabebaustein) gestartet und ruft 1000 mal einen weiteren Baustein auf, der ein Feld von zufällig gewählten Gleitkommazahlen mit Hilfe des *BubbleSort*-Algorithmus sortiert.

Als Referenz wurde zunächst eine Messung ohne die vorgeschlagene Managementinstrumentierung vorgenommen. Daraufhin wurde dieselbe Anwendung nochmals mit vollständiger Managementinstrumentierung generiert und gemessen. Um statistische

Abweichungen auszugleichen, wurden jeweils 100 Messungen vorgenommen. Abbildung 5.29 zeigt die ermittelten Mittelwerte für die Antwortzeit einer einzelnen Subtransaktion in Abhängigkeit von der Größe des sortierten Feldes.

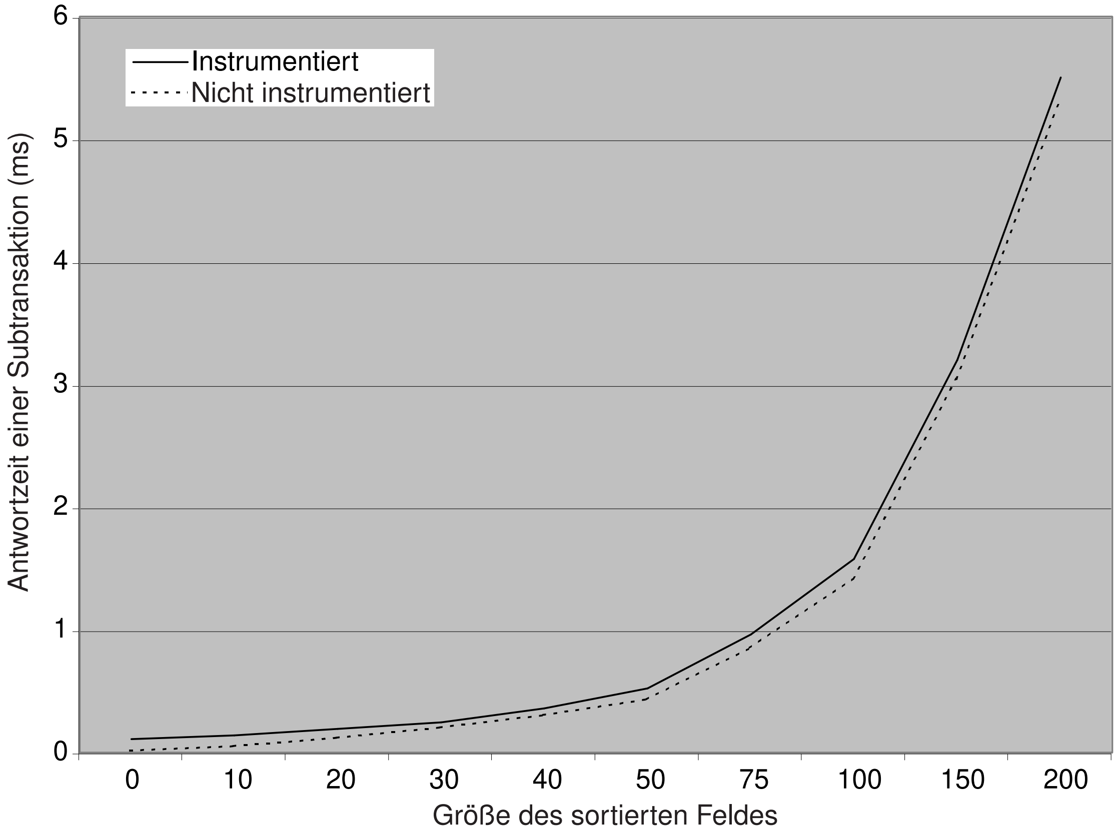

Abbildung 5.29: Antwortzeit einer Subtransaktion mit bzw. ohne Instrumentierung

Es zeigt sich, daß durch die Managementinstrumentierung nur eine unwesentliche Leistungseinbuße hervorgerufen wird. Auf dem Testsystem ergab sich für die Messung einer kompletten Subtransaktion (`startTA` und `stopTA`) lediglich ein zusätzlicher Aufwand von 0.1ms. Abbildung 5.30 stellt – wiederum abhängig von der Größe des sortierten Feldes – das Verhältnis von instrumentierter Laufzeit zu nicht instrumentierter Laufzeit dar. Man erkennt, daß bereits bei einer Feldgröße von 30 der resultiertende *Slowdown* nur annähernd 10% beträgt. Bei einem Feld der Größe 150 verlangsamt sich der Ablauf nurmehr um 3%. Da die Sortierung eines Feldes von 150 Zahlen eine relativ kurze Subtransaktion darstellt (auf dem Testsystem ca. 3ms) und in realen Szenarien mit Bausteinen zu rechnen ist, die zum Teil erheblich längere Laufzeiten erfordern, ist also von einer nur unwesentlichen Beeinflussung der zu überwachenden Anwendung durch die Instrumentierung auszugehen.

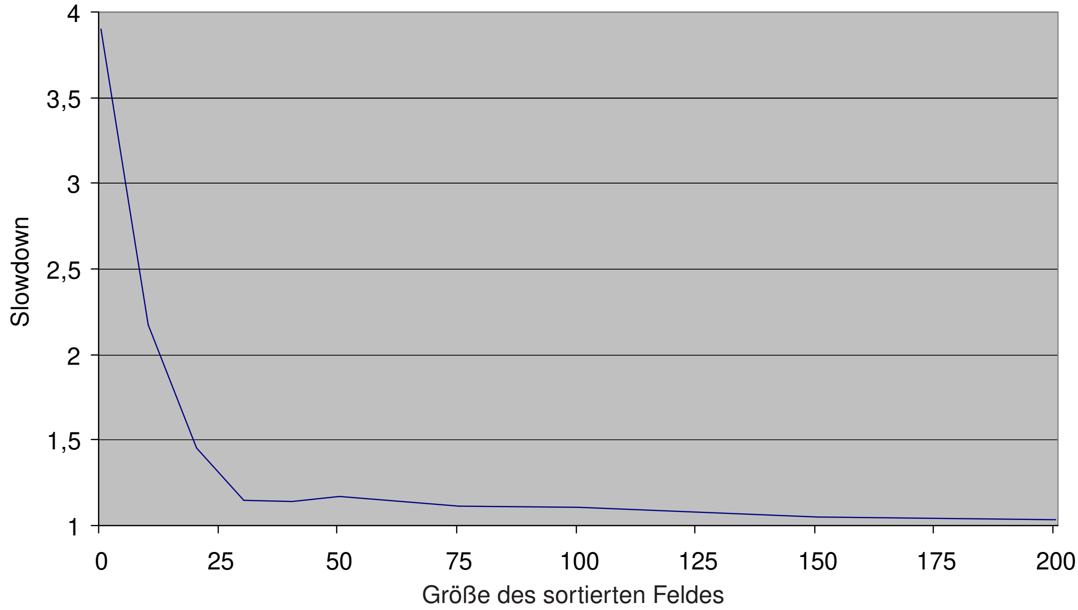

Abbildung 5.30: Slowdown durch Instrumentierung

5.5.2.2 Erzeugung und Beendigung von Kontrollflüssen

Um den zusätzlichen Aufwand bei Erzeugung und Beendigung von Kontrollflüssen angegeben zu können, wurde eine weitere Beispielanwendung erzeugt, bei der ein (durch einen Eingabebaustein aktivierter) Baustein 1000 mal einen weiteren Baustein aufruft, der lediglich einen neuen *Thread* erzeugt und zum Aufrufer zurückkehrt. Der neu erzeugte *Thread* wird unmittelbar nach seiner Erzeugung wieder beendet ohne eigene Funktionalität auszuführen.

Wiederum wurde sowohl mit als auch ohne Managementinstrumentierung jeweils 100 mal gemessen. Die Mittelwerte der Messungen sind in Abbildung 5.31 veranschaulicht. Es zeigt sich, daß selbst wenn im neu erzeugten Kontrollfluß keinerlei Berechnungen durchgeführt werden, durch die Managementinstrumentierung nur mit einer Leistungseinbuße von 6% zu rechnen ist. Auf dem Testsystem bedeutete dies eine Verlangsamung von ca. 0.02ms für jeden zusätzlich gestarteten *Thread*. Wird der neu gestartete *Thread* nicht unmittelbar wieder beendet sondern führt eigene Berechnungen aus, so nimmt die prozentuale Verlangsamung entsprechend weiterhin ab.

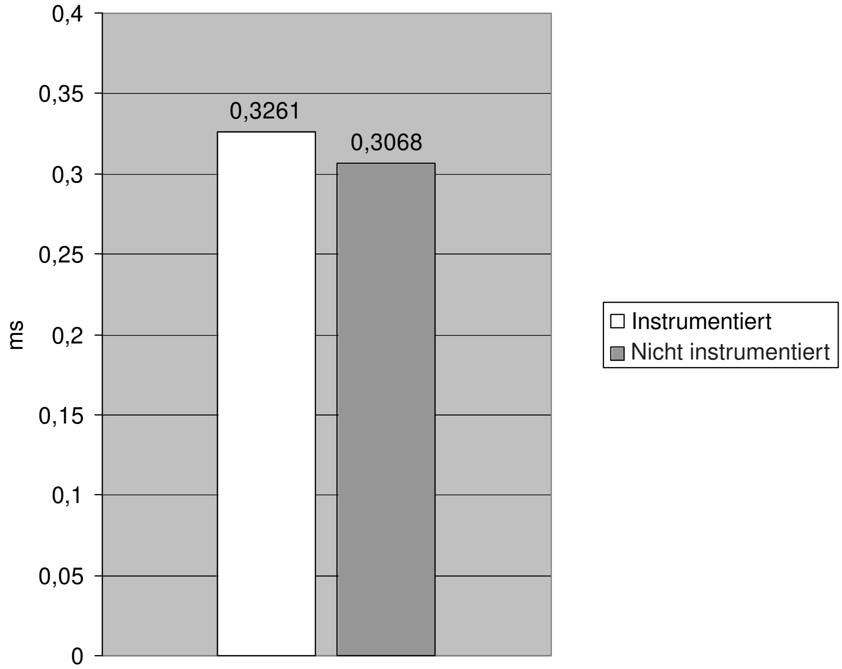

Abbildung 5.31: Generierung und Beendigung von Threads

5.6 Bestimmung der Verfügbarkeit bausteinbasierter Anwendungsdienste

Der folgende Abschnitt soll kurz beispielhaft aufzeigen, wie die mit Hilfe der vorgeschlagenen Architektur gewonnenen Ergebnisse eingesetzt werden können, um die Verfügbarkeit bausteinbasierter Anwendungsdienste bestimmen und vorhersagen zu können. Durch die dynamische Bestimmung der Abhängigkeiten zur Laufzeit der Anwendung wird es möglich, Ausfallwahrscheinlichkeiten abhängig von Dienst, BTA und Benutzer angeben zu können.

Verfügbarkeit eines Anwendungsdienstes

Die Verfügbarkeit eines Anwendungsdienstes soll hier definiert werden als das Verhältnis erfolgreich beantworteter Anfragen zur Gesamtanzahl der Anfragen. Diese kann sowohl für den gesamten Dienst als auch differenziert nach einzelnen BTAs bzw. Benutzern bestimmt werden.

Unter einer erfolgreich beantworteten Anfrage soll hier jede Anfrage verstanden werden, die innerhalb einer vorgegebenen Zeit ein korrektes Ergebnis liefern konnte.

Idee

Mit Hilfe der ermittelten Informationen ist es nicht nur möglich, die Verfügbarkeit eines Anwendungsdienstes durch Vergleich mit Schwellwerten nachträglich bestimmen zu können sondern ebenfalls eine Wahrscheinlichkeit für den Ausfall eines Dienstes bei Ausfall eines einzelnen Bausteins angeben zu können. Hierzu muß lediglich die Häufigkeit der Verwendung einzelner Bausteine bei Ausführung eines Dienstes ermittelt werden. Aufgrund dieser Informationen kann bei Ausfall eines Bausteines festgestellt werden, mit welcher Wahrscheinlichkeit die unterschiedlichen angebotenen Dienste von dem Ausfall betroffen sind.

Da es (z.B. durch Redundanzmechanismen) möglich ist, daß bei Ausfall eines Bausteins automatisch andere Bausteine zum Einsatz kommen, kann allerdings nur angegeben werden, welche Dienste mit einer bestimmten Wahrscheinlichkeit *nicht* von dem Ausfall betroffen sein werden. Dies ist aber bereits von großer praktischer Bedeutung, um bei Auftreten von Fehlfunktionen abschätzen zu können, welche wirtschaftlichen Auswirkungen durch den Ausfall zu erwarten sind und ob eine sofortige Reaktion auf den Ausfall erforderlich ist.

Durch differenzierte Auswertung der übermittelten Information wird es möglich, BTA- bzw. benutzerspezifische Wahrscheinlichkeiten angeben zu können. Dies ist von Bedeutung, da abhängig vom Nutzungsprofil einzelner Benutzer der Ausfall eines Bausteins für einen Benutzer zum Ausfall des abonnierten Dienstes führen kann, während ein anderer Benutzer den Dienst ohne jegliche Einschränkung weiterhin benutzen kann.

Bewertung

Bisherige Lösungen mußten zur Bestimmung der Verfügbarkeit von Diensten auf aufwendige, manuelle Verfahren zurückgreifen, die die Abhängigkeiten innerhalb der Anwendung statisch beschreiben (siehe z.B. [Kais 99]). Dies ist nicht nur aufwendig und fehleranfällig sondern kann die tatsächliche Situation in vielen Fällen auch nicht korrekt wiedergeben. Insbesondere die Differenzierung nach einzelnen Benutzern ist mit derartigen Verfahren aufgrund des hiermit verbundenen hohen Aufwands nicht möglich. Stattdessen kann nur ein gemeinsamer Abhängigkeitsgraph für alle Benutzer angegeben werden, der somit keine benutzerspezifischen Ergebnisse liefern kann.

Die in der vorliegenden Arbeit vorgeschlagene Architektur ermöglicht stattdessen die problemlose Bestimmung der tatsächlichen Abhängigkeiten durch Beobachtung der tatsächlich auftretenden Abläufe; die manuelle Erstellung von Abhängigkeitsgraphen entfällt vollständig. Durch weitgehende Automation der Managementinstrumentierung der zu überwachenden Anwendungen entsteht nur geringer zusätzlicher Aufwand für die Entwickler. Die zur Verfügung gestellten Informationen gestatten nicht nur die Überwachung der Verfügbarkeit in ihrer Gesamtheit sondern darüber hinaus eine einfache Differenzierung nach Benutzer bzw. gewünschter Transaktion.

Kapitel 6

Prototypische Realisierung

Im folgenden Kapitel wird eine prototypische Realisierung der in den vorangegangenen Abschnitten beschriebenen *Architektur für die Automation der Managementinstrumentierung bausteinbasierter Anwendungen* vorgestellt. Wie bereits in Abschnitt 5 gliedert sich die Darstellung in die Implementierung der *Architektur für die Automation der Managementinstrumentierung* (Abschnitt 6.1) sowie die Implementierung der *Architektur für die Ermittlung der Managementinformation* (Abschnitt 6.2). Im Anschluß daran wird in Abschnitt 6.3 anhand einfacher Beispielanwendungen der Einsatz der Architektur in der Praxis dargestellt.

