Universität Stuttgart

Institut für parallele und verteilte Höchstleistungsrechner

Prüfer: Prof. Dr. A. Reuter

Betreuer: Dipl.-Inform. F. Schwenkreis

Diplomarbeit Nr. 1312

Synchronisation in APRICOTS

Holger Schwarz

Beginn am: 16.6.1995

Beendet am: 29.12.1995

CR-Klassikikation: C.2.4, H.2.4

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Überblick

Große verteilte Systeme stellen erhöhte Anforderungen an Zuverlässigkeit und Fehlertoleranz. Diesen wird das klassische Transaktionskonzept nur begrenzt gerecht. Für Anwendungen mit großer Verweildauer im System wurde darum das ConTract-Modell entwickelt[1]. APRICOTS[2] stellt eine prototypische Implementierung eines ConTracts verarbeitenden Systems dar. Eine wichtige Eigenschaft des Systems ist, daß anwendungsabhängige "Isolationsgrade" programmiert werden können. Diese werden mit Hilfe von Prädikaten über globale Datenobjekte definiert, den sogenannten Invarianten.

Diese Arbeit beschäftigt sich mit Möglichkeiten, die Verwaltung dieser Invarianten in die bestehende Architektur von APRICOTS zu integrieren. Hierzu werden drei Architekturvorschläge erarbeitet. Diese werden anhand verschiedener Kriterien verglichen und schließlich eine Architekturvariante als die geeignetste ausgewählt. Die weitere Arbeit beschreibt dann die prototypische Implementierung eines Laufzeitsystems für die Aufgaben der Synchronisation gemäß der ausgewählten Architektur. Zunächst aber ein Überblick über den Aufbau der Arbeit.

1.1 Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen für das Verständnis der weiteren Arbeit gelegt. Hierzu gehört eine Motivation für die Einführung des ConTract-Modells, sowie die Erläuterung der wichtigsten Aspekte des Modells. Hinzu kommt eine kurze Beschreibung der Architektur von APRICOTS.

Nach dieser Einführung werden in Kapitel 3 die verschiedenen Architekturvarianten vorgestellt. Es folgt eine Darstellung der Kriterien mit deren Hilfe anschließend die geeignetste Architektur ermittelt wird.

Die gewählte Architekturvariante wird in Kapitel 4 detaillierter untersucht. Hierbei werden verschiedene Methoden zur Optimierung eingeführt. Außerdem folgt eine Betrachtung der für die Verwaltung von Invarianten notwendigen Informationen.

Kapitel 5 beschäftigt sich schließlich mit der konkreten Implementierung von Step-Servern und

1. Vgl. [WäRe90]
2. A PRototype Implementation of the COnTract System

Resource-Managern gemäß der gewählten Architekturvariante. Hier sollen besonders die Schnittstellen, verwendete Datenstrukturen und Algorithmen betrachtet werden. Zusätzlich folgen für jede Komponente Erläuterungen, wie der Prototyp für eine vollständige Integration in APRICOTS erweitert werden muß.

In Kapitel 6 werden die Ergebnisse der Arbeit zusammengefaßt und ein Ausblick gegeben.

Kapitel 2

ConTracts und APRICOTS

Zunächst müssen die Grundlagen für das Verständnis der weiteren Arbeit gelegt werden. Dieses Kapitel zeigt hierzu zunächst die Grenzen des klassischen Transaktionenkonzepts auf. Anschließend folgt eine kurze Beschreibung des ConTract-Modells und die Darstellung der bestehenden Architektur von APRICOTS.

2.1 ACID-Transaktionen

ACID-Transaktionen stellen das in den 70-er Jahren entwickelte Transaktionskonzept für Datenbanksysteme dar. Ausgangspunkt ist der konkurrierende Zugriff mehrerer paralleler oder quasi-paralleler Anwendungen auf Elemente einer gemeinsamen Datenbasis. Gemeinsames Merkmal dieser Elemente ist ihr einfacher Aufbau und die Ähnlichkeit der Struktur. Auf dieser Grundlage lassen sich folgende Anforderungen an ein Transaktionskonzept ableiten:

- Die konkurrierenden Zugriffe der verschiedenen Anwendungen auf die gemeinsamen Objekte sollen synchronisiert werden.
- Nach der Ausführung von Datenbankoperationen sollen die Ergebnisse stabil sein.
- Eine Reihe von Datenbankoperationen soll zu einer logisch zusammenhängenden Ausführungseinheit gruppiert werden.

Diesen Anforderungen genügt das Konzept der ACID-Transaktionen. Sie fassen mehrere Datenbankoperationen zu kleinen überschaubaren Einheiten zusammen. Charakteristische Merkmale einer solchen Transaktion sind:

Atomicity: Dies bedeutet, daß jede Transaktion komplett oder gar nicht ausgeführt wird. Sie stellt damit die kleinste Ausführungseinheit in einem Datenbanksystem dar.

Consistency: Jede Transaktion kann davon ausgehen daß sie bei ihrem Beginn die Datenbank in einem konsistenten Zustand vorfindet. Sie muß dann garantieren, daß nach ihrem Ende wieder ein konsistenter Zustand vorliegt. Dies wird entweder durch die erfolgreiche Ausführung der Transaktion oder mit deren vollständigem Zurücksetzen erreicht.

Isolation: Von der Arbeit einer Transaktion wird nur der Endzustand sichtbar. Dazwischen liegende Zustandsübergänge werden vor den anderen Transaktionen verborgen. Es entsteht somit ein virtueller Einbenutzerbetrieb.

Durability: Alle Änderungen, die eine Transaktion an Objekten durchführt, werden stabil gespeichert.

Mit Hilfe von Transaktionen, die nach diesem Konzept gestaltet sind, läßt sich eine breite Palette von Anwendungen realisieren. Diese sind im Bereich der administrativen Datenverarbeitung angesiedelt, wie z.B. Lager- und Buchhaltungssysteme. Transaktionen, die typischerweise im Rahmen solcher Anwendungen eingesetzt werden, sind durch folgende gemeinsame Eigenschaften gekennzeichnet:

- Die einzelnen Transaktionen greifen nur auf wenige Elemente der Datenbank zu. Dies garantiert, daß bei einem Abbruch einer Transaktion nur wenige Daten wieder in ihren alten Zustand zurückversetzt werden müssen.

- Die Verweildauer der einzelnen Transaktionen im System ist gering. Hierdurch wird das Konfliktpotential mit konkurrierenden Transaktionen verringert, da Zugriffskonflikte nur kurzzeitig bestehen.

Sobald eine Anwendung Transaktionen benötigt, die eine dieser Eigenschaften nicht aufweisen, ist der sinnvolle Einsatz von ACID-Transaktionen in Frage gestellt. Dies tritt bei einer ganzen Klasse von Anwendungen auf. Hierzu gehören Systeme zur Büroautomatisierung, CAD-, CAM- und CIM-Systeme sowie Workflow-Systeme. Werden solche Anwendungen mit Hilfe von ACID-Transaktionen realisiert, so weisen diese oft eine große Verweildauer im System auf, während der sie auf eine große Anzahl von Daten zugreifen und diese verändern.

Dies mindert die Performance des Gesamtsystems erheblich. Ein Grund hierfür ist, daß während solcher langen Transaktionen alle Sperren auf einmal angefaßte Daten gehalten werden müssen. Die Parallelität wird dadurch stark eingeschränkt. Außerdem erhöht sich der Aufwand für eventuell notwendige Recovery-Maßnahmen. Da die Transaktionen aus sehr vielen Verarbeitungsschritten bestehen, benötigt auch die Recovery lange Zeit. Das Konzept der ACID-Transaktionen findet also bei langlebigen Transaktionen seine Grenzen[3]. Bei der Lösung dieser Probleme haben sich vor allem zwei Eigenschaften von ACID-Transaktionen als nachteilig erwiesen:

- Sie sind kontextfrei. Damit hat die Transaktion keinen Zugriff auf den internen Zustand der Anwendung. Der Kontrollfluß zwischen den Transaktionen, aus denen sich eine Anwendung zusammensetzt kann nicht modelliert werden.

- Sie können nicht verzweigt oder geschachtelt werden. Dies stellt für umfangreiche, verteilte Anwendungen ein Problem dar.

Damit sind die Ausgangspunkte für das ConTract-Modell aufgezeigt. Dieses soll nun soweit beschrieben werden, wie es als Voraussetzung für die weitere Arbeit notwendig erscheint.

2.2 Das ConTract-Modell

Mit dem ConTract-Modell wird ein transaktionsübergreifender Mechanismus eingeführt, der die kontrollierte und zuverlässige Abwicklung langlebiger Aktivitäten bei der Verarbeitung komplexer Objekte ermöglicht[4]. Beschrieben wird die Struktur von Abläufen. Alle Kontrollaufgaben, wie z.B. Recovery und Synchronisation werden vom System übernommen und damit vor dem Anwendungsprogrammierer verborgen. Die Zuverlässigkeit der ConTract-Verarbeitung beruht auf der folgenden Konsistenzeigenschaft: Ein ConTract terminiert unter Garantie in endlicher Zeit in einem korrekten Endzustand. Dies bedeutet, daß mögliche Systemfehler nicht das Ende der Aktivität nach sich ziehen, sondern das System soll die Verarbeitung fortsetzen, solange hierfür mindestens ein geeigneter Knoten zur Verfügung steht. Fortsetzung kann hier auch die kontrollierte Wiederherstellung des Anfangszustandes bedeuten.

2.2.1 Programmiermodell

ConTracts weisen ein zweistufiges Programmiermodell auf. Dieses trennt die Beschreibung der Ablaufstruktur (Skript) von den elementaren Berechnungsschritten (Steps). Als Folge hiervon kann auch die Programmierung von Skript und Steps getrennt durchgeführt werden. Den Verantwortlichen für beide Bereiche wird die Arbeit erleichtert. Bei der Erstellung der Steps müssen Dinge wie Parallelität, Kommunikation, verteilte Ressourcen, Synchronisation und Recovery nicht berücksichtigt werden. Programmierer eines Skripts können dagegen auf vorgefertigte Steps zurückgreifen und sind lediglich

3. Vgl. [Gray81]
4. Vgl. [WäRe90], S. 203

dafür verantwortlich, deren korrekte Abfolge festzulegen. Die interne Realisierung der Steps muß hierfür nicht berücksichtigt werden. Die Elemente eines ConTracts haben im einzelnen folgende Aufgaben:

Steps

Als elementare Teilschritte einer Anwendung umschließen Steps logisch zusammenhängende Aktionen. Diese Basisaktionen enthalten intern keine Parallelität, sondern bestehen aus rein sequentiellem Code. Aus der Sicht des Programmierers wird jeder Step auf einer beliebig ausfallsicheren virtuellen Maschine ausgeführt. Fehlersituationen müssen hier nicht berücksichtigt werden. Statt dessen wird für jeden Step eine Kompensationsaktion angegeben.

Skript

Das Skript beschreibt den Kontrollfluß der Anwendung. Ein Beispiel wird in Abschnitt 2.2.5 erläutert. Jedes ConTract-Skript setzt sich aus folgenden Elementen zusammen:

Kontrollflußbeschreibung: Hier wird festgelegt, unter welchen Bedingungen die Steps in welcher Reihenfolge ablaufen. Hierzu können die üblichen Kontrollkonstrukte (Sequenz, Verzweigung, Schleifen, Parallelität, ...) verwendet werden.