Für die prototypische Realisierung wurde die *JavaBeans*-Architektur verwendet. Dies begründet sich mit der Verfügbarkeit des *Source Codes* von Entwicklungsumgebungen und *Java Virtual Machine*, was eine problemlose Erweiterung dieser Komponenten gestattete. Als Entwicklungsumgebung wurde die in Abschnitt 2.1.1.1 bereits vorgestellte *Beanbox* verwendet, ein einfaches Werkzeug zur Erstellung und Ausführung *JavaBeans*-basierter Anwendungen.

6.1 Implementierung der Architektur für die Automation der Managementinstrumentierung

Die Implementierung der *Architektur für die Automation der Managementinstrumentierung* ist in Abbildung 6.1 skizziert. Eine erste Version des Prototypen wurde in [Kali 01] erstellt. Im Rahmen der prototypischen Implementierung angepaßte Elemente sind grau hinterlegt dargestellt. Man kann erkennen, daß sowohl eine Erweiterung des Entwicklungswerkzeuges *Beanbox* sowie eine Instrumentierung ausgewählter *Beans* vorgenommen wurde. Zunächst wird dargestellt, welche Veränderungen an der *Beanbox* vorgenommen werden mußten, um automatisch Managementcode in zu erstellende Anwendungen einfügen zu können (Abschnitt 6.1.1). Für diejenigen Klassen von Bausteinen, für die eine automatische Instrumentierung nicht ausreicht (Oberflächen- bzw. aktive Bausteine) wird dann gezeigt, wie eine mögliche Instrumentierung aussehen könnte (Abschnitt 6.1.2).

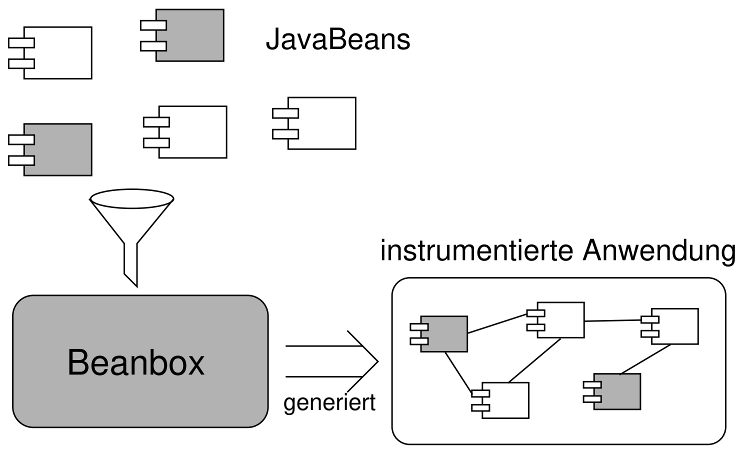

Abbildung 6.1: Implementierung der Architektur für die Automation der Managementinstrumentierung

6.1.1 Erweiterung des Entwicklungswerkzeuges Beanbox

Die erforderlichen Erweiterungen der *Beanbox* beschränken sich auf das Einfügen von `startTA`- bzw. `stopTA`-Aufrufen in die generierten Adapterklassen, also die Klassen, die die Verknüpfung zweier *Beans* realisieren, unmittelbar vor Aufruf einer *Bean* bzw. unmittelbar nach Rückkehr von einem Aufruf. Eine besondere Anpassung der *Beanbox* für die Integration instrumentierter *Beans* ist nicht erforderlich.

Die Generierung der Adapterklassen erfolgt in der *Beanbox* in der Klasse `HookupManager.java` mittels Aufrufen der `print`-Methode, die den *Javacode* der Adapterklasse erstellen. Somit konnte durch Einfügen weiterer Aufrufe von `print` die erforderliche Erweiterung auf einfache Art und Weise realisiert werden. Die eingefügten Anweisungen sind im in Abbildung 6.2 dargestellten Ausschnitt aus der Klasse `HookupManager.java` wiederum durch kräftigeren Druck hervorgehoben.

Unmittelbar bevor ein Methodenaufruf in die Adapterklasse eingefügt wird, wird nun ein `startTA`-Aufruf eingefügt, der dem Meßobjekt den Start der Subtransaktion meldet. Der eigentliche Methodenaufruf erfolgt nun in einer `try/catch`-Umgebung, die eventuell auftretende *Exceptions* (bzw. *Errors*) der aufgerufenen Methode abfängt. Je nachdem, ob eine *Exception* auftritt oder nicht, wird nach Rückkehr aus der aufgerufenen Methode ein Aufruf von `stopTA` mit dem geeigneten Wert für `status` eingefügt. Trat eine *Exception* auf, so wird diese nach Übermittlung an das Meßobjekt erneut geworfen, um den ursprüng-

```
public class HookupManager {
...
out.println("{");
if (listenerMethod.getName() == methods[k].getName()) {
   out.print("Messung.startTA(target.getClass().getName(),
      String.valueOf(System.identityHashCode(target)),\""+
      targetMethod.getName()+"\");\n");
   out.print("try {\n");
   out.print("  target." + targetMethod.getName() + "(");
   if (targetMethod.getParameterTypes().length != 0) {
     for (int i = 0; i < argTypes.length; i++) {
       if (i > 0) {
         out.print(", ");
       }
       out.print("arg" + i);
     }
   }
   out.println(");");
   out.println("}");
   out.println("catch (Error e){");
   out.println("   Messung.stopTA(\"Failed\",e.getMessage());");
   out.println("   throw e;}");
   out.println("catch (RuntimeException e){");
   out.println("   Messung.stopTA(\"Failed\",e.getMessage());");
   out.println("   throw e;}");
}
out.println("Messung.stopTA(\"Success\",null);");
out.println("}");
...
```

Abbildung 6.2: Erweiterung des Entwicklungswerzeugs Beanbox

lichen Programmablauf beizubehalten. Ein Beispiel für eine mit Hilfe dieser erweiterten Entwicklungsumgebung generierten Adapterklasse ist in Abbildung 6.3 dargestellt.

6.1.2 Instrumentierung ausgewählter JavaBeans

Wie in Abschnitt 5.4.3.1 dargestellt, ist für einige wenige Klassen von Bausteinen (Oberflächen- und aktive Bausteine) eine manuelle Instrumentierung durch den Bausteinentwickler erforderlich. Hierzu wurde in Abschnitt 5.4.3.2 eine Methodik angegeben, nach der der Bausteinentwickler bei der Instrumentierung verfahren sollte. Im folgenden wird

```
public class ___Hookup_16e8ce750f implements java.awt.
   event.ActionListener, java.io.Serializable {
...
   public void actionPerformed(java.awt.event.ActionEvent
      arg0) {
         Messung.startTA(target.getClass().getName(),String.
            valueOf(System.identityHashCode(target)),
            "starteSort");
         try {
              target.starteSort();
         }
         catch (Error e){
              Messung.stopTA("Failed",e.getMessage());
              throw e;
         }
         catch (RuntimeException e){
              Messung.stopTA("Failed",e.getMessage());
              throw e;
         }
         Messung.stopTA("Success",null);
     }
...
```

Abbildung 6.3: Von erweiterter Beanbox generierte Adapterklasse

anhand einfacher Beispiele gezeigt, wie eine Instrumentierung dieser Bausteine erfolgen könnte.

6.1.2.1 Instrumentierung von Eingabebausteinen

Die Instrumentierung von Eingabebausteinen wird anhand eines einfachen *Buttons* demonstriert, der für den Start beliebiger BTAs verwendet werden kann. Betätigt ein Benutzer diesen *StartButton*, so wird ein sogenanntes *ActionEvent* an alle registrierten *EventListener* versandt. Dies geschieht in der Methode `fireAction`, die wiederum von der in Abbildung 6.4 dargestellten Methode `mouseReleased` des *StartButtons* aufgerufen wird.

Vor Aufruf der Methode `fireAction` muß der Bausteinentwickler zunächst einen Aufruf von `startBTA` einfügen. Dieser übermittelt an das Meßobjekt den Namen der gestarteten BTA (festzulegen im Rahmen des *Customizing* des *StartButtons* durch den Anwendungsentwickler), den aktuellen Benutzernamen sowie Identifikatoren für Klasse und Instanz des aufrufenden Bausteins. Im Anschluß daran erfolgt der Aufruf von `fireAction`

```
public void mouseReleased(MouseEvent evt) {
   if (!isEnabled()) {
       return;
   }
   if (down) {
      Messung.startBTA(btaName, System.getProperties().
           getProperty("user.name"), "StartButton",
           String.valueOf(System.identityHashCode(this)));
      try{
            fireAction();
      }
      catch (Exception e){
           Messung.removeControlFlow("Failure", "Thread
                Rückkehr zur Oberfläche");
      }
      Messung.removeControlFlow("Success", "Thread Rückkehr
           zur Oberfläche");
      down = false;
      repaint();
    }
```

Abbildung 6.4: Instrumentierung eines StartButtons

für die Verteilung von *ActionEvents* an alle registrierten *Listener*. Nach Abschluß der Verteilung kehrt der Kontrollfluß zurück zur Oberfläche. Je nachdem, ob hierbei eine *Exception* auftrat oder nicht, wird dem Meßobjekt mit Hilfe des Aufrufs `removeControlFlow` ein erfolgreiches bzw. nicht erfolgreiches Ende der Subtransaktion angezeigt.

6.1.2.2 Instrumentierung von Präsentationsbausteinen

Präsentationbausteine müssen dahingehend instrumentiert werden, daß sie den Zeitpunkt der Ergebnispräsentation an das Meßobjekt mit Hilfe einer `stopBTA`-Anweisung übermitteln. Hierzu wurde eine *JavaBean* (*SortItem*), die die Sortierung einer gegebenen Menge von Zahlen durchführt dahingehend erweitert, daß bei Abschluß der Sortierung dies mittels `stopBTA` übermittelt wird, sofern der Anwendungsentwickler dies beim *Customizing* als das Ende einer BTA festgelegt hat.

6.1.2.3 Instrumentierung aktiver Bausteine

Ferner wurde ein aktiver Baustein (*ActiveBean*) so instrumentiert, daß er bei Übergabe eines Auftrages die Methode `initiatedTA` des Meßobjektes aufruft. Der zurückgelieferte

Identifikator wird gemeinsam mit der Information über den erteilten Auftrag gespeichert und bei Start der eigentlichen Bearbeitung mit Hilfe eines `startTA`-Aufrufes übergeben. Der Abschluß der Bearbeitung wird durch einen Aufruf von `removeControlFlow` an das Meßobjekt übermittelt. Wiederum ist die Angabe des Erfolgs bzw. Mißerfolgs der Bearbeitung möglich.

6.2 Implementierung der Architektur für die Ermittlung der Managementinformation

Abbildung 6.5 skizziert die *Architektur für die Ermittlung der Managementinformation* zur Laufzeit einer Anwendung. Hierzu war eine geringfügige Erweiterung von Systemklassen der *Java Virtual Machine (JVM)* erforderlich, was in der Skizze wiederum durch graue Hinterlegung dargestellt wird. Sowohl die instrumentierte Anwendung als auch die JVM liefern Managementinformation an das Meßobjekt. Dieses leitet die Information weiter an einen Managementagenten, der sie sammelt und an eine Managementanwendung übergibt. Die stärker umrandet dargestellten Elemente der Architektur (Meßobjekt, Managementagent, Managementanwendung) wurden im Rahmen der prototypischen Realisierung implementiert [Deil 01].

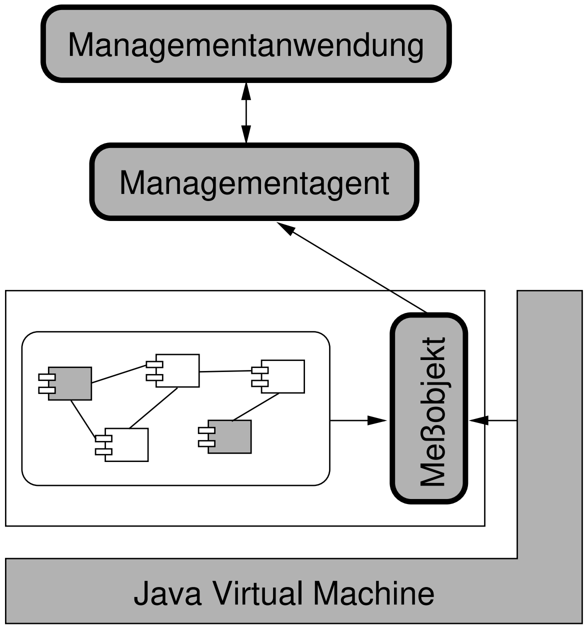

Abbildung 6.5: Implementierung der Architektur für die Ermittlung der Managementinformation

6.2.1 Implementierung des Meßobjekts

Es wurde ein Meßobjekt erstellt, das die in Abschnitt 5.4.2.2 spezifizierte Meßschnittstelle implementiert. Die Java-Implementierung der Meßschnittstelle ist in Abbildung 6.6 dargestellt.

Um den Zugriff von beliebigen Stellen innerhalb der zu überwachenden Anwendung sowie aus der JVM heraus zu ermöglichen, wurde eine Klasse `Messung` definiert, die die Meßschnittstelle mittels statischer Methoden zur Verfügung stellt. Somit ist inner-

```
public interface MessSchnittstelle {
    public void startBTA(String btaName, String userName,
            String bausteinName, String bausteinID);
    public void stopBTA(String status, String info);
    public void startTA(String bausteinName, String
            bausteinID, String taName);
    public void startTA(String bausteinName, String
            bausteinID, String taName, String parentTA);
    public void stopTA(String status, String info);
    public void addControlFlow(Thread newTh, Object Target);
    public void removeControlFlow(String status, String
            reason);
    public String initiatedTA(boolean isAsynchronous);
    public void logInfo(String info);
}
```

Abbildung 6.6: Java-Implementierung der Meßschnittstelle

halb der Anwendung keine explizite Referenz auf das Meßobjekt erforderlich. Die Klasse `Messung` instantiiert lediglich das eigentliche Meßobjekt und leitet die Aufrufe der Meßschnittstelle an das Meßobjekt weiter; darüber hinaus implementiert sie keine eigene Funktionalität.

Innerhalb des Meßobjektes wird mit Hilfe einer *Hash*-Tabelle eine Abbildung von *Threads* zu Instanzen von BTAs hergestellt. Bei jedem Aufruf einer Methode der Meßschnittstelle wird anhand des ausführenden *Threads* aus der eben beschriebenen *Hash*-Tabelle die Zuordnung zu einer Instanz einer BTA hergestellt. Die aktuelle Systemzeit wird bestimmt und gemeinsam mit weiteren Informationen in einem Vektor gespeichert. Um dies zu verdeutlichen, stellt Abbildung 6.7 die Implementierung der Methoden `startBTA` und `startTA` nochmals vereinfacht dar. In der Methode `startBTA` wird zunächst mit Hilfe der aktuellen Systemzeit sowie eines Zufallswertes, ein eindeutiger Identifikator für die Instanz der gestarteten BTA erzeugt. Dieser Identifikator wird dann gemeinsam mit dem Identifikator des aktuellen *Thread* (mit Hilfe der `put`-Methode) in die *Hash*-Tabelle `btaTab` eingefügt. In der Methode `startTA` (sowie in jeder anderen Methode des Meßobjekts) kann dann der Name des aktuellen *Thread* (mit Hilfe der `get`-Methode) aus ebendieser *Hash*-Tabelle wieder ermittelt werden. Die Informationen über die gestartete Subtransaktion (aktuelle Zeit, Bausteinname, Name der Subtransaktion, etc.) werden dann gemeinsam mit dem Identifikator der BTA in einer Datenstruktur `messPunkt` gesammelt und in dem Vektor `messTabelle` gespeichert.

In regelmäßigen Abständen wird ein *Thread* niedriger Priorität gestartet, der die im Vektor gesammelten Daten über eine *Socket*-Schnittstelle an den Managementagenten außer-

```
public void startBTA(String btaName, String userName,
          String bausteinName, String bausteinID){
   ...
   // eindeutigen Identifikator generieren
   String BTA_instanz = new String(Long.toString(System.
             currentTimeMillis()+randy.nextInt()));

   // Identifikator und Thread in Tabelle
   btaTab.put(Thread.currentThread().getName(),BTA_instanz);
   ...

}


public void startTA(String bausteinName, String bausteinID,
          String taName){

   // Identifikator der BTA-Instanz ermitteln
   String btaInstanz = (String)btaTab.get(Thread.
             currentThread().getName());

   // Messpunkt erzeugen
   if(btaInstanz!=null){
        MessPunkt messPunkt = new MessPunkt(btaInstanz,
                       System.currentTimeMillis(),"startTA",
                       Thread.currentThread().getName());
        messPunkt.componentClass=bausteinName;
        messPunkt.componentInstanceID=bausteinID;
        messPunkt.taName=taName;

    // Messpunkt in Messtabelle einfügen
        messTabelle.addElement(messPunkt);
    }
}
```

Abbildung 6.7: Java-Implementierung des Meßobjekts (Ausschnitt) halb des Prozesses der überwachten Anwendung übergibt. Durch die Verwendung einer *Socket*-Schnittstelle wurde sowohl die Möglichkeit geschaffen, die Information über Systemgrenzen hinweg zu übermitteln, als auch Unabhängigkeit von der gewählten Programmiersprache erreicht.

6.2.2 Instrumentierung von Systemklassen der Java Virtual Machine

Die vorgeschlagene Architektur erfordert es, den Start sowie die Beendigung von *Threads* erkennen zu können, um die automatische Zuordnung von Meßwerten zu Transaktionen zu ermöglichen. Um eine Korrelation von Managementinformation verteilter Systeme zu gestatten, müssen darüber hinaus die Mechanismen zur Interprozeßkommunikation instrumentiert werden. Da *JavaBeans* typischerweise nicht zur Erstellung verteilter Anwendungen Verwendung finden, konnte in Rahmen der prototypischen Implementierung auf eine Instrumentierung der Interprozeßkommunikationsmechanismen verzichtet werden. Diese könnte allerdings durch Instrumentierung der *Remote Method Invocation (RMI)* Mechanismen von *Java* jederzeit erfolgen.

Eine Möglichkeit zur Ermittlung der Information über den Start und die Beendigung von *Threads* stellt in Version 1.2 des *Java Development Kits (JDK)* das sogenannte *Java Virtual Machine Profiler Interface (JVMPI)* [JVMPI 99] bereit. Dies befindet sich aktuell allerdings noch in einem experimentellen Stadium und kann auch nur einen Teil der gewünschten Information liefern. Beispielsweise ist es mit Hilfe des JVMPI nicht möglich, das in einem neu gestarteten *Thread* auszuführende Objekt zu ermitteln.

Aus diesen Gründen wurde auf eine Verwendung des JVMPI verzichtet und stattdessen eine Instrumentierung der Klasse `java.lang.Thread` der JVM vorgenommen. Die Firma *Sun* stellt den *Source Code* der JVM zu derartigen Zwecken uneingeschränkt zur Verfügung [J2CS 01]. Vereinfacht ausgedrückt wurde die Methode `start` der Klasse `java.lang.Thread` dahingehend erweitert, daß sie einen Aufruf der Methode `addControlFlow` des Meßobjekts durchführt. So kann jeder Start eines *Threads* mit Sicherheit erkannt werden. Da die Methode `start` noch im aufrufenden *Thread* ausgeführt wird, ist eine Zuordnung des gestarteten *Threads* zu einer Instanz einer BTA zu diesem Zeitpunkt möglich. Desweiteren wurde in die Methode `exit` der Klasse `java.lang.Thread` ein Aufruf der Methode `removeControlFlow` eingefügt. Die Methode `exit` wird immer bei Beendigung eines *Threads* von der JVM aufgerufen und ermöglicht somit die sichere Identifikation der Beendigung beliebiger *Threads*.

Wenn zukünftig mit Hilfe des JVMPI (oder einer anderen Schnittstelle) neben der Benachrichtigung über den Start und das Ende von *Threads* auch das im neu gestarteten *Thread* auszuführende Objekt ermittelt werden kann, kann auf eine derartige Instrumentierung vollständig verzichtet werden und stattdessen eine Implementierung auf Basis des JVMPI eingesetzt werden.

6.2.3 Implementierung eines prototypischen Managementagenten

Der Managementagent nimmt über *Socket*-Verbindungen die ermittelte Managementinformation aller beteiligten Anwendungen und Systeme entgegen. Er generiert aus den übermittelten Informationen eine XML-konforme Beschreibung, die der in Abschnitt 5.4.2.2 beschriebenen DTD entspricht. Diese wird gespeichert und auf Anforderung beliebigen Managementanwendungen zur weiteren Verarbeitung zur Verfügung gestellt. Abbildung 6.8 stellt einen Ausschnitt aus einer derartigen XML-Beschreibung für eine einfache Benutzertransaktion `SortTA` dar.

6.2.4 Implementierung einer prototypischen Managementanwendung

Mit Hilfe der prototypischen Managementanwendung *StatisticView* [Deil 01] kann die ermittelte Managementinformation visualisiert werden. Diese erhält hierzu – wiederum über eine *Socket*-Verbindung – die XML-kodierte Information und stellt sie grafisch dar. Abbildung 6.9 zeigt die Oberfläche der erstellten Anwendung.

Im linken oberen Teil des dargestellten Fensters erkennt man die Möglichkeit zur Auswahl einer Klasse von BTAs (*Transaction(s) by Name*). Neben dem Namen der BTA wird statistische Information zur Anzahl erfolgreicher bzw. nicht erfolgreicher Instanzen dieser BTA angezeigt. Nach Auswahl einer BTA-Klasse kann eine Instanz dieser BTA-Klasse ausgewählt werden (*Transaction(s) by Instance*). Zu jeder Instanz wird die gemessene Antwortzeit sowie der Status (erfolgreich oder nicht erfolgreich) dieser Instanz angegeben.

Die ausgewählte Instanz der BTA wird dann im rechten Teil des Fensters grafisch aufbereitet dargestellt. Die BTA sowie sämtliche Subtransaktionen werden durch Rechtecke symbolisiert. Diese enthalten Informationen über den ausführenden Baustein sowie den Namen der ausgeführten Transaktion. Synchron aufgerufene Subtransaktionen werden mit Hilfe einer vertikalen Linie mit der aufrufenden Transaktion verbunden, während horizontale Linien den asynchronen Anstoß einer Transaktion darstellen. Mit Hilfe unterschiedlicher Farben wird der Status der jeweiligen Transaktion (erfolgreich, nicht erfolgreich, noch nicht abgeschlossen, unbekannt) veranschaulicht. Klickt man eine Transaktion an, so erhält man im linken unteren Teil des Fensters detailliertere Informationen zu dieser Transaktion wie z.B. die ausführende Instanz des Bausteins oder zusätzliche, mit Hilfe von `logInfo` übergebene Informationen.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE Messung SYSTEM "Messung.dtd">
<bta>
   <userName>hauck</userName>
   <btaName>SortTA</btaName>
   <time>9816253648233</time>
   <duration>12735</duration>
   <status>Success</status>
   <completed>true</completed>
   <info>Sortierung erfolgreich beendet</info>
   <syncTA>
      <taInfo>
         <componentClass>StartButton</componentClass>
         <componentInstanceID>-489173256</componentInstanceID>
         <taName>BI fuer sortTA</taName>
         <duration>8757</duration>
         <status>Success</status>
         <completed>true</completed>
         <info>Thread Rückkehr zur Oberfläche</info>
      </taInfo>
      <syncTA>
         <taInfo>
            <componentClass>AlternativBean</componentClass>

      ...