Kontextdefinition: Der Kontext enthält die persistenten Variablen der Anwendung. Sie stellen den aktuellen Verarbeitungszustand des ConTracts dar. Auf diesen muß auch nach Ausfall einer Systemkomponente zurückgegriffen werden können.

Transaktionsbeschreibung: Sie legt fest, welche Steps zu Transaktionen zusammengefaßt werden und wie deren Abhängigkeiten aussehen.

Invariantenbeschreibung: Die Invarianten ermöglichen die Synchronisation parallel ablaufender Steps. Dieses wird in Abschnitt 2.2.4 ausführlicher diskutiert.

Kompensationsbeschreibung: Hier wird für jeden Step die Aktion angegeben, die im Fall des Abbruchs Konsistenzerhaltung gewährleistet. Näheres hierzu in Abschnitt 2.2.3.

2.2.2 Recovery

In einem ConTracts verarbeitenden System soll dafür gesorgt werden, daß Veränderungen am System, Kommunikationsfehler, Knotenausfälle und andere Fehler nicht zu einem undefinierten Zustand der Anwendung führen. Bei der Verwendung klassischer Transaktionen stellt das System dafür keine Dienste zur Verfügung. Die Anwendungsprogrammierer müssen hier selbst die notwendigen Vorkehrungen treffen.

- Ein gewöhnlicher Betriebssystemprozeß ist nach dem Systemausfall verloren. Der Anwender muß wissen, welche Anwendungen betroffen sind, in welchem Zustand sie abgebrochen wurden und wie erneut aufgesetzt werden kann.

- Nach einem Systemausfall setzt ein Transaktionssystem einen konsistenten Zustand wieder her. Hierbei werden alle nicht abgeschlossenen Transaktionen auf ihren Anfangszustand zurückgesetzt. Bei langdauernden Aktivitäten dauert dies zu lange und es geht zu viel erfolgreiche Arbeit verloren.

In einem ConTracts-System soll das Verhalten nach einem Systemausfall anders gestaltet sein. Hier wird eine sogenannte Forward-Recovery angestrebt. Dies bedeutet, daß nach der Unterbrechung ein konsistenter Zustand wiederhergestellt werden soll, der die Fortsetzung des ConTracts ermöglicht. Hierdurch wird sichergestellt, daß möglichst wenig erfolgreiche Arbeit verloren geht. Voraussetzung ist allerdings, daß der aktuelle Zustand eines ConTracts auf stabilem Speicher festgehalten wird. Wichtige Zustandsinformationen sind:

- Der globale Systemzustand.

- Der lokale Zustand eines ConTracts. Hierzu gehören z.B. Programmvariablen, die in mehreren Steps genutzt werden.

- Der globale Verarbeitungszustand der Anwendung. Hier wird festgehalten, welche Steps bereits ausgeführt wurden und welche nicht.

2.2.3 Kompensation

ACID-Transaktionen sperren globale Objekte, auf die einmal zugegriffen wurde, für die gesamte Dauer der Transaktion. Dies ist bei langdauernden Anwendungen nicht zu tolerieren, da die Performance des

Systems erheblich sinkt. Also müssen im ConTract-Modell einzelne Objekte schon vor dem ConTract-Ende wieder freigegeben werden. Dies kann jeweils an den Übergängen zwischen Transaktionen, die ein oder mehrere Steps zusammenfassen, erfolgen. Ist ein Objekt einmal freigegeben, so können andere ConTracts darauf zugreifen. Dies führt zu Problemen beim Rücksetzen eines ConTracts. Hierbei sollen auch solche Steps zurückgesetzt werden können, deren umgebende Transaktion bereits abgeschlossen ist. In anderen Fällen würde das Rollback einer Transaktion auch die Aktivitäten von Steps eines anderen ConTracts zunichte machen. Es wird deutlich, daß es zur Kompensation eines ConTracts nicht ausreicht, Transaktionen einfach zurückzusetzen. Darum wird in diesem Modell eine andere Kompensationsmethode verwendet. Zu jedem Step wird ein Kompensationsstep programmiert, der ein semantisches Undo der von einem Step vorgenommenen Veränderungen an globalen Datenobjekten durchführt.

Aus der Verwendung von Kompensationsaktionen ergeben sich folgende Anforderungen an das System, die ein korrektes Funktionieren des Mechanismus gewährleisten sollen:

- Für jeden Step im Skript muß genau ein Step zur Kompensation existieren.

- Am Ende eines Steps müssen alle Daten vorliegen, die für dessen Kompensation notwendig sind. Aktuelle Werte globaler Objekte müssen bei Bedarf im Kontext gespeichert werden.

- Für alle von einem Step angefaßten Objekte, die auch für dessen Kompensation von Bedeutung sind, muß deren Existenz bis zum Ende des ConTracts sichergestellt sein.

- Soll ein ConTract komplett zurückgesetzt werden, so dürfen auf anderen Knoten ablaufende Steps keine weiteren Steps mehr anstoßen. Alle Steps müssen abgebrochen und kompensiert werden, sobald die Entscheidung über das Rücksetzen des ConTracts auf allen betroffenen Knoten bekannt ist. Vorher kann ein ConTract nicht abgeschlossen werden.

- Kompensationsaktionen können ebenfalls abgebrochen werden. Für diesen Fall ist die Durchführung der Kompensation solange zu versuchen, bis sie erfolgreich verlaufen ist.

- Läßt sich eine Kompensation auch nach der als Obergrenze festgelegten Anzahl von Versuchen nicht durchführen, so muß dieses Problem von außen durch einen Systemadministrator behoben werden.

2.2.4 Synchronisation

Innerhalb eines ConTracts werden Sperren auf Objekte jeweils an den definierten Transaktionsgrenzen aufgehoben. Damit haben noch während ein ConTract abläuft andere die Möglichkeit, auf bereits veränderte Objekte zuzugreifen. Es gibt aber trotzdem Bedingungen, die für ein korrektes Ablaufen der Anwendung über die einzelnen Transaktionsgrenzen hinaus erfüllt sein müssen. Diese werden in Form von Invarianten formuliert. Sie ermöglichen die Festlegung von Prädikaten über Objekte, deren Einhaltung der Anwendung eine ausreichende Isolation garantieren.

Zwei Arten von Invarianten werden unterschieden. Ausgangsinvarianten beschreiben die Bedingungen, denen ein Objekt gehorchen soll, nachdem ein bestimmter Step beendet wurde. Eingangsinvarianten legen Eigenschaften von Objekten fest, die für die korrekte Ausführung eines Steps notwendig sind. Welche Ausdrucksmächtigkeit für diese Invarianten sinnvoll ist und wie eine Syntax aussehen kann ist in [Schw95] festgelegt. Beispiele für den Einsatz von Invarianten sind beim Beispielskript des folgenden Abschnitts aufgeführt.

Die Verarbeitung von Invarianten kann nach drei unterschiedlichen Strategien erfolgen:

1. Strategie: Der Wert eines Objekts wird mit Hilfe von Sperren oder anderen Synchronisationstechniken “eingefroren”. Dies bedeutet, daß Sperren über Transaktionsgrenzen hinaus gehalten werden müssen.

2. Strategie: Es wird sichergestellt, daß Objektwerte nur dann geändert werden können, wenn hierdurch keine der existierenden Invarianten verletzt wird. Wird eine Invariantenverletzung festgestellt, so kann der Zugriff auf das Objekt nicht zugelassen werden.

3. Strategie: Bei Zugriffen auf Objekte wird eine Verletzung von Invarianten nicht überprüft. Dies erfolgt erst beim Start eines Steps. Hier werden sämtliche Eingangsinvarianten des Steps überprüft. Gibt es Objekte, die eine der Invarianten verletzen, so müssen im Skript die für die Konfliktauflösung notwendigen Schritte angegeben sein.

Dieser Arbeit liegt die zweite Strategie zugrunde. Dies bedeutet, daß bei sämtlichen Objektzugriffen eine Überprüfung der Invarianten notwendig wird.

Nachdem in den vorigen Abschnitten alle wichtigen Merkmale von ConTracts erklärt wurden, soll nun ein Beispiel für ein Skript näher erläutert werden.

2.2.5 Beispielskript

Abbildung 1 stellt ein exemplarisches Skript in einer graphischen Notation dar[5]. Mit Hilfe dieses Beispiels sollen in der weiteren Arbeit verschiedene Sachverhalte erklärt werden. Das Skript ist ein Steuerprogramm für einen Montagestand mit mehreren Robotern, an dem einzelne Teile eines Autos montiert werden. Dies sind Vorderräder, Hinterräder, die Windschutzscheibe und die Heckscheibe. Sind diese vier Teile montiert, so erfolgt eine Endkontrolle für diesen Montagestand. Die einzelnen Arbeitsschritte werden jeweils durch Steps eines ConTracts realisiert. Geht man davon aus, daß ein Montagestand desselben Typs mehrfach vorhanden ist, so muß dieser ConTract in mehreren Instanzen parallel ablaufen. Alle Montagestände desselben Typs greifen auf einen einzigen Materialpuffer zu, der einen kleinen Sicherheitspuffer zwischen der Just-in-time-Anlieferung der Einzelteile und den Montagerobotern darstellt. Der Bestand des Puffers reicht jeweils nur für wenige Montageschritte. Die gemeinsame Nutzung des Materialpuffers bedingt eine Synchronisation der Zugriffe darauf. Das Unterbinden aller Zugriffe auf den Puffer während des Ablaufs eines ConTracts würde sämtliche parallel arbeitenden Montagestände blockieren und ist darum nicht akzeptabel. Der Einsatz von ConTracts kann diese Blockierung verhindern.

Der ConTract für einen Montagestand ist folgendermaßen aufgebaut. Zunächst wird ein Chassis von den vorgelagerten Montageständen übernommen. Danach folgen zwei parallele Abläufe. Ein Roboter sorgt nacheinander für das Anbringen der Vorderräder und der Frontscheibe. Ein anderer Roboter montiert die Hinterräder und die Heckscheibe. Sind diese Aufgaben alle erfolgreich abgeschlossen, so erfolgt die Endkontrolle für diesen Montagestand.

Invarianten sind für diesen ConTract an folgenden Stellen notwendig: Sofern ein neues Chassis (CH) von der Vormontage übernommen werden kann muß für die weiteren Steps sichergestellt sein, daß von den Vorderrädern (VR) und den Hinterräder (HR) jeweils zwei Stück zur Verfügung stehen. Außerdem muß der Zugriff auf eine Windschutzscheibe (WS) und eine Heckscheibe (HS) möglich sein. Dies wird durch die Invarianten VR>=2, HR>=2, WS>=1 und HS>=1 dargestellt. Diese Invarianten können jeweils dann aufgehoben werden, wenn ein Step beendet ist, der die Montage des entsprechenden Teils gesteuert hat. So wird z.B. VR>=2 nach dem Step Vorderräder_anbringen gestrichen.

Im Beispielskript von Abbildung 1 sind die notwendigen Steps zur Kompensation nicht dargestellt. Diese werden ausgeführt, wenn bei einem der Steps ein Fehler auftritt. Im Step Vorderräder_anbringen kann es beispielsweise passieren, daß die gewünschte Ausgangsinvariante FS>=1 vom System nicht akzeptiert wird, weil gerade keine einzige Frontscheibe im Materialpuffer vorhanden ist.