         </taInfo>
      </syncTA>
   </syncTA>
</bta>
```


Abbildung 6.8: Ausschnitt aus einer XML-Beschreibung der ermittelten Managementinformation

6.3 Beispielanwendungen

Die Funktionsweise der prototypischen Implementierung soll abschließend anhand zweier einfacher Beispielanwendungen demonstriert werden.

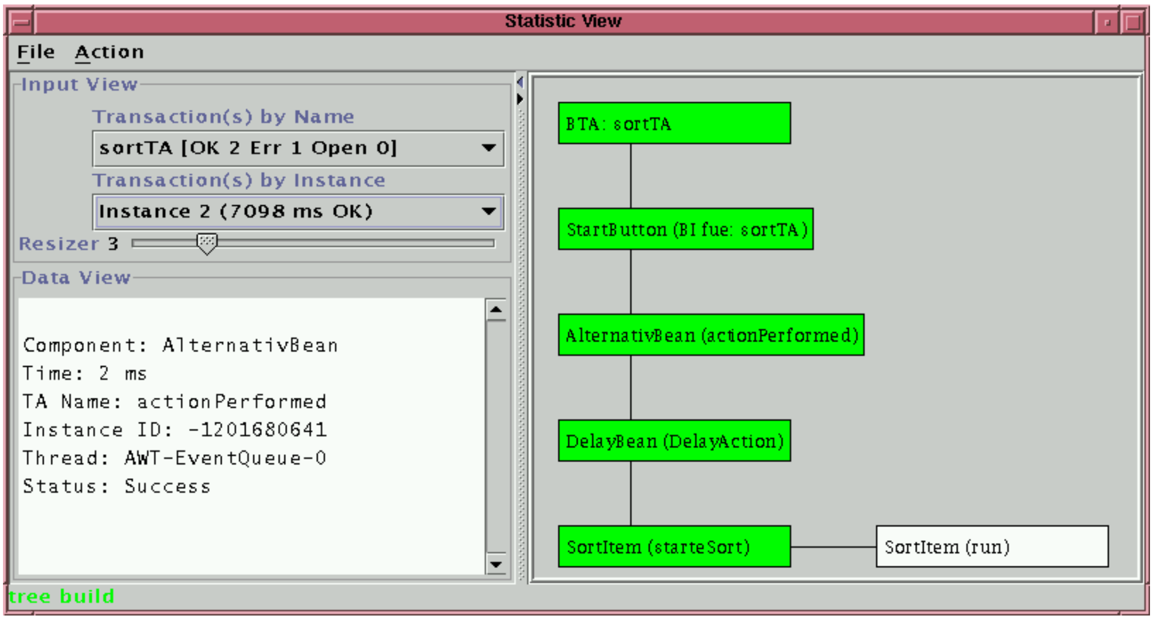

Abbildung 6.9: Prototypische Managementanwendung zur Visualisierung der ermittelten Information

6.3.1 Parallele Ausführung unterschiedlicher BTAs

Zur Demonstration der parallelen Ausführung unterschiedlicher BTAs wurde eine einfache Beispielanwendung erstellt, die zwei BTAs anbietet, die jeweils eine gegebene Menge von Zahlen sortieren und dies grafisch veranschaulichen. Die eine der beiden BTAs bedient sich hierfür des *BubbleSort*-Algorithmus (`BubbleSortBTA`), während die zweite BTA den *QuickSort*-Algorithmus verwendet (`QuickSortBTA`). Um die gemeinsame Verwendung eines Bausteins in unterschiedlichen BTAs zu demonstrieren, wurde die sogenannte `DelayBean` eingesetzt. Diese wird bei Anstoß jeder der beiden BTAs aufgerufen und simuliert beliebige Verarbeitung durch eine konfigurierbare Verzögerung. Der eigentliche Sortiervorgang wird erst nach Abschluß dieser Verzögerung gestartet.

Abbildung 6.10 stellt die *Beanbox* nach Erstellung einer entsprechenden Anwendung dar. Man erkennt zwei *Buttons* (`Starte BubbleSort` und `Starte QuickSort`), mit denen die jeweilige BTA angestoßen werden kann. Weiterhin ist eine Instanz der `DelayBean` (mit einer konfigurierten Verzögerung von sechs Sekunden) sowie zwei *Beans* für die Veranschaulichung der Sortiervorgänge, sogenannte `SortItems` zu erkennen.

Es wurde ein Testlauf durchgeführt, bei dem zunächst eine `BubbleSortBTA` angestoßen wurde. Diese führt – wie konfiguriert – zunächst eine Verzögerung von sechs Sekunden aus. Anschließend wird der Sortiervorgang gestartet, der in einem separaten Kontrollfluß ausgeführt wird. Nach Anstoß des Sortiervorgangs kehrt der ursprüngliche Kontrollfluß somit zur Oberfläche zurück und es können weitere BTAs gestartet werden. Unmittelbar

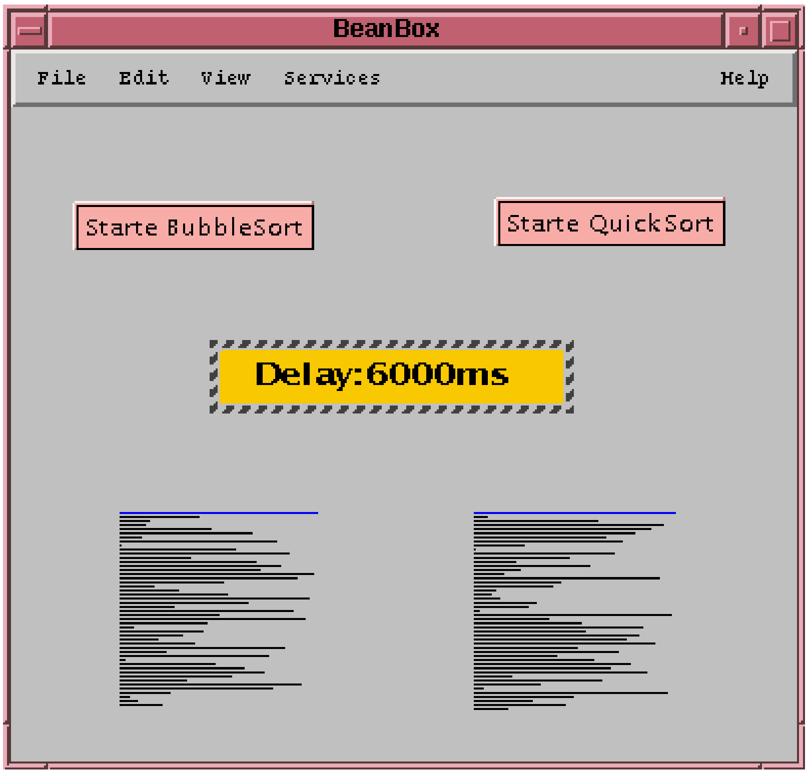

Abbildung 6.10: Einfache Beispielanwendung in der *Beanbox*

nach Rückkehr des Kontrollflusses zur Oberfläche (also während die eigentliche Sortierung noch läuft) wird eine BTA vom Typ `QuickSortBTA` angestoßen, die somit parallel zur ersten BTA ausgeführt wird.

Die Meßergebnisse des Testlaufes wurden an die prototypische Managementanwendung übermittelt. Auf der rechten Seite des Fensters in Abbildung 6.11 erkennt man, daß die `BubbleSortBTA` zunächst mit einer BI innerhalb des `StartButtons` beginnt, der daraufhin synchron zunächst die `DelayBean` und dann die Methode `starteSort` der *Bean* `SortItem` aufruft (zu erkennen an den horizontal leicht versetzten, vertikalen Linien zu `DelayBean` bzw. `SortItem`, die die serielle Ausführung der beiden Transaktionen durch den `StartButton` veranschaulichen). Der Aufruf von `SortItem` führt zu einem Start eines neuen Kontrollflusses, der wiederum die *Bean* `SortItem` ausführt (diesmal aber deren `run`-Methode). Ein ähnliches Bild ergibt sich für die in Abbildung 6.12 dargestellte Visualisierung der `QuickSortBTA`.

Ein wesentlicher Unterschied, der sich zwischen den beiden BTAs ergibt, ist die gemessene Antwortzeit. Wie zu erwarten war, benötigte die `QuickSortBTA` erheblich weniger Zeit (15049ms) als die `BubbleSortBTA` (36046ms). Dies ist im linken oberen Teil der Abbildungen zu erkennen. Durch Selektieren der Subtransaktion, die die eigentliche Sortierung durchführt, könnte dieser Unterschied noch wesentlich detaillierter bestimmt werden (im Beispiel 9029ms vs. 30029ms, in den Abbildungen nicht dargestellt).

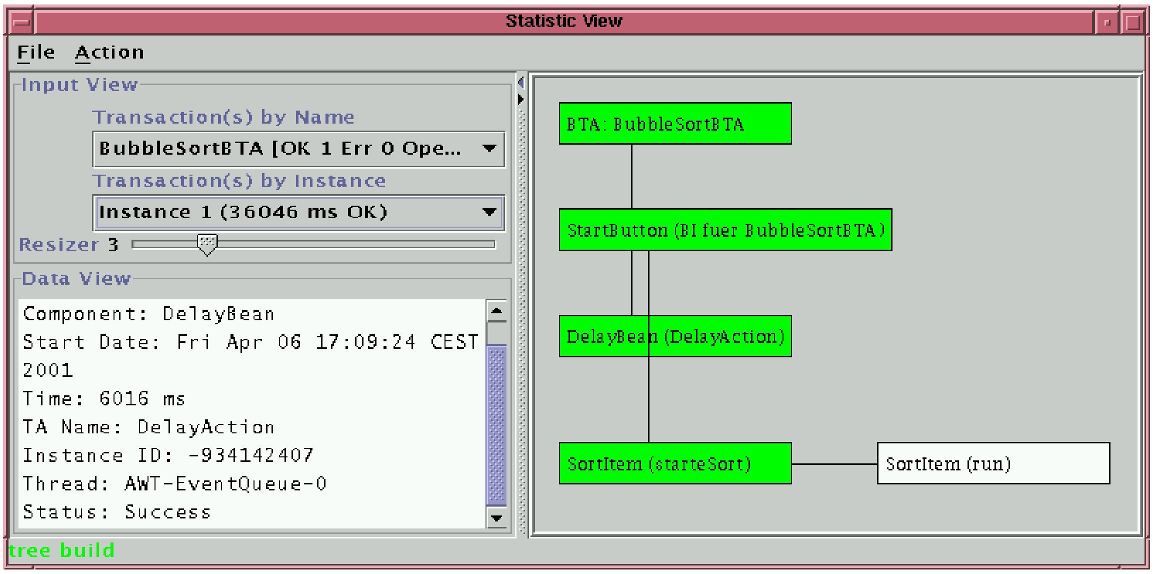

Abbildung 6.11: Visualisierung einer Instanz der `BubbleSortBTA`

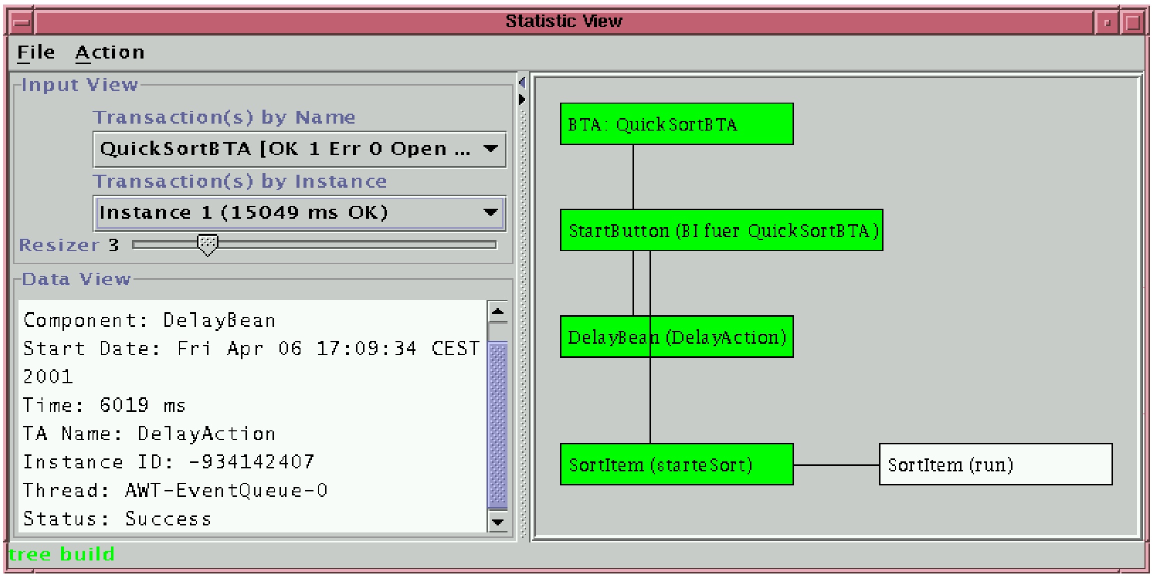

Abbildung 6.12: Visualisierung einer Instanz der `QuickSortBTA`

Weiterhin ist jeweils im linken unteren Bereich der Fenster die Detailinformation zur von der `DelayBean` erbrachten Subtransaktion angegeben. Die Subtransaktion dauerte in beiden Fällen (wie konfiguriert) annähernd sechs Sekunden. Man erkennt (an der identischen `Instance ID`), daß tatsächlich bei beiden BTAs dieselbe Instanz der `DelayBean` zum Einsatz kam und daß beide Male die `DelayBean` im *Thread* der Oberfläche (`AWT-EventQueue-0`) ausgeführt wurde. Dennoch war eine korrekte Zuordnung zur jeweiligen BTA durch die dynamische Zuordnung von Kontrollflüssen zu BTAs problemlos und automatisch möglich. Dies wäre bei statischer Beschreibung der Abhängigkeiten zwischen den einzelnen Bausteinen nicht möglich gewesen.

Die Abläufe während der Ausführung dieser Beispielanwendung werden durch das Sequenzdiagramm in Abbildung 6.14 nochmals verdeutlicht. Aufgrund des Umfangs des Diagramms mußte auf eine Darstellung der Aufrufe der `DelayBean` verzichtet werden. Die grundsätzliche Funktionsweise ist aber dennoch zu erkennen. Zunächst wird der *Button* zum Start des *BubbleSort* (*Button1*) im *Thread* der Oberfläche ausgeführt. `Button1` meldet den Start einer BTA an das Meßobjekt. Das Meßobjekt erzeugt einen eindeutigen Identifikator für die Instanz der BTA (im Beispiel `Bubble1`) und speichert diesen gemeinsam mit der Information über den ausführenden *Thread* (die Liste mit der Zuordnung von BTA-Instanzen zu *Threads* ist jeweils am linken Rand des Sequenzdiagramms veranschaulicht). Nach Rückkehr zum *Button* ruft dieser die Anwendungslogik auf, die ihrerseits die eigentliche *BubbleSort-Bean* aufruft. Vor Aufruf dieser *Bean* wird allerdings wiederum das Meßobjekt vom Start der neuen Subtransaktion verständigt. Anhand des *Threads*, in dem diese Subtransaktion gestartet wird, ist es dem Meßobjekt möglich, die Zuordnung dieser Subtransaktion zur BTA-Instanz `Bubble1` vorzunehmen. Die *BubbleSortBean* erzeugt einen neuen *Thread* durch Aufruf einer Bibliotheksfunktion der *Java Virtual Machine*. Diese informiert das Meßobjekt über die Hinzunahme des weiteren *Threads*. Das Meßobjekt hat somit Kenntnis über die beiden *Threads*, die zu diesem Zeitpunkt die BTA-Instanz `Bubble1` ausführen. Der Start des neuen *Threads* wird vom Meßobjekt gleichzeitig als der Start einer neuen Subtransaktion interpretiert (in der Abbildung nicht dargestellt). Während der ursüprüngliche *Thread* über die *BubbleSortBean* zur Anwendungslogik zurückkehrt, wird im neu generierten *Thread* nunmehr die eigentliche Sortierung ausgeführt. Die Anwendungslogik meldet die Rückkehr des *Threads* als das Ende einer Subtransaktion, die wiederum problemlos der BTA-Instanz `Bubble1` zugeordnet werden kann (Die Zuordnung zum zugehörigen `startTA` erfolgt anhand der Reihenfolge der Aufrufe innerhalb eines *Threads*). Nach Rückkehr zum `Button1` meldet dieser das Verlassen des Kontrollflusses aus der Bearbeitung dieser Instanz einer BTA. Das Meßobjekt löst folglich die Zuordnung des Oberflächen-*Threads* zur Instanz `Bubble1`.

Im Anschluß daran folgt ein analoger Ablauf zum Anstoß des zweiten Sortiervorgangs mit Hilfe des *QuickSort*-Algorithmus. Man erkennt an der Tabelle für die Zuordnung, daß weiterhin eine Zuordnung von Instanz `Bubble1` zu dem *Thread* existiert, der den *BubbleSort*-Algorithmus ausführt. Darüberhinaus enthält die Tabelle nun aber auch einen Eintrag für die zweite BTA-Instanz `Quick1`. Zunächst existiert eine Zuordnung der BTA-Instanz `Quick1` zum *Thread* der Oberfläche. Nach Hinzunahme eines weiteren *Threads* für den eigentlichen Sortiervorgang erscheint auch dieser in der Tabelle. Der Oberflächen-*Thread* verläßt daraufhin die Bearbeitung und die beiden BTA-Instanzen werden weiter in jeweils einem *Thread* ausgeführt.

Endet nun einer dieser beiden *Threads* (im Beispiel zunächst der Thread der BTA-Instanz `Quick1`), so meldet die JVM dies an das Meßobjekt, das die Zuordnung zur BTA-Instanz auflöst und dies gleichzeitig als Ende des Sortiervorgangs erkennt. Analog erfolgt die Auflösung der Zuordnung bei Ende des verbliebenen *Threads*. Nach Abschluß dieses *Threads* verbleibt keine aktive BTA-Instanz mehr in der Tabelle.

6.3.2 Aktive Beans

Die zweite Beispielanwendung dient der Demonstration der Verwendung aktiver *Beans*. Die Anwendung besteht lediglich aus einem *Button* zum Anstoß einer BTA (`ActiveBTA`) und der `ActiveBean`. Diese speichert den Auftrag in einer Warteschlange und überprüft in regelmäßigen Abständen, ob Aufträge zur Bearbeitung vorliegen. Abbildung 6.13 zeigt die Visualisierung einer Instanz der `ActiveBTA` mit Hilfe der erstellten prototypischen Managementanwendung. Man erkennt, daß trotz der vollständigen Entkoppelung der beteiligten Kontrollflüsse durch die Instrumentierung der `ActiveBean` eine Zuordnung möglich ist. Links unten im Fenster ist darüber hinaus zu erkennen, wie vom Bausteinentwickler in die `ActiveBean` eingefügte `doLog`-Aufrufe weitere wertvolle Informationen beispielsweise über die Nummer des aktuell ausgeführten Auftrags übermitteln können.

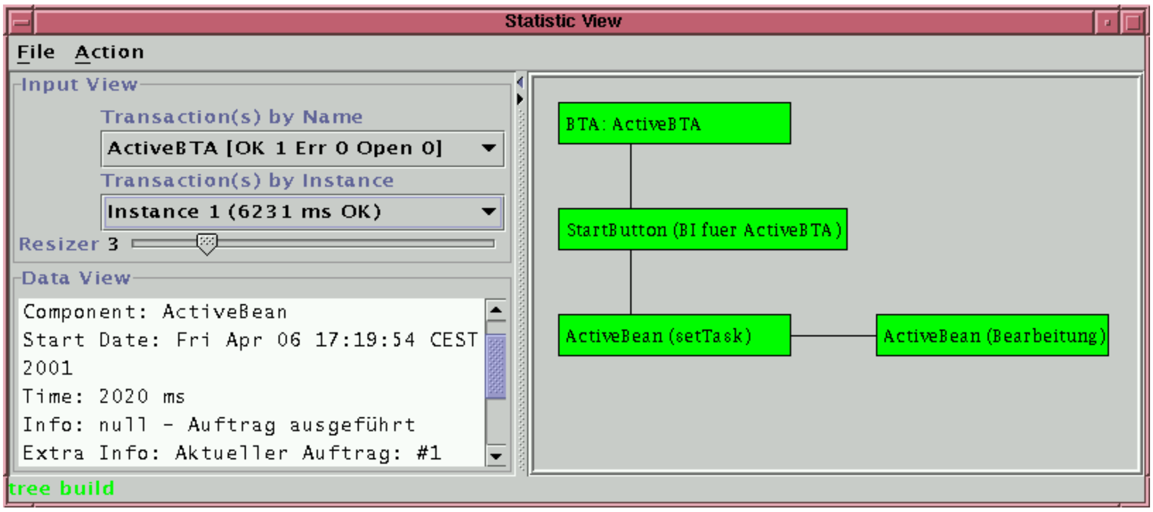

Abbildung 6.13: Visualisierung einer Instanz der `ActiveBTA`

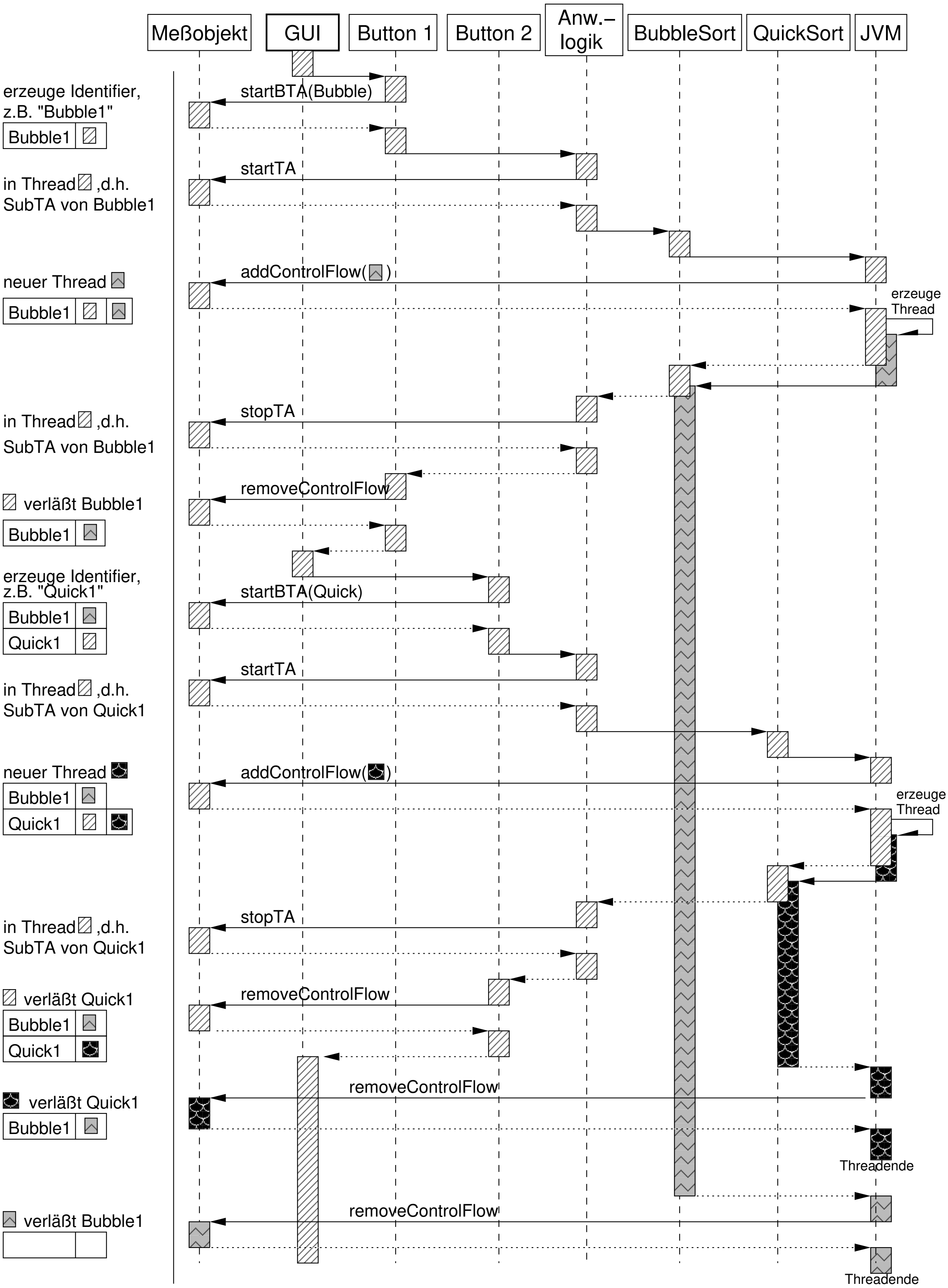

Abbildung 6.14: Abläufe in der Beispielanwendung

Kapitel 7

Ausblick

7.1 Zusammenfassung und wesentliche Ergebnisse der Arbeit

In der vorliegenden Arbeit wurde ausgehend von der Beobachtung, daß herkömmliche Verfahren für die Anwendungsüberwachung nicht in der Lage sind, die geforderten Parameter mit vertretbarem Aufwand zu liefern, eine Architektur vorgestellt, die den Aufwand für die Managementinstrumentierung bausteinbasierter Anwendungen erheblich reduzieren kann.

Im Rahmen einer ausführlichen Anforderungsanalyse wurde gezeigt, daß es für die Überwachung von Anwendungsdiensten erforderlich ist, die Dienstgüte messen zu können, die tatsächlich einem Benutzer bei Inanspruchnahme eines Dienstes zur Verfügung steht. Es wurde gezeigt, daß für den Benutzer die Zeitdauer für die Bearbeitung einer Transaktion sowie die korrekte Bearbeitung von Transaktionen diejenigen Parameter sind, denen derzeit die herausragendste Bedeutung zukommt. Für den Anbieter des Dienstes sind darüber hinaus weitaus detailliertere Messungen erforderlich, um im Falle der Auftretens eines Fehlers die Ursache schnell und einfach eingrenzen und beheben zu können. Die wesentlichste nicht-funktionale Anforderung ist die Minimierung des durch die Überwachung entstehenden zusätzlichen Aufwands für alle Beteiligten, also insbesondere des zusätzlichen Entwicklungsaufwands, da andernfalls nicht damit gerechnet werden kann, daß entsprechende Vorkehrungen für die Dienstgüteüberwachung auch getroffen werden.

Um die Zeitdauer für die Bearbeitung einer Transaktion messen zu können, war zunächst eine eindeutige Definition von Start- und Endpunkt einer Transaktion erforderlich. Hierbei zeigte sich, daß zwischen der *Antwortzeit*, also der Zeit zwischen dem Start einer Transaktion bis zum Zeitpunkt der Ergebnispräsentation, und der *Transaktionsdauer*, also der Gesamtzeit für die vollständige Bearbeitung einer Transaktion, unterschieden werden muß. Es zeigte sich, daß für eine Transaktion beide Parameter gemessen werden können und je nach Anwendungsbereich auch von Managementanwendungen benötigt werden.