5. Die Notation orientiert sich an [WäRe91]

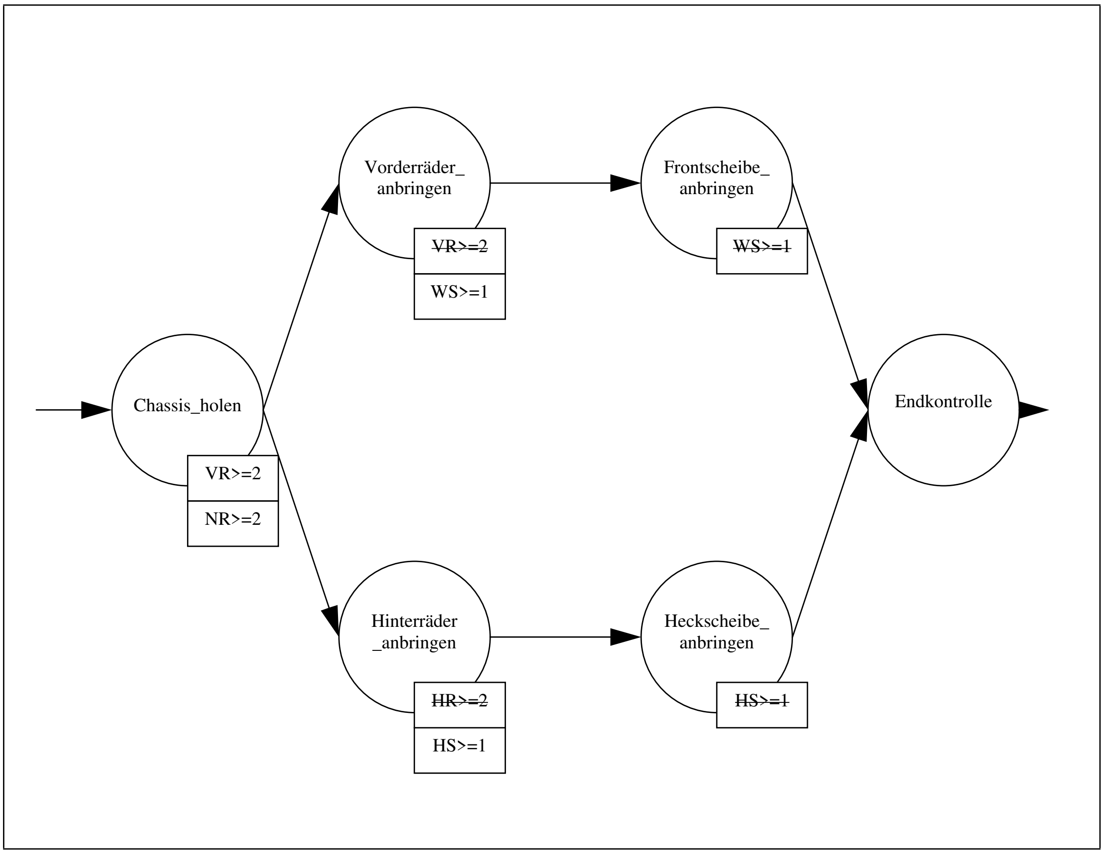

Abb. 1: Beispielskript

2.3 APRICOTS

APRICOTS ist die prototypische Implementierung eines Systems zur Verarbeitung von ConTracts. Die bestehende Architektur des Prototypen ist in Abbildung 2 dargestellt[6]. Hier sind alle zum weiteren Verständnis der Arbeit notwendigen Komponenten des Prototyps abgebildet. Wichtig an dieser Architektur ist der modulare Aufbau. Dieser führt zu einer großen Flexibilität des Systems. Einzelne Komponenten können leicht hinzugefügt werden oder durch überarbeitete Versionen ersetzt werden. In diesem Abschnitt soll zu jeder Komponente kurz beschrieben werden, welche Funktionalität sie aufweist, welche Schnittstellen sie besitzt und wie die Zusammenarbeit mit anderen Komponenten prinzipiell gestaltet ist.

6. Vgl. [Schü93], S. 15-19

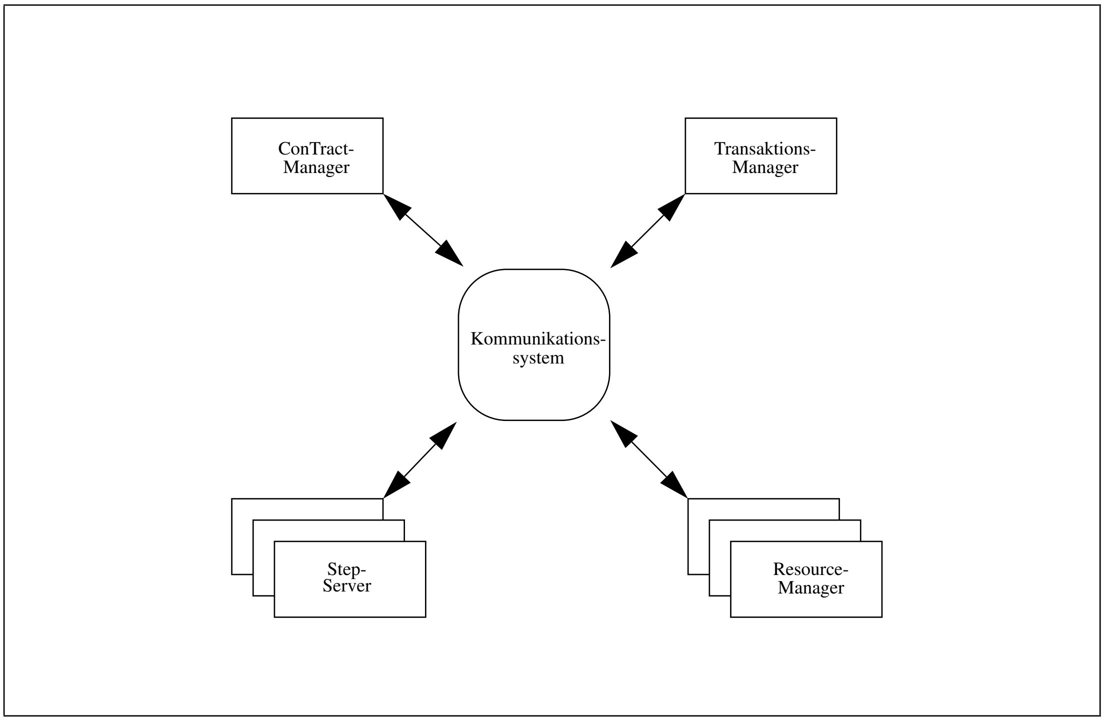

Abb. 2: Architektur von APRICOTS

2.3.1 Der ConTract-Manager

Die Aufgabe dieser Komponente ist es, für eine korrekte und zuverlässige Abwicklung eines ConTracts zu sorgen. Ausgehend vom vorliegenden Skript muß der Kontrollfluß interpretiert werden. Steps und Transaktionsgrenzen sind zu initiieren. Zusätzlich ist der ConTract-Manager dafür verantwortlich, daß ein ConTract auch nach einem Systemausfall oder einem ähnlichen Fehler fortgesetzt wird. Eine weitere Aufgabe ist das Kontextmanagement. Hierbei muß der für die Ausführung eines Steps notwendige Kontext bereitgestellt und aktualisiert werden. Zwischen ConTract-Manager und Step-Servern findet ein bidirektionaler Nachrichtenfluß statt.

2.3.2 Die Step-Server

Ihnen obliegt die Verwaltung der Steps, die für die Ausführung von ConTracts notwendig sind. Zur Ausführung eines bestimmten Steps wird ein Step-Server durch einen ConTract-Manager aufgefordert. Damit übernimmt der Step-Server für den ConTract-Manager die Rolle eines Servers. Die Steps selbst nehmen bei der Ausführung die Dienste verschiedener Resource-Manager in Anspruch. Für diese ist jeder Step-Server damit ein Client. Zwischen den Step-Servern und dem ConTract-Manager findet ebenfalls bidirektionale Kommunikation statt.

2.3.3 Die Resource-Manager

Globale Objekte werden durch Resource-Manager verwaltet. Als Beispielobjekte seien Fertigungsmaschinen, Drucker, Scanner oder Datenbanken genannt. Gegenüber den Step-Servern bieten die Resource-Manager Dienste an. Die verschiedenen Resource-Manager können in Klassen organisiert werden. Ein Beispiel für eine solche Klasse sind Druckerdienste. Diese Klasse kann dann Resource-Manager für Laserdrucker und solche für Matrixdrucker enthalten. Die Kommunikation zwischen den Resource-Managern und ihrer Umgebung ist bidirektional, d.h. der Resource-Manager wird - mit Parametern versorgt - aufgerufen und beendet seinen Aufruf mit einer Parameterrückgabe bzw. einer Statusmeldung.

2.3.4 Der Transaktionsmanager

Ein oder mehrere Steps werden jeweils zu Transaktionen zusammengefaßt. Die Kontrolle und Abwicklung dieser Transaktionen ist Aufgabe des Transaktions-Managers. Hierzu wird ein 2-Phasen-Commit-Protokoll verwendet. Ein ConTract-Skript erlaubt außerdem die Darstellung von Beziehungen zwischen Transaktionen. So kann z.B. der Abbruch einer Transaktion den Abbruch parallel ablaufender Transaktionen nach sich ziehen. Der Transaktions-Manager ist für die Verwaltung solcher Beziehungen zuständig.

2.3.5 Das Kommunikationssystem

Das Kommunikationssystem stellt die Verbindung zwischen den einzelnen Komponenten her. Dies setzt eine zuverlässige und korrekte Übermittlung von Nachrichten voraus. Aufgaben des Kommunikationssystems sind hierbei die Installation von Kommunikationskanälen, die Empfänger- und Adreßbestimmung und die Nachrichtenversendung zur Dienstinitiierung.

Kapitel 3

Synchronisation in APRICOTS

3.1 Aufgaben der Synchronisationsverwaltung

Sobald mehrere ConTracts parallel ablaufen, müssen deren Zugriffe auf gemeinsame Objekte synchronisiert werden. Dies erfolgt im ConTract-Modell wie in Abschnitt 2.2 beschrieben mit der Hilfe von Invarianten. Diese Invarianten müssen zusammen mit der notwendigen Verwaltungsinformation gespeichert werden. Alle Teile von APRICOTS-Modulen, die Invarianten oder Verwaltungsinformation für diese speichern und verarbeiten, werden in dieser Arbeit als Synchronisationsverwaltung bezeichnet. In diesem Abschnitt sollen nun die Aufgaben dieser Synchronisationsverwaltung näher erläutert und mit Hilfe des Beispieskripts aus Abschnitt 2.2.5 verdeutlicht werden.

Invarianten einführen

Der ConTract-Manager liest Invarianten aus dem Skript. Jede Invariante übergibt er direkt der Synchronisationsverwaltung zur weiteren Verarbeitung. Diese besteht aus folgenden Schritten:

- Die Invariante wird zusammen mit der erforderlichen Verwaltungsinformation gespeichert.
- Es muß überprüft werden, ob die Invariante gültig ist. Nur dann kann die Invariante akzeptiert werden. Hierbei muß die aktuelle Wertebelegung aller in der Invariante auftauchenden Objekte berücksichtigt werden.
- Es muß überprüft werden, ob die neue Invariante vereinfacht werden kann und ob eine Zusammenfassung mit anderen Invarianten möglich ist.
- Es muß als Antwort zurückgegeben werden, ob die Invariante akzeptiert wird oder nicht.

Im Beispielskript tritt diese Situation unter anderem am Ende des Steps Chassis_holen auf. Hier werden die neuen Invarianten VR>=2 und HR>=2 an die Synchronisationsverwaltung übergeben. Der weitere Kontrollfluß im Skript hängt davon ab, ob die Invariante akzeptiert wurde oder nicht.

Invarianten löschen

Nicht alle Invarianten sind bis zum Ende eines ConTracts für dessen korrekten Ablauf notwendig. So kann im Beispielskript am Ende des Steps Vorderräder_anbringen die Invariante VR>=2 gelöscht werden. Da die notwendigen Räder bereits aus dem Materialpuffer entnommen sind, hat die Anzahl der im Puffer vorhandenen Räder keinen Einfluß auf die weitere Abarbeitung des ConTracts. Die Synchronisationsverwaltung muß in einer solchen Situation folgendes leisten:

- Alle Invarianten, die durch Zusammenfassungen aus der zu löschenden Invariante hervorgegangen sind, müssen für ungültig erklärt werden.

- Die Invariante muß gelöscht werden.

- Mit den verbleibenden Invarianten kann erneut versucht werden, ob es Möglichkeiten gibt, diese zu vereinfachen und zusammenzufassen.

- Als Antwort erfolgt die Bestätigung des Löschvorgangs, falls dieser erfolgreich durchgeführt werden konnte.

Objektänderungen

Innerhalb von Steps können einzelne Datenobjekte, über die Invarianten existieren, verändert werden. Jede Veränderung ist aber nur dann möglich, wenn auch durch den neuen Wert eines Objekts die Gültigkeit aller Invarianten zu diesem Objekt erhalten bleibt. So kann z.B. der Step Hinterräder_anbringen des Beispielskripts nur dann zwei Räder aus dem Materialpuffer entnehmen, wenn hierdurch die Invarianten über dem Objekt HR nicht verletzt werden. Für die Synchronisationsverwaltung ergeben sich die folgenden Aufgaben:

- Auswertung aller betroffenen Invarianten unter der Annahme, daß das Objekt seinen neuen Wert erhält. Nur wenn alle Invarianten weiterhin gültig sind kann die Objektänderung akzeptiert werden.

- Rückmeldung, ob die Objektänderung zugelassen werden kann oder nicht.

Ende eines ConTracts

Sobald ein ConTract beendet ist können sämtliche Invarianten gestrichen werden, die von diesem ConTracts stammen. Die Synchronisationsverwaltung hat hier folgende Aufgaben:

- Alle Invarianten, die durch Zusammenfassung aus einer oder mehreren Invarianten des beendeten ConTracts hervorgegangen sind müssen gelöscht werden.

- Die Invarianten des ConTracts müssen gelöscht werden.

- Mit den verbleibenden Invarianten kann erneut versucht werden, ob Vereinfachungen und Zusammenfassungen möglich sind.

- Rückmeldung, ob alle zum ConTract-Ende gehörenden Aufgaben erledigt werden konnten.

Invariante auswerten

Das Auswerten einer Invariante kann auch ohne eine Objektänderung notwendig werden. Ein Beispiel hierfür sind die Eingangsinvarianten. Sie stellen zu Beginn eines Steps sicher, daß alle für die korrekte Ausführung des Steps notwendigen Voraussetzungen erfüllt sind. Die Synchronisationsverwaltung hat hier folgende Aufgaben:

- Auswertung der zu überprüfenden Invariante.

- Rückmeldung, ob die Invariante erfüllt ist oder nicht.

3.2 Architekturvarianten

Die Synchronisationsverwaltung kann in APRICOTS auf unterschiedliche Art und Weise realisiert werden. Dieser Abschnitt stellt die wichtigsten Merkmale der in dieser Arbeit betrachteten drei Architekturvorschläge vor. Die Architekturvarianten unterschieden sich in erster Linie darin, in welchen Modulen die Invarianten gespeichert und ausgewertet werden. Betrachtet wird eine globale Synchronisationsverwaltung, die Synchronisationsverwaltung in den Resource-Managern und eine Variante, bei der die Synchronisationsverwaltung direkt an die einzelnen Objekte gekoppelt ist.

Die folgende Darstellung enthält zu jeder Variante eine kurze Beschreibung, eine Auflistung der

Funktionen, die einzelne APRICOTS-Module übernehmen müssen und eine Darstellung der notwendigen Interaktion zwischen verschiedenen APRICOTS-Komponenten bei der Erledigung der wichtigsten Aufgaben der Synchronisationsverwaltung (neue Invariante, Invariante löschen, Zugriffskontrolle auf Objekte und Ende eines ConTracts). Das Auswerten einer Invariante als separate Aufgabe der Synchronisationsverwaltung wird hier nicht betrachtet. Der Grund hierfür liegt darin, daß die für diese Aufgabe notwendigen Kommunikationsbeziehungen denen beim Etablieren einer neuen Invariante entsprechen. Es ergeben sich also für den Architekturvergleich keine neuen Aspekte

3.2.1 Globale Synchronisationsverwaltung

Die globale Lösung bedeutet, daß eine zusätzliche Komponente in APRICOTS eingeführt wird, mit ausschließlicher Zuständigkeit für die Aufgaben der Synchronisationsverwaltung. Diese Komponente wird im weiteren Synchronisations-Manager genannt. Es muß sich hierbei um eine gänzlich separate Komponente handeln. Der Grund hierfür ist, daß alle anderen Komponenten von APRICOTS mehrfach im System vorhanden sein können. Dies gilt für Step-Server, Resource-Manager und genauso für ConTract-Manager. Jede Anbindung der Synchronisationsverwaltung an eine oder mehrere dieser Komponenten stellt damit keine vollständig globale Lösung dar. Trotzdem müssen auch Step-Server und Resource-Manager Dienste anbieten, die für die Synchronisationsverwaltung benötigt werden. Die Aufgaben der verschiedenen Komponenten werden im folgenden Abschnitt näher erläutert.

3.2.1.1 Aufgaben der Komponenten

ConTract-Manager

Der ConTract-Manager liest neue Invarianten, zu löschende Invarianten oder die Information über das Ende eines ConTracts aus dem Skript. Zu jeder Anweisung ruft er den entsprechenden Dienst des Synchronisations-Managers auf.

Synchronisation-Manager

Bei der globalen Architektur übernimmt der Synchronisations-Manager praktisch die kompletten Aufgaben der Synchronisationsverwaltung, wie sie in Abschnitt 3.1 beschrieben sind. Im einzelnen ist er für folgendes verantwortlich:

Neue_Invariante: Der Synchronisations-Manager nimmt neue Invarianten entgegen und speichert diese zusammen mit der notwendigen Verwaltungsinformation.

Invariante_löschen: Auch das Löschen von Invarianten wird vollständig vom Synchronisations-Manager durchgeführt.

Invarianten_prüfen: Hiermit wird die Gültigkeit einer Gruppe von Invarianten geprüft. Der Synchronisations-Manager muß diese Funktionalität anbieten, da die Überprüfung von Invarianten eine unabdingbare Voraussetzung für die Änderung von Objekten durch den zuständigen Resource-Manager ist. Die Überprüfung von Invarianten wird also immer durch einen Resource-Manager angestoßen. Dem Synchronisations-Manager wird in diesem Fall die Information übergeben, welches Objekt geändert werden soll.

Ende_eines_ConTracts: Wird ein ConTract beendet, so muß der Synchronisations-Manager seine Daten an die neue Situation anpassen. Dies bedeutet z.B., daß Invarianten, die vom beendeten ConTracts stammen, gelöscht werden können.

Step-Server

Step-Server sind deshalb in die Synchronisationsverwaltung involviert, da sie im Gegensatz zum Synchronisation-Manager die für bestimmte Objekte zuständigen Resource-Manager kennen. Jeder Step-Server muß diese Information zumindest für diejenigen Objekte kennen, die durch von ihm verwaltete Steps verändert werden.

Objektwert_bestimmen: Jeder Step-Server muß es anderen Komponenten ermöglichen, die Werte ihm bekannter Objekte zu erfahren.

Resource-Manager

Jeder Resource-Manager übernimmt zwei Aufgaben bei der Synchronisationsverwaltung. Er wird zum einen aufgerufen, sobald der Wert eines Objekts geändert werden soll. Zum anderen wird er in Anspruch genommen, wenn eine andere Komponente, z.B. der Synchronisations-Manager, den aktuellen Wert eines Objekts benötigt.

Objektwert_ändern: Dem Resource-Manager muß die Änderungsabsicht für ein bestimmtes Objekt angezeigt werden können. Er muß dann dafür sorgen, daß ein Objektwert nur geändert wird, wenn dies keiner der aktuellen Invarianten widerspricht. Hierzu kann er die Dienste des Synchronisations-Managers in Anspruch nehmen.

Objektwert_bestimmen: Jeder Resource-Manager muß es anderen Komponenten ermöglichen, die Werte ihm bekannter Objekte zu erfahren.

3.2.1.2 Interaktion der Komponenten

Invariante einführen

Nachdem der ConTracts-Manager eine neue Invariante aus dem Skript gelesen hat, übergibt er diese zusammen mit der Information, zu welchem ConTract die Invariante gehört, an den Synchronisations-Manager. Dieser beauftragt die zuständigen Step-Server, die Werte der in der Invariante enthaltenen globalen Objekte zu liefern. Diese Werte müssen von den Step-Servern bei den für die einzelnen Objekte zuständigen Resource-Managern erfragt werden.