Da es ein Ziel der vorliegenden Arbeit war, insbesondere die Überwachung bausteinbasierter Anwendungen zu vereinfachen, wurden die Anforderungen, die über die Anforderungen herkömmlicher Anwendungen hinausgehen, im Anschluß daran explizit untersucht. Es zeigte sich, daß ein einzelner Aufruf eines Bausteins gerade den geeigneten Detaillierungsgrad darstellt, der vom Betreiber eines Anwendungsdienstes gewünscht wird. Aufgrund der Anforderung nach Aufwandsminimierung – sowohl von Seiten des Anwendungsentwicklers als auch von Seiten der Bausteinentwickler – sollten nur ohnehin bei der Erstellung der Anwendung zum Einsatz kommende Techniken und Werkzeuge verwendet werden und muß ein hoher Grad an Automation erreicht werden.

Im folgenden wurden die existierenden Ansätze für die Anwendungsüberwachung den ermittelten Anforderungen gegenübergestellt. Um eine einfache Bewertung existierender und zukünftiger Ansätze zur Anwendungsüberwachung zu gestatten, wurde eine Klassifikation der existierenden Lösungen vorgenommen. Die aktuellen Ansätze von Standardisierungsgremien, aus dem Bereich der Forschung sowie von unterschiedlichen Herstellern von Managementwerkzeugen konnten dann den ermittelten Klassen zugeordnet und so bewertet werden. Hierbei zeigte sich, daß ein Großteil der Lösungen nicht in der Lage ist, die geforderte Information zur Verfügung zu stellen. Dies trifft insbesondere auf die Überwachung des Netzverkehrs sowie auf die Überwachung von Systemparametern zu, die dennoch aktuell den höchsten Verbreitungsgrad aufweisen. Auch die *Client*-seitige Anwendungsüberwachung liefert nur bedingt aussagekräftige Parameter, da keinerlei Detailinformation über mögliche Fehlerursachen ermittelt werden kann.

Einzig die Überwachung der Gesamtanwendung und hier insbesondere die Anwendungsinstrumentierung ermöglicht es, die tatsächliche Dienstnutzung durch aussagekräftige Parameter sowie hinreichende Detailinformation für den Dienstanbieter zu ermitteln. Dennoch hat die Instrumentierung von Anwendungen aktuell eine sehr geringe Verbreitung, was im wesentlichen auf den erheblichen Aufwand für den Anwendungsentwickler zurückzuführen ist, der mit der Instrumentierung einer Anwendung verbunden ist. Insbesondere im Falle bausteinorientierter Anwendungen ist eine Anwendungsinstrumentierung in vielen Fällen mit den heute zur Verfügung stehenden Mitteln vollständig ausgeschlossen, da eine Propagierung von Transaktionsidentifikatoren über Bausteingrenzen hinweg eine Anpassung aller zum Einsatz kommenden Bausteine erfordern würde.

Ziel der vorliegenden Arbeit war es somit, den Aufwand für die Managementintrumentierung von Anwendungen soweit zu verringern, daß er kein Hindernis mehr für eine Verbreitung dieser Technik der Anwendungsüberwachung darstellt. Idee war es hierbei, die Bausteinstruktur heutiger und zukünftiger Anwendungen auszunutzen und eine – zumindest teilweise – Automation der Instrumentierung zu erreichen.

Der zunächst verfolgte Ansatz basierte auf der *Komposition der Managamentschnittstellen instrumentierter Bausteine*. Dieser Ansatz erforderte die Managementinstrumentierung sämtlicher zum Einsatz kommender Bausteine. Aufgrund der genauen Kenntnis der Verknüpfung einzelner Bausteine zu einer Gesamtanwendung zum Zeitpunkt der Anwendungserstellung ist es dann möglich, aus den Managementinformationen der einzelnen Bausteine auf die entsprechende Managementinformation der Gesamtanwendung zu schließen bzw. diese daraus zu berechnen. Es sollte also eine „Managementlogik“ erstellt werden, die die Verknüpfung der Managementinformationen der einzelnen Bausteine beschreibt und an einer Managementschnittstelle zur Verfügung stellt. Es zeigte sich allerdings, daß dieser Ansatz z.B. hinsichtlich des immer noch hohen Aufwands durch die Instrumentierung aller Bausteine oder der erheblicher Schwierigkeiten bei der Zuordnung der Teilinformation (z.B. im Falle mehrfach genutzter Bausteine oder dynamischer Abhängigkeiten innerhalb der Anwendung) nicht geeignet ist, die gestellten Anforderungen zu erfüllen.

Aus diesem Grund wurde ein zweiter Ansatz, die *Automation der Managementinstrumentierung bausteinbasierter Anwendungen* eingeführt. Die Idee dieses Ansatzes basiert auf dem automatischen Einfügen von Meßpunkten durch die Entwicklungsumgebung an den Verknüpfungsstellen zwischen Bausteinen. Somit wird es möglich, die Laufzeit sowie die erfolgreiche bzw. nicht erfolgreiche Bearbeitung jedes Bausteins ohne Eingreifen des Entwicklers zu überwachen. Anhand der Kontrollflüsse, in denen die einzelnen Transaktionen einer Anwendung ausgeführt werden, ist es möglich, vollständig automatisch die ermittelten Meßwerte einer beobachtbaren Transaktion zuzuordnen sowie deren Reihenfolge zu rekonstruieren. Leider erwies sich dieser Ansatz an einigen wenigen Stellen aber ebenfalls als nicht ausreichend, um die vollständige Überwachung zu gestatten. Ein Problem stellte insbesondere die Identifikation des genauen Beginns und Endes einer vom Benutzer angestoßenen Transaktion sowie die eindeutige Zuordnung von Meßwerten im Falle sogenannter aktiver Bausteine dar.

Die letztlich vorgeschlagene Architektur basierte daher auf einer Kombination beider Ansätze, wobei der Ansatz der *Automation der Managemeninstrumentierung bausteinbasierter Anwendungen* überwiegend verwendet wurde und nur an einigen wenigen Stellen auf eine Instrumentierung von Bausteinen wie im Ansatz der *Komposition von Managementschnittstellen instrumentierter Bausteine* beschrieben zurückgegriffen werden mußte. Es wurden sowohl die erforderlichen Schnittstellen als auch Methodiken für Anwendungs– und Bausteinentwickler angegeben, die eine problemlose Instrumentierung von Anwendungen und Bausteinen gestatten. Anschließend wurde anhand eines kurzen Beispiels eine mögliche Anwendung der gewonnenen Information für die Bestimmung und Schätzung der Verfügbarkeit von Anwendungsdiensten differenziert nach Anwendung, gewünschter Transaktion und Benutzer dargestellt.

Um die Umsetzbarkeit der vorgeschlagenen Architektur zu demonstrieren, wurde im Rahmen einer prototypischen Implementierung ein Entwicklungswerkzeug für *JavaBeans*, die sogenannte *Beanbox* dahingehend erweitert, daß beim Erstellen von Anwendungen automatisch Management-*Code* an den Verknüpfungsstellen zweier *JavaBeans* eingefügt wird. Die – nur in wenigen Ausnahmefällen erforderliche – Instrumentierung von *JavaBeans* wurde anhand einiger Beispiele demonstriert. Aus den instrumentierten sowie einer großen Anzahl von unveränderten *JavaBeans* wurden dann einfache Beispielanwendungen erstellt. Die durch die automatische Instrumentierung gewonnenen Meßwerte einiger Testläufe wurden mit Hilfe einer prototypischen Managementanwendung visualisiert.

Die vorliegende Arbeit konnte somit eine Architektur angegeben, die den Aufwand für die Managementinstrumentierung bausteinbasierter Anwendungen erheblich verringert. Damit konnte einer der wesentlichen Hinderungsgründe für eine zügige Verbreitung von Instrumentierungstechniken zur Anwendungsüberwachung beseitigt werden. Die Anzahl an verfügbaren instrumentierten Anwendungen könnte durch Verwendung der vorgeschlagenen Architektur zukünftig also wesentlich erhöht werden.

7.2 Zukünftige Forschungsfragestellungen

Die Arbeit verfolgte das Ziel der Automation der Mangementinstrumentierung bausteinbasierter Anwendungen. Viele der hierbei erzielten Ergebnisse sind allerdings keinesfalls auf das Umfeld der bausteinorientierten Anwendungsentwicklung beschränkt, sondern können auch in anderen Bereichen der Sofwareentwicklung zum Einsatz kommen. Vor allem ist hier das Prinzip der Korrelation von Meßwerten anhand der ausführenden *Threads* zu nennen. In weiterführenden Arbeiten wäre zu untersuchen, inwieweit eine entsprechende Korrelation auch im Falle herkömmlicher, nicht-bausteinorientierter Anwendungsentwicklung zum Einsatz kommen könnte. Dies würde zwar die Aufgabe der Positionierung der Meßstellen im Anwendungs-*Code* weiterhin dem Anwendungsentwickler überlassen, dieser könnte sich aber auf das bloße Einfügen von Meßpunkten beschränken und müßte sich keinerlei Gedanken über die Propagierung von Identifikatoren (z.B. für die Korrelation von Transaktionen) innerhalb der Anwendung – ggf. sogar über Systemgrenzen hinweg – machen.

Die mit Rücksicht auf den Umfang der Arbeit getroffene Beschränkung auf Bausteinarchitekturen, bei denen die Verknüpfung zwischen Bausteinen durch eine explizite Anwendungslogik in Form von generierten Adaptern erfolgt, könnte im Rahmen zukünftiger Arbeiten aufgehoben werden. Insbesondere Architekturen, die eine sogenannte *Interception* durchführen, also jeden Aufruf eines Bausteins zunächst „abfangen“ und dann an den eigentlichen Zielbaustein weiterleiten, kommen ebenso für eine automatisierte Überwachung in Frage. Hier gilt es zu untersuchen, inwieweit durch Erweiterung des die Aufrufe abfangenden *Containers* Meßwerte automatisch ermittelt und weitergeleitet werden können.

Die vorliegende Arbeit beschränkte sich im wesentlichen auf die Mechanismen zur Ermittlung der Managementinformation aus zu überwachenden Anwendungen. Aufbauend auf den gewonnenen Erkenntnissen wäre es nun erforderlich, geeignete Managementanwendungen zu untersuchen und zu entwickeln, die die Informationen geeignet verarbeiten und veranschaulichen können. Ein Beispiel für eine mögliche Managementanwendung wurde in Abschnitt 5.6 gegeben. Die dort vorgeschlagene Anwendung erlaubt die Bestimmung der Verfügbarkeit von Anwendungen und deren Transaktionen ausgehend von Information über Ausfälle einzelner Bausteine. Durch eine systematische Untersuchung könnte hier eine Vielzahl weitere Anwendungen gefunden werden.

Ziel dieser Arbeit war es nicht, Dienstgüteparameter für Anwendungsdienste zu bestimmen und zu formalisieren, sondern einen Mechanismus bzw. Methodiken anzugeben, wie die derzeit bedeutendsten Dienstgüteparameter, die Antwortzeit bzw. Transaktionsdauer sowie der Erfolg bzw. Mißerfolg einer Transaktion mit möglichst geringem Aufwand gemessen werden können. Selbstverständlich lassen sich nicht alle Kundenanforderungen vollständig auf diese beiden Parameter reduzieren. In zukünftigen Arbeiten ist es deshalb erforderlich, weitere bedeutsame Parameter zu identifizieren und soweit zu formalisieren, daß sie in Dienstvereinbarungen festgelegt werden können. Schmidt [Schm 00] befaßt sich in seiner Arbeit mit der Formalisierung entsprechender Dienstvereinbarungen. Im Anschluß daran müssen dann, vergleichbar mit der vorliegenden Arbeit, Mechanismen entwickelt werden, wie die festgelegten Parameter einfach und effizient überwacht werden können.

Wie beschrieben ermitteln die heutzutage vorherrschenden Verfahren zur Anwendungsüberwachung ihre Informationen durch Überwachung des Netzverkehrs oder von Systemparametern. Dies liegt im wesentlichen an der großen Erfahrung und damit verbundenen Ausgereiftheit der eingesetzten Verfahren. Auch wenn diese Verfahren nicht die gewünschten benutzerorientierten Informationen liefern können, so können sie dennoch einem erfahrenen Administrator auf Seiten des Dienstanbieters wertvolle Hinweise z.B. im Falle eines Dienstausfalls auf die Ursache geben. Um eine optimale Verwendung dieser Informationen zu erreichen, wäre es erforderlich, die Korrelation dieser Parameter mit den in der vorliegenden Arbeit ermittelten benutzerorientierten Parametern zu erreichen. So wäre es beispielsweise sinnvoll, die CPU-Nutzung oder den Speicherbedarf einer Anwendung den einzelnen Transaktionen und somit Benutzern zuordnen zu können. Auch in Falle bausteinorientierter Anwendungen wäre eine Zuordnung wünschenswert. Hier wäre es z.B. sinnvoll, die einzelnen Systemparameter dem verursachenden Baustein zuordnen zu können, um im Falle überlasteter Systeme Anhaltspunkte gewinnen zu können, welcher der Bausteine ursächlich für die hohe Systemlast ist.