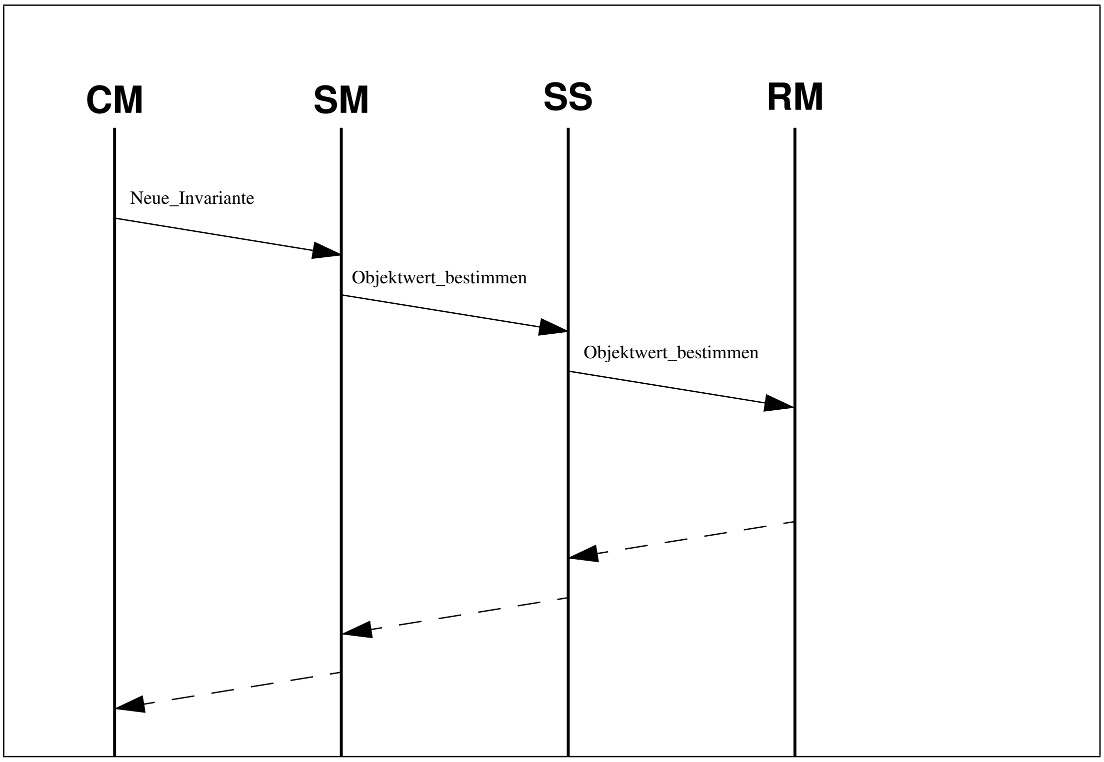

Abb. 3: Globale Synchronisationsverwaltung, Neue Invariante

Die aufgerufenen Resource-Manager liefern die angeforderten Werte an die jeweiligen Step-Server, die sie wiederum an den Synchronisations-Manager weiterleiten. Dieser kann nun die neue Invariante evaluieren und feststellen, ob sie zum aktuellen Zeitpunkt gültig ist. Nur dann kann dem ConTract-

Manager mitgeteilt werden, daß die Invariante zugelassen wird. Der gesamte Ablauf ist in Abbildung 3 dargestellt.

An dieser Stelle einige Kommentare zu Abbildung 3 bis Abbildung 10. Es sind jeweils entlang einer Zeitachse (von oben nach unten) die Aufrufreihenfolgen für einzelne Aufgaben der Synchronisationsverwaltung angegeben. Durchgezogene Pfeile stellen den Aufruf dar. Ihre Beschriftung zeigt an, welche Aufgabe die aufgerufene Komponente ausführen soll. Die jeweiligen Antworten sind mit gestrichelten Pfeilen dargestellt. Bei der später folgenden Betrachtung zur Kommunikation wird lediglich die Aufrufrichtung gezählt. Die Abkürzungen an der Spitze der Zeitachsen stehen für die einzelnen Komponenten. CM steht für ConTract-Manager, SM für Synchronisations-Manager, SS für Step-Server und RM schließlich für Resource-Manager.

Invariante löschen

Die Anweisung zum Löschen einer Invariante liest der ConTract-Manager ebenfalls aus dem Skript. Er übergibt die zu löschende Invariante an den Synchronisations-Manager zur weiteren Bearbeitung. Siehe hierzu Abbildung 4.

Abb. 4: Globale Synchronisationsverwaltung, Invariante löschen

Objektwert ändern

Änderungen globaler Objekte werden direkt bei den zuständigen Resource-Managern angestoßen. Dies kann einerseits durch einen Step-Server erfolgen. Andererseits haben auch Anwendungen, die nicht innerhalb eines ConTracts ablaufen, Zugriff auf globale Objekte. Bevor der Resource-Manager die Änderung akzeptieren kann muß er den Synchronisations-Manager auffordern, die Invarianten für das Objekt zu überprüfen. Dieser kann dies erst dann tun, wenn er zu allen Objekten, die in den fraglichen Invarianten auftauchen die aktuellen Werte kennt. Für jedes Objekt wird also der zuständige Step-Server beauftragt, den Wert zu liefern. Dieser wiederum nimmt hierzu die Hilfe der zuständigen Resource-Manager in Anspruch. Sobald von hier alle benötigten Werte beim Synchronisations-

Manager eingegangen sind, kann dieser die Invarianten auswerten, und so feststellen, ob sie auch mit dem neuen Wert des zu ändernden globalen Objekts erfüllt sein werden. Nur dann kann dem Resource-Manager mitgeteilt werden, daß die Änderung zulässig ist. Der Ablauf ist in Abbildung 5 dargestellt.

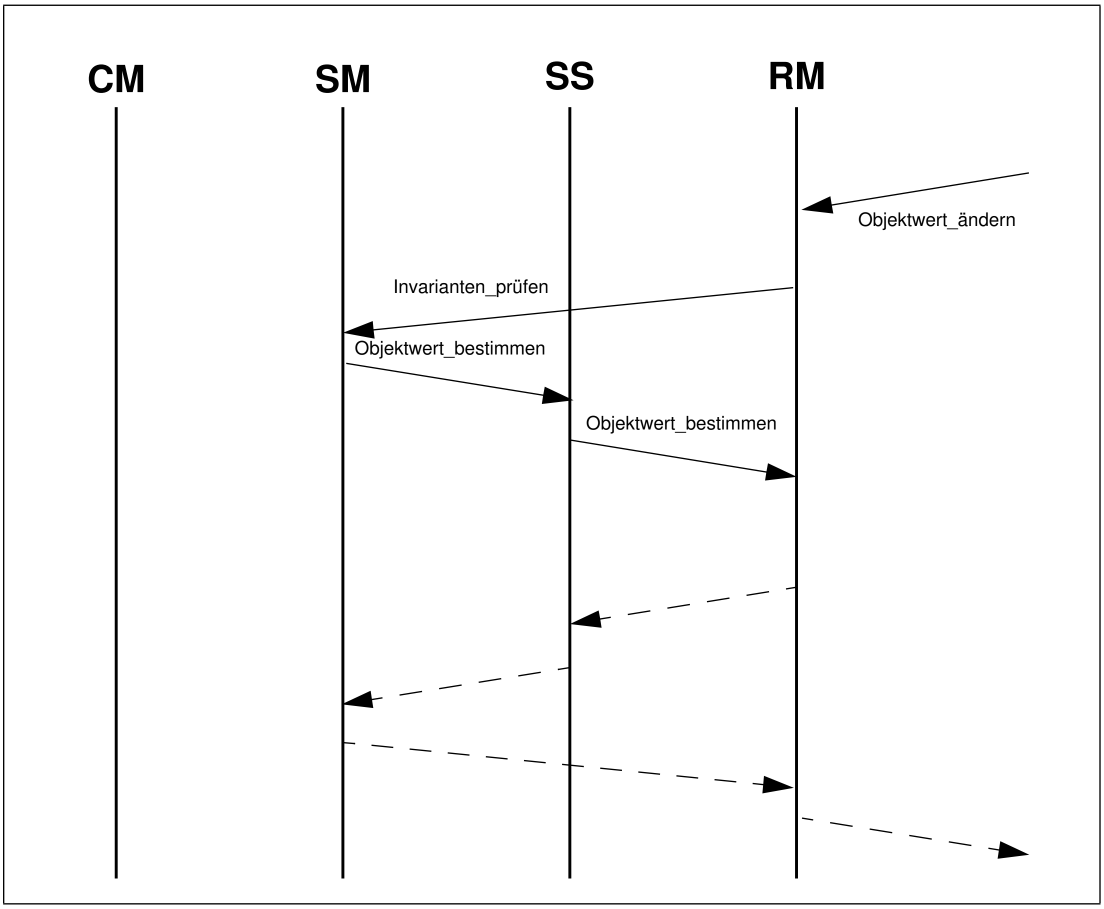

Abb. 5: Globale Synchronisationsverwaltung, Objektwert ändern

ConTract beenden

Sobald der ConTract-Manager aus dem Skript die Information abliest, daß das entsprechende ConTract beendet ist, ruft er den Synchronisations-Manager auf und übergibt diesem die ID des ConTracts. Der

Synchronisations-Manager kann nun alle Invarianten des beendeten ConTracts löschen. Siehe hierzu Abbildung 6.

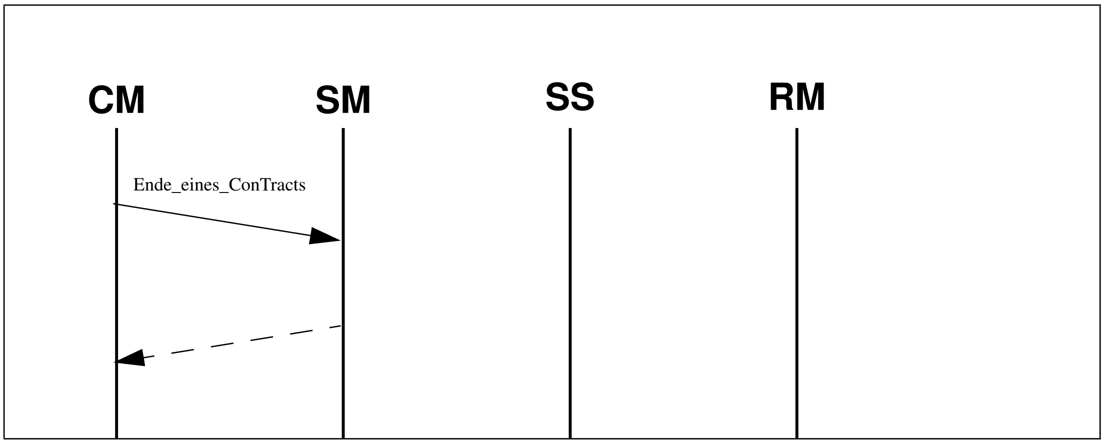

Abb. 6: Globale Synchronisationsverwaltung, Ende eines ConTracts

3.2.2 Synchronisationsverwaltung im Resource-Manager

Diese Architekturvariante zeichnet sich dadurch aus, daß die Invarianten in den einzelnen Resource-Managern verwaltet und evaluiert werden. Hierbei ist jeder Resource-Manager für all diejenigen Invarianten zuständig, die mindestens ein globales Objekt enthalten, das er verwaltet. Für die meisten Objekte kommt nur eine sehr eingeschränkte Anzahl verschiedener Invarianten in Frage. Es müssen also nur solche Mechanismen zur Verwaltung und Evaluierung in einem Resource-Manager implementiert werden, die bei den lokal vorhandenen Objekten notwendig sind.

3.2.2.1 Aufgaben der Komponenten

Auch für die Synchronisationsverwaltung in den Resource-Managern zunächst eine kurze Übersicht über die Funktionen, die die verschiedenen Komponenten im Rahmen der Synchronisationsverwaltung anbieten müssen.

ConTract-Manager

Der ConTract-Manager liest neue Invarianten, zu löschende Invarianten oder die Information über das Ende eines ConTracts aus dem Skript. Zu jeder Anweisung ruft er den entsprechenden Dienst der zuständigen Step-Server auf.

Step-Server

Neue_Invariante: Jeder Step-Server muß Invarianten entgegennehmen können und diese an die zuständigen Resource-Manager zur Speicherung weiterleiten.

Invariante_löschen: Auch Invarianten, die gelöscht werden sollen, werden zunächst von Step-Servern entgegengenommen und an die zuständigen Resource-Manager verteilt.