Weiterhin ist eine Verwendung der gewonnenen Information im Bereich des Abrechnungsmanagements anzustreben. In der vorliegenden Arbeit lag der Hauptfokus auf den Bereichen des Fehler- und Leistungsmanagements. Wenn zukünftig eine verursacherbezogene und volumenorientierte Abrechnung von Anwendungsdiensten auf Transaktionsbasis durchgeführt werden soll, sind genau die mit Hilfe der vorgeschlagenen Architektur zu ermittelnden Informationen von Bedeutung, nämlich die Häufigkeit der Nutzung bestimmter Anwendungsdienste bzw. Transaktionen differenziert nach Benutzer, die Antwortzeit sowie die erfolgreiche bzw. nicht erfolgreiche Bearbeitung der Transaktion. Somit können aufbauend auf der vorgeschlagenen Architektur Abrechnungsverfahren entwickelt werden, die eine verursachergerechte Abrechnung abhängig von der erreichten Dienstgüte gestatten.

Durch die Beschränkung auf das Fehler- und Leistungsmanagement konnte im Rahmen dieser Arbeit die sichere und verläßliche Übertragung der gewonnenen Meßwerte an die Managementsysteme außer Acht gelassen werden. Geht man allerdings von einer zukünftigen Verwendung im Abrechnungsmanagement aus, so stellt der Verlust von Meßwerten ein ernstzunehmendes Problem dar. Meßwerte können beispielsweise durch Übertragungsprobleme oder durch Ressourcenmangel in der *Client*-Komponente verloren gehen. Eine für die Abrechnung erforderliche spätere Rekonstruktion der genauen Dienstnutzung und verfügbaren Dienstgüte ist dann nicht mehr gewährleistet. Es sind also geeignete Verfahren zu entwickeln, die einerseits die Wahrscheinlichkeit für den Verlust von Meßwerten minimieren, andererseits aber auch über geeignete Mechanismen verfügen, um im Falle des Verlusts von Meßwerten z.B. mit Hilfe heuristischer Verfahren, eine gute Näherung der tatsächlichen Werte bestimmen zu können.

Abkürzungsverzeichnis

ACAP WG	Application Configuration Access Protocol Working Group
ACAP	Application Configuration Access Protocol
ACT	Application Components Team
ADF	Application Description File
AIC	Application Instrumentation & Control
AL	Application Library
AML	Activity Monitoring Language
AMP	Application Management Package
AMS	Application Management Specification
APM	Application Performance Measurement
ARM	Application Response Measurement
ASP	Application Service Provider
BI	Benutzerinteraktion
BSM	Business System Mapping
BTA	Benutzertransaktion
CAMI	Common Application Management Interface
CCM	CORBA Component Model
CDF	Component Description File
CIM	Common Information Model
CL	Client Library

CMG	Computer Measurement Group
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DAP WG	Distributed Application Performance Working Group
DCE	Distributed Computing Environment
DCOM	Distributed Component Object Model
DLL	Dynamic Link Library
DMI	Distributed Management Interface
DMTF	Distributed Management Task Force
DTD	Document Type Definition
EJB	Enterprise JavaBeans
GAMOCs	Generic Application Managed Object Classes
GDF	Global Description File
GEM	Global Enterprise Manager
HR MIB	Host Resources MIB
IDL	Interface Definition Language
IETF	Internet Engineering Task Force
JMS	Java Messaging Service
MIB	Management Information Base
MIF	Management Information Format
MOF	Managed Object Format
NMF	Network Management Forum
NSM	Network Services Monitoring
ODP	Open Distributed Processing
OSF	Open Software Foundation
RFC	Request for Comment
RMI	Remote Method Invocation

RTFM WG	Realtime Traffic Flow Measurement Working Group
RTFM	Realtime Traffic Flow Management
SNMPv2	Simple Network Management Protocol Version 2
SNMP	Simple Network Management Protocol
TAPM	Tivoli Application Performance Manager
TBSM	Tivoli Business System Manager
TIM	Technology Integration Map
TMB	Tivoli Module Builder
TMD	Tivoli Module Designer
TMForum	TeleManagement Forum
TMF	Tivoli Management Framework
TOM	Telecom Operations Map
UML	Unified Modelling Language
UoW	Unit of Work
XML	Extensible Markup Language

ABBILDUNGEN

1.1 Graphische Darstellung der Vorgehensweise 6

2.1 Verknüpfung von Bausteinen über generierte Adapter 8
2.2 Verknüpfung von *JavaBeans* mit Hilfe der *Beanbox* 11
2.3 Beispiel für eine von der Beanbox generierte Adapterklasse 12
2.4 Mit Hilfe von Visual Basic teilweise automatisch generierte Prozedur 14
2.5 Verknüpfung zweier Bausteine im Client 15
2.6 Interception durch den Container 15
2.7 Darstellung unterschiedlicher Kontrollflüsse in Sequenzdiagrammen 22
2.8 Allgemeine Darstellung einer Benutzertransaktion 25
2.9 Darstellung der Zusammenhänge im Bereich der Transaktionsüberwachung . 27
2.10 Dienstmodell nach [SMTF 01] 28

3.1 Modellierung von Anwendungsdiensten 32
3.2 Modellierung bausteinbasierter Anwendungsdienste 36

4.1 Möglichkeiten der Überwachung verteilter Anwendungen 44
4.2 Zusammenfassung: Gegenüberstellung von Überwachungstechniken und Anforderungen 49
4.3 CIM Distributed Application Performance Schema 2.4 [Dap 24] 59
4.4 ARM API: Architektur 62
4.5 Funktionsweise der AIC API (aus [C910]) 65
4.6 Die *Building Blocks* der AMS und ihre Einordnung im CIM der DMTF . . . 76

5.1 Komposition von Managementschnittstellen mittels Managementlogik 87
5.2 Sequenzdiagramme für synchrone und asynchrone Aufrufe von Bausteinen . 88
5.3 Algorithmus für die Berechnung der Antwortzeit 89
5.4 Algorithmus für die Berechnung der Transaktionsdauer 90

5.5 Algorithmus für die Berechnung der Verfügbarkeit 91
5.6 Strukturgraph einer einfachen Beispielanwendung 91
5.7 Automation der Managementinstrumentierung bausteinbasierter Anwendungen 96
5.8 BTA eines bausteinorientierten Anwendungsdienstes 98
5.9 Identifikation des Meßpunktes für den Beginn einer BTA 99
5.10 Identifikation des Meßpunktes für die Ergebnispräsentation 100
5.11 Identifikation des Meßpunktes für die Rückkehr zur Oberfläche 101
5.12 ARM-Instrumentierung eines Webbrowsers (nach [HaRe 00]) 102
5.13 Identifikation der Meßpunktes für Subtransaktionen 103
5.14 Aufruf innerhalb eines Kontrollflusses 104
5.15 Zuordnung von Subtransaktionen bei Verwendung der ARM API (nach [John 97]) 105
5.16 Problematik der Zuordnung von Subtransaktionen zu BTAs 106
5.17 Korrelation anhand von Kontrollflüssen: Mögliche Probleme 108
5.18 Varianten der Erbringung von BTAs 108
5.19 Start neuer Kontrollflüsse mittels Bibliothek 109
5.20 Vollständige Entkoppelung der Kontrollflüsse im Falle aktiver Bausteine . . . 111
5.21 Instrumentierung aktiver Bausteine 117
5.22 Architektur für die Ermittlung der Managementinformation 119
5.23 Kommunikationsschnittstellen der Architektur 121
5.24 Spezifikation der Meßschnittstelle 122
5.25 DTD der an der Managementschnittstelle übergebenen Information 127
5.26 Beispiel für einen automatisch instrumentierten Adapter 130
5.27 Beispiel für die Instrumentierung eines Eingabebausteins 132
5.28 Beispiel für die Instrumentierung eines aktiven Bausteins 134
5.29 Antwortzeit einer Subtransaktion mit bzw. ohne Instrumentierung 139
5.30 Slowdown durch Instrumentierung 140
5.31 Generierung und Beendigung von Threads 141

6.1 Implementierung der Architektur für die Automation der Managementinstrumentierung 144
6.2 Erweiterung des Entwicklungswerzeugs Beanbox 145
6.3 Von erweiterter Beanbox generierte Adapterklasse 146
6.4 Instrumentierung eines StartButtons 147
6.5 Implementierung der Architektur für die Ermittlung der Managementinformation 148
6.6 Java-Implementierung der Meßschnittstelle 149
6.7 Java-Implementierung des Meßobjekts (Ausschnitt) 150

6.8 Ausschnitt aus einer XML-Beschreibung der ermittelten Managementinformation 153
6.9 Prototypische Managementanwendung zur Visualisierung der ermittelten Information 154
6.10 Einfache Beispielanwendung in der *Beanbox* 155
6.11 Visualisierung einer Instanz der `BubbleSortBTA` 156
6.12 Visualisierung einer Instanz der `QuickSortBTA` 156
6.13 Visualisierung einer Instanz der `ActiveBTA` 158
6.14 Abläufe in der Beispielanwendung 159

LITERATUR

[ActX 00] MICROSOFT CORPORATION: *ActiveX Controls*, 2000, http://www.microsoft.com/com/tech/activex.asp.

[AdWi 96] ADAMS, E.K. und K.J. WILLETTS: *The Lean Communications Provider: Surviving the Stakeout through Service Management Excellence*. McGraw–Hill, 1996.

[Alde 00] AL-DEBES, FARID: *AMS gestützte Anwendungsüberwachung mit Tivoli TME 10 bei der BMW AG*. Diplomarbeit, Technische Universität München, November 2000.

[AMS 2] *Application Management Specification*. Version 2.0, Tivoli Systems, 1997.

[Appl 24] DMTF APPLICATION WORKING GROUP: *Understanding the Application Management Model*. Specification Version 1.0, Distributed Management Task Force, Mai 1998, http://www.dmtf.org/spec/Whitepapers/CIM_Applications_wp.pdf.

[Appl 98] APPLEMAN, D.: *Developing COM/ActiveX Components with Visual Basic 6.0*. Sams Publishing, 1998. ISBN 1562765760.

[ApplMOF 25] DMTF APPLICATION WORKING GROUP: *Application MOF Specification 2.5*. CIM Schema CIM_Application25.mof, Distributed Management Task Force, Dezember 2000, http://www.dmtf.org/spec/CIM_Schema25/CIM_Application25.mof.

[BEP+ 94] BROWER, D., EDITOR, B. PURVY, RDBMSMIB WORKING GROUP CHAIR, A. DANIEL, M. SINYKIN und J. SMITH: *RFC 1697: Relational Database Management System (RDBMS) Management Information Base (MIB) using SMIv2*. RFC, IETF, August 1994, ftp://ftp.isi.edu/in-notes/rfc1697.txt.

[BHP+ 00] BROY, M., H.-G. HEGERING, A. PICOT, A. BUTTERMANN, M. GARSCHHAMMER, R. HAUCK und S. VOGEL: *Kommunikations– und Informationstechnik 2010, Trends in Technologie und Markt*. Secu-Media Verlag, ISBN 3-922746-35-7, Ingelheim, September 2000.

[BiHa 97] BIAGGIOLINI, V. und J. HARMS: *Toward Automatic, Run-Time Fault Management for Component-Based Applications*. In: WECK, W., J. BOSCH und C. SZYPERSKI (Herausgeber): *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP '97)*, Nummer 5 in *TUCS General Publication*, Seiten 5–12, Jyväskylä, Finland, September 1997. Turku Centre for Computer Science, ISBN 952-12-0039-1, `http://www.tucs.abo.fi/publications/general/G5.html`.

[BMR 97] BROWNLEE, N., C. MILLS und G. RUTH: *RFC 2063: Traffic Flow Measurement: Architecture*. RFC, IETF, Januar 1997, `ftp://ftp.isi.edu/in-notes/rfc2063.txt`.

[Brad 96] BRADNER, S.: *RFC 2026: The Internet Standards Process – Revision 3*. RFC, IETF, Oktober 1996, `ftp://ftp.isi.edu/in-notes/rfc2026.txt`.

[C807] *Application Response Measurement (ARM) API* . Technical Standard C807, The Open Group, Juli 1998.

[C910] *Application Instrumentation and Control (AIC) API, Version 1.0*. Technical Standard C910, The Open Group, November 1999.

[CFSD 90] CASE, J.D., M. FEDOR, M.L. SCHOFFSTALL und C. DAVIN: *RFC 1157: Simple Network Management Protocol (SNMP)*. RFC, IETF, Mai 1990, `ftp://ftp.isi.edu/in-notes/rfc1157.txt`.

[CIM 2.2] *Common Information Model (CIM) Specification Version 2.2*. Specification, Juni 1999, `http://www.dmtf.org/spec/cims.html/`.

[COM 00] MICROSOFT CORPORATION: *COM*, 2000, `http://www.microsoft.com/com/tech/com.asp`.

[CORBA 2.2] *The Common Object Request Broker: Architecture and Specification*. OMG Specification Revision 2.2, Object Management Group, Februar 1998.

[Core 24] DMTF SYSTEM AND DEVICES WORKING GROUP: *Common Information Model (CIM) Core Model*. White Paper, Distributed Management Task Force, August 2000, `http://www.dmtf.org/spec/Whitepapers/DSP111.pdf`.

[CoreMOF 25] DMTF SYSTEM AND DEVICES WORKING GROUP: *Core MOF Specification 2.5*. CIM Schema CIM_Core25.mof, Distributed Management Task Force, Dezember 2000, `http://www.dmtf.org/spec/CIM_Schema25/CIM_Core25.mof`.

[Dap 24] DMTF DISTRIBUTED APPLICATION PERFORMANCE WORKING GROUP: *Distributed Application Performance*. CIM Schema 2.4, Distributed Management Task Force, Februar 2000, `http://www.dmtf.org/spec/CIM_Schema24/CIM_DAP24.pdf`.