Ende_eines_ConTracts: Beim Ende eines ConTracts muß der Step-Server alle betroffenen Resource-Manager hiervon unterrichten.

Resource-Manager

Neue_Invariante: Jeder Resource-Manager muß neue Invarianten entgegennehmen können und diese speichern.

Invariante_löschen: Einmal gespeicherte Invarianten müssen durch einen Resource-Manager auch wieder gelöscht werden können.

Ende_eines_ConTracts: Beim Ende eines ConTracts muß der Resource-Manager alle Invarianten dieses ConTracts löschen.

Objektwert_ändern: Einem Resource-Manager muß mitgeteilt werden können, daß sich der Wert eines Objekts ändern soll. Er muß dann die vor der Änderung des Objekts notwendigen Prüfungen durchführen.

Objektwert_bestimmen: Hiermit kann von einem Resource-Manager der aktuelle Wert eines Objekts erfragt werden.

3.2.2.2 Interaktion der Komponenten

Invariante einführen

Nachdem der ConTract-Manager eine neue Invariante aus dem Skript gelesen hat übergibt er diese zusammen mit der Information, aus welchem ConTract sie stammt an die zuständigen Step-Server.

Diese wiederum geben die einzelnen Invarianten weiter an die jeweils zuständigen Resource-Manager. Der Ablauf ist in Abbildung 7 dargestellt.

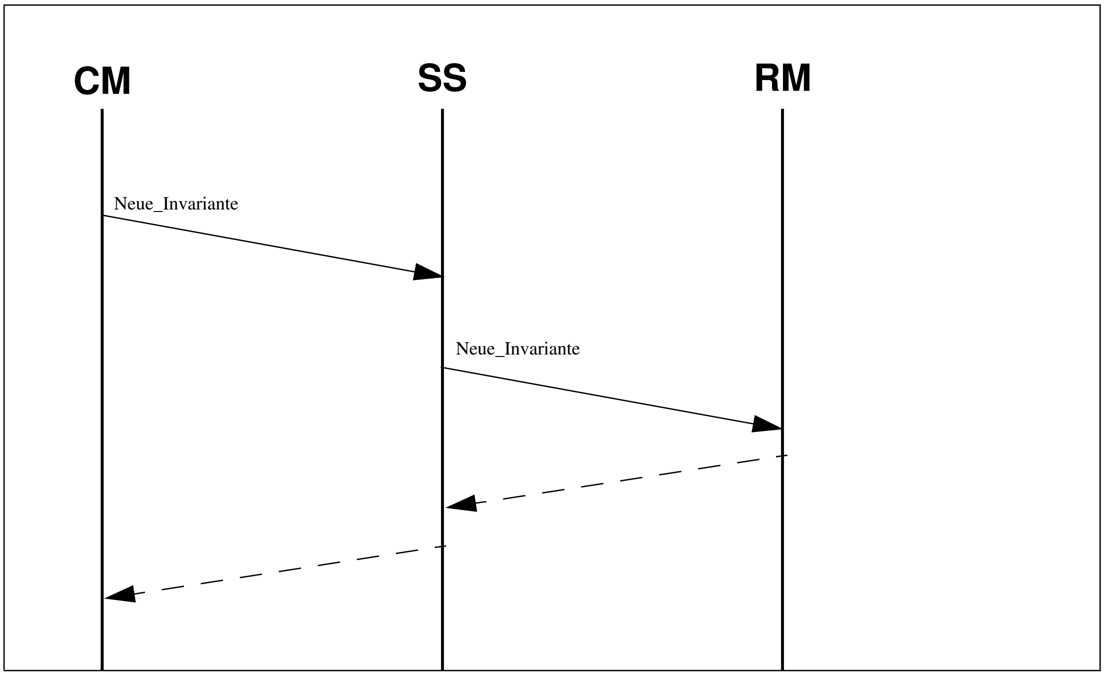

Abb. 7: Synchronisationsverwaltung im Resource-Manager, Neue Invariante

Invariante löschen

Liest der ConTract-Manager die Anweisung aus einem Skript, daß eine bestimmte Invariante zu löschen ist, so gibt er diese Anweisung an die zuständigen Step-Server weiter. Diese rufen ihrerseits alle zuständigen Resource-Manager auf und weisen diesen an die Invariante für den ConTract zu löschen. Der Ablauf wird in Abbildung 8 verdeutlicht.

Objektwert ändern

Soll ein globales Objekt geändert werden, so müssen verschiedene Fälle unterschieden werden. Gibt es für diese Objekt nur Invarianten, die nur Objekte von einem Resource-Manager enthalten, so kann die Evaluierung der Invarianten von diesem einen Resource-Manager vorgenommen werden. Sind in den Invarianten für das zu löschende Objekt noch Objekte anderer Resource-Manager enthalten, so sind diese an der Evaluierung zu beteiligen. Das heißt im Standardfall, daß von diesen der aktuelle Wert der Objekte erfragt werden muß, um die Evaluierung zu ermöglichen. In Abbildung 8 ist dies für die Invariante A+C>x dargestellt. Hierbei wird angenommen, daß A und C von verschiedenen Resource-Managern verwaltet werden.

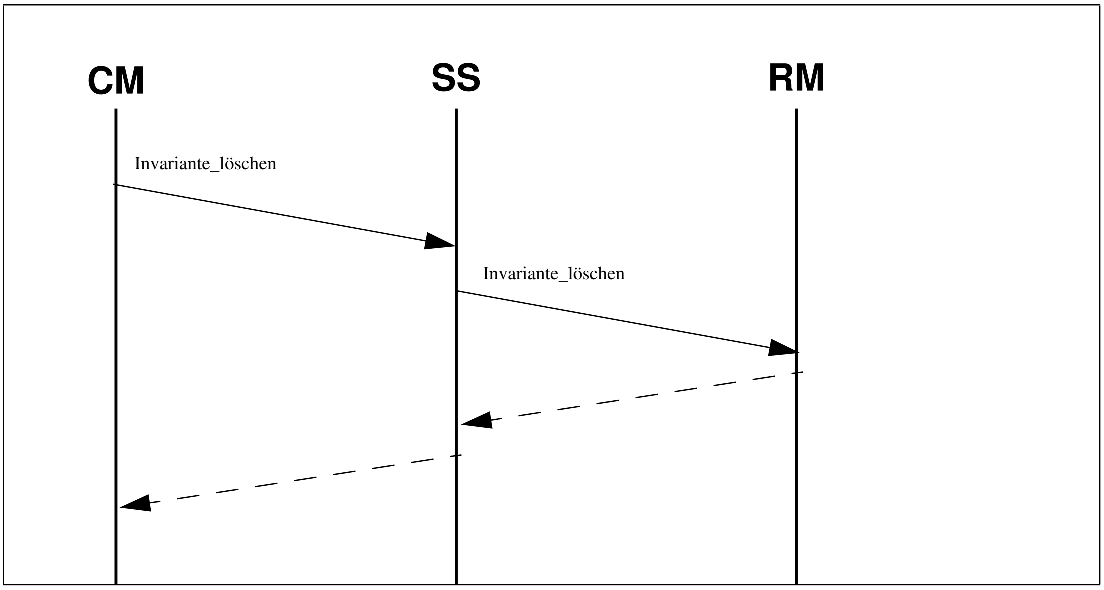

Abb. 8: Synchronisationsverwaltung im Resource-Manager, Invariante löschen

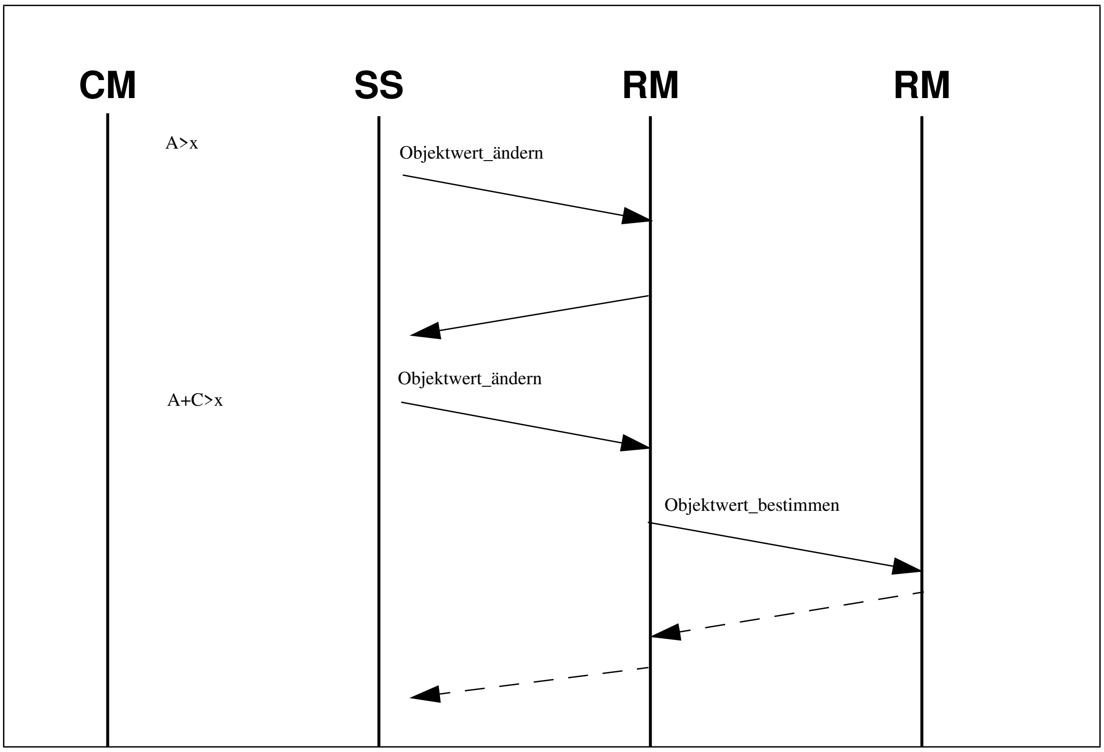

Abb. 9: Synchronisationsverwaltung im Resource-Manager, Objektwert ändern

ConTract beenden

Der Ablauf für das Ende eines ConTracts ist in Abbildung 10 dargestellt. Der ConTract-Manager gibt die Information über das Ende des ConTract über die Step-Server an alle zuständigen Resource-Manager weiter.