[DapMOF 24] DMTF DISTRIBUTED APPLICATION PERFORMANCE WORKING GROUP: *DAP MOF Specification 2.4*. CIM Schema CIM_Dap24.mof, Distributed Management Task Force, August 2000, `http://www.dmtf.org/spec/CIM_Schema23/CIM_Dap24.mof`.

[Deil 01] DEILER, T.: *Prototyp zur Visualisierung von leistungsbezogenen Messwerten bausteinbasierter Anwendungen*. Systementwicklungsprojekt, Technische Universität München, Mai 2001.

[DeSo 97] DESOTO, A.: *Using the Beans Development Kit*. Tutorial, JavaSoft, September 1997, `http://java.sun.com/beans/docs/Tutorial-Sep97.pdf`.

[DMI 2.0s] *Distributed Management Interface Specification, Version 2.0s*. Specification, Distributed Management Task Force, Juni 1998, `http://www.dmtf.org/spec/spec.html`.

[DMTF 00] DISTRIBUTED MANAGEMENT TASK FORCE, INC.: *The Distributed Management Task Force: Driving industry standards for systems management to reduce total cost of ownership*, 2000, `http://www.dmtf.org/download/pres/overview.pdf`.

[EJB 2.0] *Enterprise JavaBeans Specification Version, Version 2.0 – Proposed Final Draft*. Specification, Sun Microsystems, Oktober 2000, `http://java.sun.com/products/ejb/docs.html`.

[ETE 00] CANDLE CORPORATION: *CandleNet ETEWatch: Measuring Application Performance From the Customer Perspective*, 2000, `http://www.candle.com/solutions_t/enduser_solutions/application_responsetime_internal/etewatch.html`.

[Fisc 01] FISCHER, M.: *Evaluierung von Werkzeugen zur Antwortzeitüberwachung bei der DeTeSystem*. Systementwicklungsprojekt, Technische Universität München, April 2001, `http:`

//www.nm.informatik.uni-muenchen.de/common/Literatur/MNMPub/Fopras/fisc01/fisc01.shtml.

[FJP 99] FROLUND, SVEND, MUDITA JAIN und JIM PRUYNE: *SoLOMon: Monitoring End-ser Service Levels*. In: *Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management (IM'99), Boston, MA*, Mai 1999.

[GCM+ 96a] GROUP, SNMPV2 WORKING, J. CASE, K. MCCLOGHRIE, M. ROSE und S. WALDBUSSER: *RFC 1905: Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*. RFC, IETF, Januar 1996, ftp://ftp.isi.edu/in-notes/rfc1905.txt.

[GCM+ 96b] GROUP, SNMPV2 WORKING, J. CASE, K. MCCLOGHRIE, M. ROSE und S. WALDBUSSER: *RFC 1907: Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2)*. RFC, IETF, Januar 1996, ftp://ftp.isi.edu/in-notes/rfc1907.txt.

[GEM 99] *Instrumenting Enterprise Applications Using Tivoli GEM*. IBM Redbook SG24-5399-00, IBM Tivoli, August 1999, http://www.redbooks.ibm.com/abstracts/sg245399.html.

[Grif 98] GRIFFEL, F.: *Componentware: Konzepte und Techniken eines Softwareparadigmas*. dpunkt–Verlag, 1998.

[GrWa 93] GRILLO, P. und S. WALDBUSSER: *RFC 1514: Host Resources MIB*. RFC, IETF, September 1993, ftp://ftp.isi.edu/in-notes/rfc1514.txt.

[HAN 99] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integrated Management of Networked Systems – Concepts, Architectures and their Operational Application*. Morgan Kaufmann Publishers, ISBN 1-55860-571-1, 1999. 651 p.

[HAN 99a] HEGERING, H.-G., S. ABECK und B. NEUMAIR: *Integriertes Management vernetzter Systeme — Konzepte, Architekturen und deren betrieblicher Einsatz*. dpunkt-Verlag, ISBN 3-932588-16-9, 1999, http://www.dpunkt.de/produkte/management.html. 607 S.

[HaRa 00] HAUCK, R. und I. RADISIC: *Monitoring Application Service Performance – Classification and Analysis of Existing Approaches*. In: *Workshop of the OpenView University Association (OVUA 2000), Santorini, Greece*, Juni 2000.

[HaRe 00] HAUCK, R. und H. REISER: *Monitoring Quality of Service across Organizational Boundaries.* In: LINNHOFF-POPIEN, C. und H.-G. HEGERING (Herausgeber): *Trends in Distributed Systems: Torwards a Universal Service Market. Proceedings of the third International IFIP/GI Working Conference, USM 2000*, Nummer 1890 in *Lecture Notes in Computer Science (LNCS)*, Munich, Germany, September 2000. Springer, `http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Publikationen/hare00/hare00.shtml`.

[HaRe 99] HAUCK, R. und H. REISER: *Monitoring of Service Level Agreements with flexible and extensible Agents.* In: *Workshop of the OpenView University Association (OVUA'99)*, Bologna, Italy, Juni 1999. , `http://www.hpovua.org/PUBLICATIONS/PROCEEDINGS/6_HPOVUAWS/Papers/hauck_reiser.pdf`.

[Hawo 97] HAWORTH, MARTIN: *Service Management using the Application Response Measurement API without Application Source Code Modification.* Technischer Bericht, Computer Measurement Group, Turnersville, NJ 08012, Juni 1997.

[HKS 99] HAZEWINKEL, H., C. KALBFLEISCH und J. SCHOENWAELDER: *RFC 2594: Definitions of Managed Objects for WWW Services.* RFC, IETF, Mai 1999, `ftp://ftp.isi.edu/in-notes/rfc2594.txt`.

[HMMT 99] HELLERSTEIN, J.L., M.M. MACCABEE, W.N. MILLS III und J. TUREK: *ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems.* In: *Proceedings of the 19th International Conference on Distributed Computing Systems*, Seiten 152–162, Austin, TX, USA, Juni 1999. IEEE Computer Society.

[Hojn 99] HOJNACKI, M.: *Einsatz des Java Dynamic Management Kit (JDMK) zur Antwortzeitüberwachung bei der DeTeSystem* . Diplomarbeit, Technische Universität München, Mai 1999, `http://wwwmnmteam.informatik.uni-muenchen.de/common/Literatur/MNMPub/Diplomarbeiten/hojn99/hojn99.shtml`.

[ISO 7498-4] *Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 4: Management Framework.* IS 7498-4, International Organization for Standardization and International Electrotechnical Committee, 1989.

[ISO 9596-1] *Information Technology – Open Systems Interconnection – Common Management Information Protocol – Part 1: Specification*. IS 9596-1, International Organization for Standardization and International Electrotechnical Committee, 1991.

[J2CS 01] SUN MICROSYSTEMS, INC.: *Welcome to the Java Community Source Developers' Area*, 2001, `http://developer.java.sun.com/developer/products/java2cs/index.html`.

[JavaBeans 1.01] *JavaBeans*. Specification, Sun Microsystems, Juli 1997, `http://java.sun.com/beans/docs/beans.101.pdf`.

[John 99] JOHNSON, M. W.: *ARM 3.0 — Enabling Wider Use of ARM in the Enterprise*. In: *Computer Measurement Group's 1999 International Conference (CMG99)*, Reno, Nevada, USA, December, 5–10 1999. . `http://www.cmg.org/regions/cmgarmw/arm30enhancements.pdf`.

[John 97] JOHNSON, MARK: *The Application Response Measurement (ARM) API, Version 2*. Technischer Bericht, Computer Measurement Group, Turnersville, NJ 08012, November 1997.

[JVMPI 99] SUN MICROSYSTEMS, INC.: *Java Virtual Machine Profiler Interface (JVMPI)*, 1999, `http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/`.

[Kais 99] KAISER, T.: *Methodik zur Bestimmung der Verfügbarkeit von verteilten anwendungsorientierten Diensten*. Dissertation, Technische Universität München, April 1999.

[Kali 01] KALIX, E.: *Prototyp zur Automation von Performanz-Messungen in JavaBeans-basierten Anwendungen*. Fortgeschrittenenpraktikum, Ludwig–Maximilians–Universität München, April 2001.

[KiFr 94a] KILLE, S. und N. FREED: *RFC 1565: Network Services Monitoring MIB*. RFC, IETF, Januar 1994, `ftp://ftp.isi.edu/in-notes/rfc1565.txt`.

[KiFr 94b] KILLE, S. und N. FREED: *RFC 1566: Mail Monitoring MIB*. RFC, IETF, Januar 1994, `ftp://ftp.isi.edu/in-notes/rfc1566.txt`.

[KKC 00] KAR, G., A. KELLER und S. CALO: *Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis*. In: HONG, J. W. und R. WEIHMAYER (Herausgeber): *NOMS*

2000 IEEE/IFIP Network Operations and Managment Symposium — The Networked Planet: Management Beyond 2000, Seiten 61–74, Honolulu, Hawaii, USA, April 2000. IEEE.

[KKPS 99] KALBFLEISCH, C., C. KRUPCZAK, R. PRESUHN und J. SAPERIA: *RFC 2564: Application Management MIB*. RFC, IETF, Mai 1999, `ftp://ftp.isi.edu/in-notes/rfc2564.txt`.

[KrSa 98] KRUPCZAK, C. und J. SAPERIA: *RFC 2287: Definitions of System-Level Managed Objects for Applications*. RFC, IETF, Februar 1998, `ftp://ftp.isi.edu/in-notes/rfc2287.txt`.

[MaKi 94] MANSFIELD, G. und S. KILLE: *RFC 1567: X.500 Directory Monitoring MIB*. RFC, IETF, Januar 1994, `ftp://ftp.isi.edu/in-notes/rfc1567.txt`.

[Maye 00] MAYER, JOACHIM: *Modellierung von IT-Prozessen mit Tivoli GEM für die Überwachung im Telefonie–/Internet–Brokerage Umfeld der Advance Bank*. Diplomarbeit, Technische Universität München, November 2000.

[McRo 91] MCCLOGHRIE, K. und M.T. ROSE: *RFC 1213: Management Information Base for Network Management of TCP/IP-based internets:MIB-II*. RFC, IETF, März 1991, `ftp://ftp.isi.edu/in-notes/rfc1213.txt`.

[Mons 00] MONSON-HAEFEL, R.: *Enterprise JavaBeans*. The Java Series. O'Reilly, Zweite Auflage, 2000. ISBN: 1-56592-869-5.

[MPS 99] MCCLOGHRIE, K., D. PERKINS und J. SCHOENWAELDER: *RFC 2578: Structure of Management Information Version 2 (SMIv2)*. RFC, IETF, April 1999, `ftp://ftp.isi.edu/in-notes/rfc2578.txt`.

[NeMy 97] NEWMAN, C. und J. G. MYERS: *RFC 2244: ACAP – Application Configuration Access Protocol*. RFC, IETF, November 1997, `ftp://ftp.isi.edu/in-notes/rfc2244.txt`.

[Neum 98] NEUMAIR, B.: *Distributed Applications Management based on ODP Viewpoint Concepts and CORBA*. In: *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS 98)*, Seiten 559–569, New Orleans, USA, Februar 1998. IEEE Communications Society.

[OMG 00-03-01] *Unified Modeling Language (UML) 1.3 specification*. Technischer Bericht formal/00-03-01, März 2000, `ftp://ftp.omg.org/pub/docs/formal/00-03-01.pdf`.

[OMG 00-06-02] *Generic Requirements for Telecommunications Management Building Blocks*. Technischer Bericht orbos/00-06-02, Juni 2000, ftp://ftp.omg.org/pub/docs/orbos/00-06-02.pdf.

[OMG 96-09-03] *The Reference Model of Open distributed Processing (ODP)*. Technischer Bericht ad/96-09-03, September 1996, ftp://ftp.omg.org/pub/docs/ad/96-09-03.pdf.

[OMG 97-06-12] *CORBA Component Model RFP, Final Version*. Technischer Bericht orbos/97-06-12, Juni 1997.

[OMG 99-07-01] *Component Spec - Volume I*. Technischer Bericht orbos/99-07-01, Juli 1999, ftp://ftp.omg.org/pub/docs/orbos/99-07-01.pdf.

[OMG 99-07-02] *Component Spec - Volume II*. Technischer Bericht orbos/99-07-02, Juli 1999, ftp://ftp.omg.org/pub/docs/orbos/99-07-02.pdf.

[OMG 99-07-03] *Component Spec - Volume III*. Technischer Bericht orbos/99-07-03, Juli 1999, ftp://ftp.omg.org/pub/docs/orbos/99-07-03.pdf.

[RoMc 90] ROSE, M.T. und K. MCCLOGHRIE: *RFC 1155: Structure and identification of management information for TCP/IP-based internets*. RFC, IETF, Mai 1990, ftp://ftp.isi.edu/in-notes/rfc1155.txt.

[Schm 00] SCHMIDT, H.: *Service Contracts based on Workflow Modeling*. In: AMBLER, A., S.B. CALO und G. KAR (Herausgeber): *Proceedings of the 11th Annual IFIP/IEEE International Workshop on Distributed Systems: Operations & Management (DSOM 00)*, Nummer 1960 in *Lecture Notes in Computer Science*, Austin, Texas, USA, Dezember 2000. Springer.

[Skla 00] SKLAREVSKIY, V.: *Modellierung von Betriebsabläufen mit Unicenter TNG*. Fortgeschrittenenpraktikum, Ludwig–Maximilians–Universität München, November 2000, http://www.nm.informatik.uni-muenchen.de/common/Literatur/MNMPub/Fopras/skla00/skla00.shtml.

[Smea 99] SMEAD, S.: *ARM 3.0 API*. Technischer Bericht, Computer Measurement Group, April 1999. http://www.cmg.org/regions/cmgarmw/arm30description.pdf.

[SMTF 01] GARSCHHAMMER, M., R. HAUCK, H.-G. HEGERING, B. KEMPTER, M. LANGER, M. NERB, I. RADISIC, H. ROELLE und H. SCHMIDT: *Towards generic Service Management Concepts – A Service Model Based Approach*. In: PAVLOU, G., N. ANEROUSIS und A. LIOTTA (Herausgeber): *Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001)*, Seattle, Washington, USA, Mai 2001. IEEE Publishing, `http://www.nm.informatik.uni-muenchen.de/Literatur/MNMPub/Publikationen/smtf01/smtf01.shtml`.