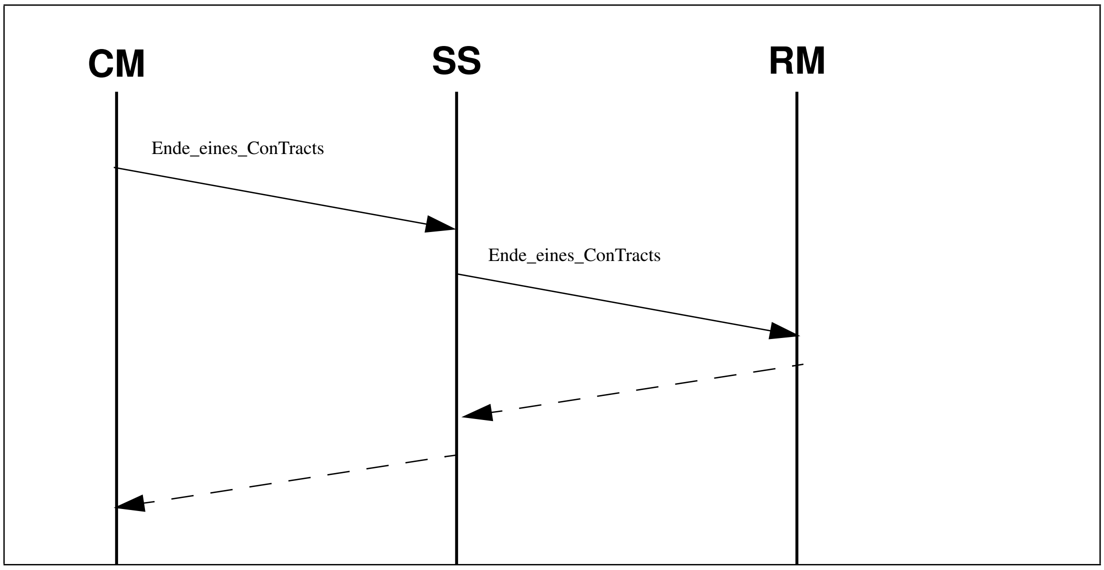

Abb. 10: Synchronisationsverwaltung im Resource-Manager, Ende eines ConTracts

3.2.3 Synchronisationsverwaltung beim Objekt

Bei dieser Variante der Architektur für die Synchronisationsverwaltung in APRICOTS werden die Invarianten und zugehörige Verwaltungsinformation direkt bei den Objekten gespeichert. Für diesen Ansatz muß näher betrachtet werden, um welche Art eines Objekts es sich handelt.

Objekte, die durch ein DBMS verwaltet werden: Hier greift der Resource-Manager mit Hilfe des DBMS auf die Daten zu. Das DBMS bietet keine Schnittstelle an, die es ermöglichen würde Invarianten zu übergeben. Dies hat zur Folge, daß die Synchronisationsverwaltung wiederum vollständig vom Resource-Manager abgewickelt werden muß. Für diesen Fall ergibt sich dann kein Unterschied zum Ansatz mit der Synchronisationsverwaltung im Resource-Manager, wie er in Abschnitt 3.2.2 beschrieben ist.

Objekte im Sinne abstrakte Datentypen: Hier kann für ein Objekt bzw. eine Objektklasse eine zusätzliche Schnittstelle angeboten werden, die für ein bestimmtes Objekt die aktuellen Invarianten entgegennimmt, überprüft oder diese löscht. In diesem Fall kann die Speicherung der Invarianten also direkt beim Objekt erfolgen. Für die weiteren Betrachtungen dieses Abschnitts wird diese Art von Objekten zugrunde gelegt.

3.2.3.1 Aufgaben der Komponenten

ConTract-Manager

Seine Aufgabe entspricht in dieser Architektur exakt der für die Synchronisationsverwaltung in den Resource-Managern beschriebenen.

Step-Server

Auch hier sind die Aufgaben mit den in Abschnitt 3.2.2.1 beschriebenen identisch.

Resource-Manager

Neue_Invariante: Jeder Resource-Manager muß neue Invarianten entgegennehmen können und diese dem betroffenen Objekt zur Speicherung übergeben.

Invariante_löschen: Einmal gespeicherte Invarianten müssen durch einen Resource-Manager beim betroffenen Objekt auch wieder gelöscht werden können.

Ende_eines_ConTracts: Beim Ende eines ConTracts muß der Resource-Manager alle Invarianten dieses ConTracts löschen.

Objektwert_ändern: Einem Resource-Manager muß mitgeteilt werden können, daß sich der Wert eines Objekts ändern soll. Er übergibt dann diesen Änderungswunsch direkt dem Objekt.

Objektwert_bestimmen: Hiermit kann von einem Resource-Manager der aktuelle Wert eines Objekts erfragt werden.

Objekt

Neue_Invariante: Diese Operation eines Objekts nimmt eine neue Invariante für das Objekt entgegen.

Invariante_löschen: Diese Operation eines Objekts löscht eine seiner Invarianten.

Objektwert_ändern: Diese Operation ändert den Wert eines Objekts. Die Operation wird nur dann ausgeführt, wenn die Invarianten, die über das Objekt definiert sind, nicht verletzt werden.

3.2.3.2 Interaktion der Komponenten

Bei dieser Architekturvariante entsprechen die notwendigen Interaktionen weitgehend denen, die bei der Synchronisationsverwaltung im Resource-Manager notwendig sind. Durch das Anbinden der Invarianten an die einzelnen Objekte muß lediglich der Ablauf innehalb der Resource-Manager näher spezifiziert werden.

Invariante einführen

Der ConTract-Manager übergibt eine aus dem Skript gelesene Invarianten den zuständigen Resource-Managern. Diese übergeben sie dem jeweils zuständigen Resource-Manager, der sie an das betroffene Objekt bindet. Hierzu ist gegenüber der in Abbildung 7 dargestellten Interaktion keine zusätzliche Kommunikation zwischen APRICOTS-Komponenten notwendig.

Invariante löschen

Auch hier ergibt sich gegenüber der Darstellung aus Abbildung 8 kein zusätzlicher Kommunikationsaufwand. Wie bei der Einführung einer neuen Invariante verändern sich lediglich die Abläufe innerhalb der Resource-Manager.

Objektwert ändern

Der Wunsch, einen Objektwert zu ändern, wird wie bei den anderen Architekturvarianten direkt dem Resource-Manager angezeigt. Dieser leitet den Änderungswunsch an das betroffene Objekt weiter, das die Änderung nach Überprüfung der etablierten Invarianten ausführt oder ablehnt. Die notwendigen Interaktionen entsprechen also der Darstellung von Abbildung 9. Sobald es Invarianten gibt, die mehrere Objekte betreffen und eventuell auch auf verschiedene Resource-Manager verteilt sind, ist es nicht mehr ausreichend, nur die bei einem Objekt gespeicherten Prädikate zu überprüfen. Es ist wie in Abbildung 9 unten dargestellt notwendig, mit anderen Resource-Managern zu kommunizieren.

ConTract beenden

Wird ein ConTract beendet, so gelangt diese Mitteilung über die Step-Server zu den betroffenen Resource-Managern. Diese müssen dann dafür sorgen daß bei allen Objekten Invarianten des beendeten ConTracts gelöscht werden. Der Ablauf entspricht der Darstellung aus Abbildung 10.

3.3 Kriterien

Bevor die in Abschnitt 3.2 beschriebenen Varianten der Synchronisationsverwaltung verglichen werden können, ist es notwendig die Kriterien für diesen Vergleich festzulegen. Es werden einmal Aufwandskriterien betrachtet. Hierzu gehört eine Abschätzung der notwendigen Kommunikation zwischen APRICOTS-Komponenten und die Betrachtung des Speicheraufwands. Hinzu kommt die Beurteilung der Robustheit der verschiedenen Varianten. Als weitere Kriterien wird geklärt werden müssen, ob und mit welchem Aufwand die verschiedenen Invariantenklassen in der einen oder anderen Architektur verarbeitet werden können, ob und mit welchem Aufwand Optimierungen durchgeführt werden können und in wieweit das Programmiermodell, das den ConTracts zugrunde liegt, unterstützt wird. Diese Kriterien werden in den folgenden Abschnitten näher erläutert und ihre Bedeutung dargelegt, bevor in Abschnitt 3.4 der eigentliche Vergleich erfolgt.

3.3.1 Kommunikationsaufwand

An der Abarbeitung eines ConTracts in APRICOTS sind mehrere Komponenten des Prototyps beteiligt, von denen jede eine Reihe von Diensten anbietet. Für die Integration der Synchronisationsverwaltung müssen je nach gewählter Architektur in die verschiedenen Komponenten zusätzliche Dienste integriert werden. Für die Erledigung einer speziellen Aufgabe der Synchronisationsverwaltung, wie z.B. das Etablieren einer neuen Invariante, ergibt sich dadurch eine spezifische Aufrufreihenfolge für diese neuen Dienste. Diese Reihenfolge und damit die Anzahl der Dienstaufrufe pro Aufgabe der Synchronisationsverwaltung, hängt von der jeweils betrachteten Architekturvariante ab. Jeder Dienstaufruf stellt einen potentiellen Austausch von Nachrichten über das die APRICOTS-Komponenten verbindende Netz dar. Dies resultiert aus der Tatsache, daß die einzelnen Komponenten von APRICOTS auf verschiedene Rechnerknoten verteilt werden können. Jede Nachricht verzögert aber den Ablauf eines ConTracts. Bei einer effizienzorientierten Betrachtung sollte also die Zahl der auszutauschenden Nachrichten pro Aufgabe der Synchronisationsverwaltung minimiert werden.

Darüber hinaus muß auch die Zahl der Nachrichten betrachtet werden, die je Zeiteinheit zwischen APRICOTS-Komponenten ausgetauscht werden müssen. Diese Betrachtung ist separat notwendig, da nicht alle Aufgaben der Synchronisationsverwaltung mit gleicher Häufigkeit in einem ConTract auftreten. Das Ziel muß es sein, die Belastung des Netzes mit Nachrichten, die durch die Synchronisationsverwaltung bedingt sind, so gering wie möglich zu halten.

3.3.2 Speicheraufwand

Für die Synchronisationsverwaltung muß eine Reihe von Verwaltungsinformationen bereitgestellt werden, die weit über das bloße Speichern der Invarianten hinausgehen. Genaueres hierzu wird später in Abschnitt 4.3 beschrieben. Diese Informationen werden abhängig von der jeweiligen Architekturvariante in unterschiedlichen Komponenten gehalten. Für die Bewertung der Architekturen ist es nun notwendig zu betrachten, wie die Informationen jeweils auf die Komponenten verteilt sind, ob Informationen repliziert vorliegen, ob der notwendige Speicherplatz für die Informationen insgesamt variiert und ob die Wahl der Datenstrukturen durch die Architektur beeinflußt wird.

3.3.3 Robustheit

Jede an der Synchronisationsverwaltung beteiligte Komponente kann ausfallen. Sei es, daß der Prozeß der Komponente abgebrochen wird oder aber der Rechnerknoten, auf dem der Komponentenprozeß abläuft, wird heruntergefahren. Abhängig von der gewählten Architektur beeinträchtigt der Ausfall einer Komponente die Leistung des Gesamtsystems unterschiedlich stark. Ziel beim Entwurf der bisherigen Architektur von APRICOTS war es unter anderem, diese so zu gestalten, daß der Ausfall einer einzelnen Komponente toleriert werden kann. Das System muß in einem solchen Fall dafür sorgen, daß die Komponente auf demselben oder einem anderen Rechnerknoten erneut gestartet wird. Die vorgeschlagenen Architekturvarianten sollen also auch unter dem Aspekt beurteilt werden, in wieweit sie dieses allgemeine Architekturziel unterstützen oder diesem entgegenstehen.

3.3.4 Invariantenklassen

In [Schw95] wird eine Grammatik angegeben, die sinnvolle Invarianten beschreibt. Nach dieser Grammatik ist eine breite Palette von Ausdrücken unterschiedlicher Komplexität möglich. Nicht alle Invarianten können in den verschiedenen Architekturen gleich verarbeitet werden. Es ist prinzipiell möglich, daß die gewählte Architektur zu einer Einschränkung der möglichen Invarianten führt. Selbst wenn dies nicht der Fall ist kann sich der für die Verarbeitung notwendige Aufwand bei derselben Invariante je nach Architektur unterschiedlich darstellen. Die Betrachtung darüber, welche Invarianten in welcher Architektur mit welchem Aufwand verarbeitet werden können stellt also ein weiteres Kriterium dar.