[StBu 98] STURM, R. und W. BUMPUS: *Foundations of Application Management*. Wiley Computer Publishing, 1998.

[STK 96] SCHADE, A. und P. TROMMLER: *A CORBA-Based Model for Distributed Applications Management*. In: *Proceedings of the IFIP/IEEE Seventh International Workshop on Systems: Operations & Management (DSOM'96), L'Aquila, Italy*, 1996.

[SWMIF 2] *Software Standards Groups Definition*. Version 2.0, Distributed Management Task Force, November 1995.

[Szyp 98] SZYPERSKI, CLEMENS: *Component Software, Beyond Object–Oriented Programming*. Addison-Wesley, 1998. ISBN 0-201-17888-5.

[TAPM 99] *Introducing Tivoli Application Performance Management*. IBM Redbook SG24-5508-00, IBM Tivoli, Dezember 1999, `http://www.redbooks.ibm.com/abstracts/sg245508.html`.

[Tiv 99] *An Introduction to Tivoli Enterprise*. IBM Redbook SG24-5494-00, IBM Tivoli, Oktober 1999, `http://www.redbooks.ibm.com/abstracts/sg245494.html`.

[TMF 909] *Technology Integration Map (GB 909)*. Technischer Bericht, TeleManagement Forum, Oktober 1998, `ftp://ftp.tmforum.org/specifications/techmap/TechMap.doc`.

[TOM1.1] *SMART TMN Telecom Operations Map*. Evaluation Version 1.1 GB910, TeleManagement Forum, April 1999, `http://www.tmforum.com/pages/publications/books.html\#telops`.

[TSK 96] SCHADE, A., P. TROMMLER und M. KAISERSWERTH: *Object Instrumentation for Distributed Applications Management*. In: *Proceedings of the IFIP/IEEE International Conference on Distributed Platforms ICDP'96, Dresden, Germany*, 1996.

[Wald 95] WALDBUSSER, S.: *RFC 1757: Remote Network Monitoring Management Information Base*. RFC, IETF, Februar 1995, `ftp://ftp.isi.edu/in-notes/rfc1757.txt`.

[Wald 97] WALDBUSSER, S.: *RFC 2021: Remote Network Monitoring Management Information Base Version 2 using SMIv2*. RFC, IETF, Januar 1997, `ftp://ftp.isi.edu/in-notes/rfc2021.txt`.

[Wald 01] WALDBUSSER, S.: *Application Performance Measurement MIB*. Technischer Bericht, März 2001, `http://www.normos.org/ietf/draft/draft-ietf-rmonmib-apm-mib-03.txt`.

[WHP 99] WIJNEN, B., D. HARRINGTON und R. PRESUHN: *RFC 2571: An Architecture for Describing SNMP Management Frameworks*. RFC, IETF, April 1999, `ftp://ftp.isi.edu/in-notes/rfc2571.txt`.

INDEX

A

Abrechnungsmanagement, *siehe* Management
ACAP, *siehe* Application Configuration Access Protocol
ACAP WG, *siehe* Application Configuration Access Protocol Working Group
ACT, *siehe* Application Components Team
ActiveX, 12–14
Activity Monitoring Language, 71
Adapterklasse, 10, 11, 144, 145
ADF, *siehe* Application Description File
AIC, *siehe* Application Instrumentation & Control
AL, *siehe* Application Library
AML, *siehe* Activity Monitoring Language
AMP, *siehe* Application Management Package
AMS, *siehe* Application Management Specification
Antwortzeit, 26, 33, 46, 60, 72, 79, 97, 99, 100
  -überwachung, 55, 79
  -verhalten, 55, 56, 61
Anwendungs
  -beschreibung, 47, 48
  -dienst, 1, 2, 31–37, 45, 46, 49, 51–53, 68, 71, 72, 84, 85, 92, 121, 128, 141
  -entwickler, 3, 21, 34, 36–38, 48, 66, 84, 92, 93, 96, 98, 100, 106, 116, 117, 122, 131, 135, 136
  -entwicklung
    bausteinorientierte, 2–4, 7, 8
  -ersteller, 20
  -erstellung, 21, 40, 47, 66, 84, 92, 94, 98, 106, 116
  -instrumentierung, 47–50
  -logik, 21, 21, 36, 37, 39–41, 69, 84, 86, 87, 94–96, 109, 110, 112, 128
  -management, *siehe* Management
  -paket, 53, 54
  -protokoll, 55
  -überwachung, *siehe* Überwachung
  -zustand, 33
APM, *siehe* Application Performance Measurement
Application Performance Measurement MIB, 51
Application Instrumentation & Control, 65
Application Management MIB, 55
Application Response Measurement, 61
Application Components Team, 67
Application Configuration Access Protocol, 51, 56
Application Configuration Access Protocol Working Group, 55, 56
Application Description File, 75
Application Instrumentation & Control, 65
Application Instrumentation & Control, 50, 61, 65, 66
Application Library, 65, 66
Application Management MIB, 51, 53, 54
Application Management Package, 76
Application Management Specification, 57
Application Perfomance Measurement MIB, 55
Application Perfomance Measurement MIB, 55
Application Performance Measurement, 55
Application Performance Measurement MIB, 55
Application Response Measurement, 50, 54, 105, 136
Application Service Provider, 1
Application Service Provisioning, 54
ARM, *siehe* Application Response Measurement
ASP, *siehe* Application Service Provider

B

Baustein, 19–20
- -architektur, 8–19, 68, 69, 94, 112, 164
- -entwickler, 3, 21, 36, 37, 39, 69, 93, 95, 119, 123, 128, 131, 135
- aktiver, 20, 107, 109, 110, 113, 114, 124, 125, 128, 129, 144, 145, 147, 163
- architektur, 20
- Eingabe-, 20
- Oberflächen-, 20
- passiver, 20
- Präsentations-, 20
- Server-, 20

Beanbox, 10, 11, 143–158
Benutzeradministration, 24
Benutzerinteraktion, 26
Benutzertransaktion, 4, 25, 26, 45–48, 54, 58, 73, 96–100, 102, 104–111, 116–125, 128, 131, 133, 135
Bestandsführung, 24
BSM, *siehe* Business System Mapping
BTA, *siehe* Benutzertransaktion
Business System Mapping, 77

C

CAMI, *siehe* Common Application Management Interface
CDF, *siehe* Component Description File
CIM, *siehe* Common Information Model
CL, *siehe* Client Library
Client Library, 65
CMG, *siehe* Computer Measurement Group
COM, *siehe* Component Object Model
Common Application Management Interface, 67
Common Information Model, 50
Common Object Request Broker Architecture, 73
Component Description File, 76
Component Object Model, 12–14
Compound Document Model, 7
Computer Measurement Group, 61
CORBA, *siehe* Common Object Request Broker Architecture
CORBA Component Model, 18–19
Customization, 21

D

DAP WG, *siehe* Distributed Application Performance Working Group
DCE, *siehe* Distributed Computing Environment
DCOM, *siehe* Distributed Component Object Model
Dienst, 27
- -erbringer, 27
- -güteparameter, 29
- -implementierung, 29
- -kunde, 27
- -management, *siehe* Management, 29
- -nehmer, 27
- -nutzer, 27
- -orientierung, 26
- -vereinbarung, 29
- -zugangspunkt, 29
- Individual-, 2

Distributed Application Performance Working Group, 50
Distributed Component Object Model, 12
Distributed Computing Environment, 57
Distributed Management Interface, 56
Distributed Management Task Force, 50, 56, 126
DLL, *siehe* Dynamic Link Library
DMI, *siehe* Distributed Management Interface
DMTF, *siehe* Distributed Management Task Force
Document Type Definition, 126
DTD, *siehe* Document Type Definition
Dynamic Link Library, 13

E

Enterprise JavaBeans, 16–18
- Entity Beans, 16
- Session Beans, 16

EnterpriseJava Beans
- Message-driven Beans, 16

Entwicklungsumgebung, 8–11, 14, 17, 19–21, 21, 39, 96, 99, 101, 103, 112, 124, 128, 129, 143, 145, 163
Extensible Markup Language, 126
Extensible Markup Language, 17

F

Fehlermanagement, *siehe* Management

G

GAMOCs, *siehe* Generic Application Managed Object Classes
GDF, *siehe* Global Description File
GEM, *siehe* Global Enterprise Manager
Generic Application Managed Object Classes, 69
Global Description File, 75

Global Enterprise Manager, 73

H

Host Resources MIB, 51–53
HR MIB, *siehe* Host Resources MIB

I

IDL, *siehe* Interface Definition Language
IETF, *siehe* Internet Engineering Task Force
Individualdienst, *siehe* Dienst
Interception, 15, 17, 19, 164
Interface Definition Language, 57
Internet Engineering Task Force, 50
Inventory Management, *siehe* Management

J

Java
  Beans, 9–11, 16, 17, 143–158
  Messaging Service, 16

K

Konfigurationsmanagement, *siehe* Management
Kontrollfluß, 21

L

Leistungsmanagement, *siehe* Management

M

Mail Monitoring MIB, 51, 53
Managed Object Format, 57
Management
  -funktionalität, 28
  -funktionsbereiche, 23
  -instrumentierung, 24, 24, 47, 48, 53, 54, 64, 84, 92, 94, 162, 164
    Automation der, 143
    Automation der, 128, 142, 147
    Automation der, 2, 8, 18, 85, 95–114, 128–131, 143
  -schnittstelle, 3, 37, 41, 47, 68, 83, 85–87, 112, 120, 121, 126
  Abrechnungs-, 24, 71
  Anwendungs-, 23–26
  Desktop-, 25
  Dienst-, 26
  Fehler-, 24
  Inventory-, 24
  Konfigurations-, 24
  Leistungs-, 24
  Server-, 25
  Sicherheits-, 24
  System-, 24
Management Information Base, 50–52
Management Information Format, 56
MIB, *siehe* Management Information Base
MIF, *siehe* Management Information Format
MOF, *siehe* Managed Object Format
Monitoring, *siehe* Überwachung

N

Network Management Forum, 67
Network Services Monitoring MIB, 51
Network Services Monitoring, 52
Network Services Monitoring MIB, 52
NMF, *siehe* Network Management Forum
NSM, *siehe* Network Services Monitoring
Nutzungsfunktionalität, 28

O

ODP, *siehe* Open Distributed Processing
Open Distributed Processing, 70
Open Group, 50, 61, 64, 65
Open Software Foundation, 61
OSF, *siehe* Open Software Foundation
Outsourcing, 1

R

Realtime Traffic Flow Management, 51
Realtime Traffic Flow Measurement Working Group, 56
Relational Database Management System MIB, 51, 53
Request for Comment, 50
RFC, *siehe* Request for Comment
RMON MIB, 55
RMON2 MIB, 55
RTFM, *siehe* Realtime Traffic Flow Management
RTFM WG, *siehe* Realtime Traffic Flow Measurement Working Group

S

Service Provisioning, 1
Sicherheitsmanagement, *siehe* Management
Simple Network Management Protocol, 51
Simple Network Management Protocol Version 2, 51
SNMP, *siehe* Simple Network Management Protocol

SNMPv2, *siehe* Simple Network Management Protocol Version 2
Software
-Deployment, 24
-Distribution, 24
-Installation, 24
-Verteilung, 24
Steuerungsfokus, 22
Subtransaktion, 4, 26, 48, 60, 97, 102, 105–107, 109–111, 118–120, 123, 124, 128, 134
System Application MIB, 51, 53
Systemmanagement, *siehe* Management

T

TAPM, *siehe* Tivoli Application Performance Manager
TBSM, *siehe* Tivoli Business System Manager
Technology Integration Map, 67
Telecom Operations Map, 67
TeleManagement Forum, 67
TIM, *siehe* Technology Integration Map
Tivoli Application Performance Manager, 73
Tivoli Business System Manager, 77
Tivoli Management Framework, 73
Tivoli Module Builder, 77
Tivoli Module Designer, 77
TMB, *siehe* Tivoli Module Builder
TMD, *siehe* Tivoli Module Designer
TMF, *siehe* Tivoli Management Framework
TMForum, *siehe* TeleManagement Forum
TOM, *siehe* Telecom Operations Map
Transaktionsdauer, 26
Transaktionsüberwachung, *siehe* Überwachung

U

Überwachung, 24
Anwendungs-, 1–4, 16, 19, 23, 24–25, 43–46, 49, 50, 58, 61, 68, 73, 78, 79
Transaktions, 25
UML, *siehe* Unified Modelling Language
Unified Modelling Language, 9, 22, 57
Unit of Work, 58
UoW, *siehe* Unit of Work
User Administration, *siehe* Benutzeradministration

V

Visual Basic, 14

W

WWW Service MIB, 51, 54

X

X.500 Directory Monitoring MIB, 51, 53
XML, *siehe* Extensible Markup Language

Lebenslauf

Persönliche Daten:

Name:	Rainer Hauck
Anschrift:	Paul-Ehrlich-Weg 45 80999 München
Geburtsdatum	22. März 1972
Geburtsort:	München
Familienstand:	ledig
Eltern:	Alfred Otto Hauck, Dipl.–Ing. (FH) Hedwig Hauck, geb. Wirtz, Hausfrau

Ausbildung und beruflicher Werdegang:

Sep. 1978 – Aug. 1982:	Besuch der Grundschule an der Eversbuschstr. 182, München
Sep. 1982 – Juli 1991:	Besuch des städt. Louise–Schroeder–Gymnasiums, München
Nov. 1991 – Mai 1996:	*Studium der Informatik* an der Technischen Universität München Nebenfach: Elektrotechnik Vertiefungsfach: Rechnernetze Diplomarbeit: „Sicherheitskonzept für den BMW Internetzugang“
Dez. 1995 - Juni 1996:	*Technical Support Engineer* bei der Firma IQproducts GmbH, München
Aug. 1996 - Juli 2001:	*Wissenschaftlicher Mitarbeiter* an der Ludwig-Maximilians-Universität München, Institut für Mathematik und Informatik, Lehrstuhl für Kommunikationssysteme und Systemprogrammierung