Um diese Untersuchung besser zu strukturieren wird im folgenden eine Klassifizierung der möglichen Invarianten durchgeführt. Der Vergleich der Architekturvarianten kann sich dann auf die Betrachtung von Invariantenklassen konzentrieren.

3.3.4.1 Einführung der Klassen

Für die Klasseneinteilung müssen zunächst die Merkmale herausgearbeitet werden, in denen sich die Invarianten unterscheiden. Die Merkmale werden so gewählt, daß Invarianten, die sich in einem Merkmal unterscheiden, einer separaten Behandlung durch die Synchronisationsverwaltung bedürfen. Vor der Auflistung der Invariantenklassen hier die Liste der Unterscheidungsmerkmale, mit einer kurzen Anmerkung zu ihrer Relevanz:

- Besteht die Invariante aus einem oder aus mehreren Prädikaten. Für die Verarbeitung bedeuten mehrere Prädikate einen wesentlich höheren Aufwand bei der Auswertung, da hier mehrere Bedingungen überprüft werden müssen.

- Enthält die Invariante mehrere Prädikate, so ist es von Bedeutung, durch welche logischen Operatoren diese verknüpft sind. Unterschieden wird hier die Verknüpfung rein durch UND (&&), die durch ODER (||), sowie das gemischte Auftreten beider Operatoren. Diese Unterscheidung ist sinnvoll, da z.B. eine Invariante, die aus UND-verknüpften Prädikaten besteht nicht immer als Ganzes bearbeitet werden muß. Durch eine Zerlegung kann sich hier eine einfachere Verarbeitung ergeben. Siehe hierzu auch Abschnitt 4.2.

- Macht eine Invariante Aussagen zu einem oder zu mehreren Objekten. Dieses Merkmal ist interessant, da beim Auftreten mehrerer Objekte diese verknüpft sein können bzw. für ihre

Verwaltung unterschiedliche Resource-Manager zuständig sein können. Beides erhöht die Komplexität der Verarbeitung.

- Ein weiteres Unterscheidungsmerkmal bei Invarianten mit mehreren Objekten ist die Frage, ob alle Objekte von einem Resource-Manager verwaltet werden oder nicht. Sind mehrere zuständig, so ergibt sich zumindest für die Auswertung einer Invariante ein höhere Kommunikationsaufwand. Schließlich muß der Wert aller Objekte abgefragt werden.

- Treten mehrere Objekte in einer Invarianten auf, so können diese direkt verknüpft sein. Hier kommen sowohl arithmetische Operatoren, als auch Vergleichsoperatoren in Frage. Auch dies kann zu einer erhöhten Verarbeitungskomplexität führen.

Die aufgeführten Unterscheidungsmerkmale führen zu insgesamt 16 Invariantenklassen. Diese sind in Tabelle 1 mit ihren jeweiligen Merkmalsausprägungen aufgeführt. Abbildung 11 stellt alle Invariantenklassen mit jeweils einem Beispiel dar. Die Beispielinvarianten könnten in der in Abschnitt 2.2.5 dargestellten Beispielumgebung auftreten. Genutzt werden die vier Objekte VR, HR, WS und HS, wobei VR und HR, sowie WS und HS jeweils von einem Resource-Manager verwaltet werden.

Klasse	**Zahl der Prädikate**	**Prädikatverknüpfung**	**Zahl der Objekte**	**Objektverknüpfung**	**zuständige Resource-Manager**
1	1	-	1	-	1
2	> 1	&&	1	-	1
3	> 1	||	1	-	1
4	> 1	&&, ||	1	-	1
5	1	-	> 1	arithmetisch	1
6	1	-	> 1	Vergleich	1
7	1	-	> 1	arithmetisch, Vergleich	1
8	> 1	&&	> 1	-	1
9	> 1	||	> 1	-	1
10	> 1	&&, ||	> 1	-	1
11	1	-	> 1	arithmetisch	> 1

Tab. 1: Invariantenklassen

Klasse	Zahl der Prädikate	Prädikatverknüpfung	Zahl der Objekte	Objektverknüpfung	zuständige Resource-Manager
12	1	-	> 1	Vergleich	> 1
13	1	-	> 1	arithmetisch, Vergleich	> 1
14	> 1	&&	> 1	-	> 1
15	> 1	||	> 1	-	> 1
16	> 1	&&, ||	> 1	-	> 1

Tab. 1: Invariantenklassen

Mit 16 Klassen ergibt sich eine sehr feine Einteilung der Invarianten. Dies ist sowohl für den Vergleich der Architekturvarianten (siehe Abschnitt 3.4) als auch für die konkrete Implementierung (siehe Kapitel 4) hilfreich.

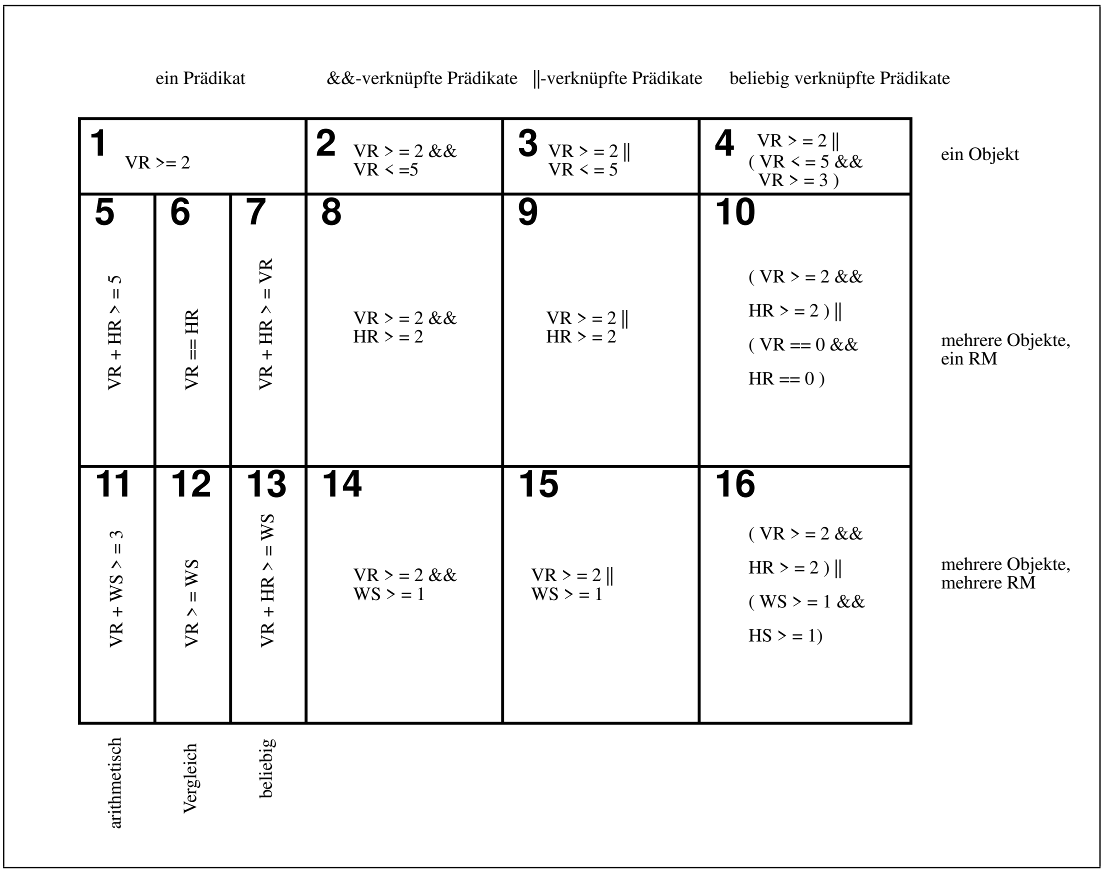

Abb. 11: Invariantenklassen

3.3.5 Programmiermodell

ConTracts liegt ein spezielles Programmiermodell zugrunde. Jeder ConTract wird durch ein Skript beschrieben. Dieses gibt unter anderem den Kontrollfluß zwischen einzelnen Steps an. Der Programmierer des Skripts kennt lediglich die Funktionalität der vorhandenen Steps, die er zum Aufbau des gesamten ConTracts verwendet. Aus dem Skript heraus sind keinerlei Zugriffe auf globale Objekte möglich. Diese werden nur innerhalb der verschiedenen Steps ausgeführt. Die Zugriffsmöglichkeiten auf Objekte sind damit nur für die Programmierer der Steps von Interesse. Es ergibt sich ein zweistufiges Programmiermodell, wie in Abbildung 12 dargestellt. Die Stufe der Programmierung von Resource-Managern wird hier nicht zur eigentlichen ConTract-Programmierung hinzugerechnet.

Dieses Programmiermodell soll sich auch in den Kommunikationsbeziehungen zwischen den Laufzeitkomponenten wiederspiegeln. Der ConTract-Manager ist für die Abarbeitung des Skripts zuständig. Hierzu nimmt er die Dienste von Step-Servern in Anspruch. Diese sorgen für die Ausführung einzelner Steps. Die Dienste der Resource-Manager werden von den Steps für Zugriffe auf globale Objekte genutzt. Das Programmiermodell hat also beispielsweise zur Folge, daß der ConTract-Manager nicht direkt auf einen Resource-Manager zugreifen kann.

Auch nach der Integration der Synchronisationsverwaltung sollten sich keine anderen Kommunikationsbeziehungen als die in Abbildung 12 dargestellten ergeben. Nur so wird das Programmiermodell von ConTracts nicht unterlaufen. Die vorgeschlagenen Architekturen sind daraufhin zu untersuchen.

3.3.6 Optimierungsmöglichkeiten

Es gibt verschiedene Möglichkeiten der Optimierung. In [Schw95] sind Möglichkeiten dargestellt, wie mehrere Invarianten zusammengefaßt werden können. Dies ist die einzige Art der Optimierungen, die für sämtliche Architekturvarianten betrachtet werden soll. Andere Möglichkeiten, wie z.B. die Vermeidung von Kommunikation zwischen den Komponenten, ergeben sich oft erst durch eine bestimmte Architektur und sind in anderen Architekturen gar nicht notwendig. Sie werden darum zum Vergleich nicht herangezogen. In Abschnitt 4.4 werden solche Arten der Optimierung für die gewählte Architekturvariante allerdings genauer betrachtet.

Einer der Vorteile von Invariantenzusammenfassungen ist, daß vor einer Objektänderung eine kleinere Anzahl Invarianten überprüft werden muß. So führen diese Optimierungen in vielen Fällen zu einer beschleunigten Abarbeitung eines ConTracts. Es muß darum für jede Architekturvariante betrachtet werden, welche der Optimierungen bei ihr möglich sind und welche nicht. Bei prinzipiell möglichen Optimierungen ist zu berücksichtigen, ob der erforderliche Aufwand durch entsprechende Vorteile gerechtfertigt ist.

Die in [Schw95] beschriebenen Optimierungsmöglichkeiten betreffen die in dieser Arbeit einge-

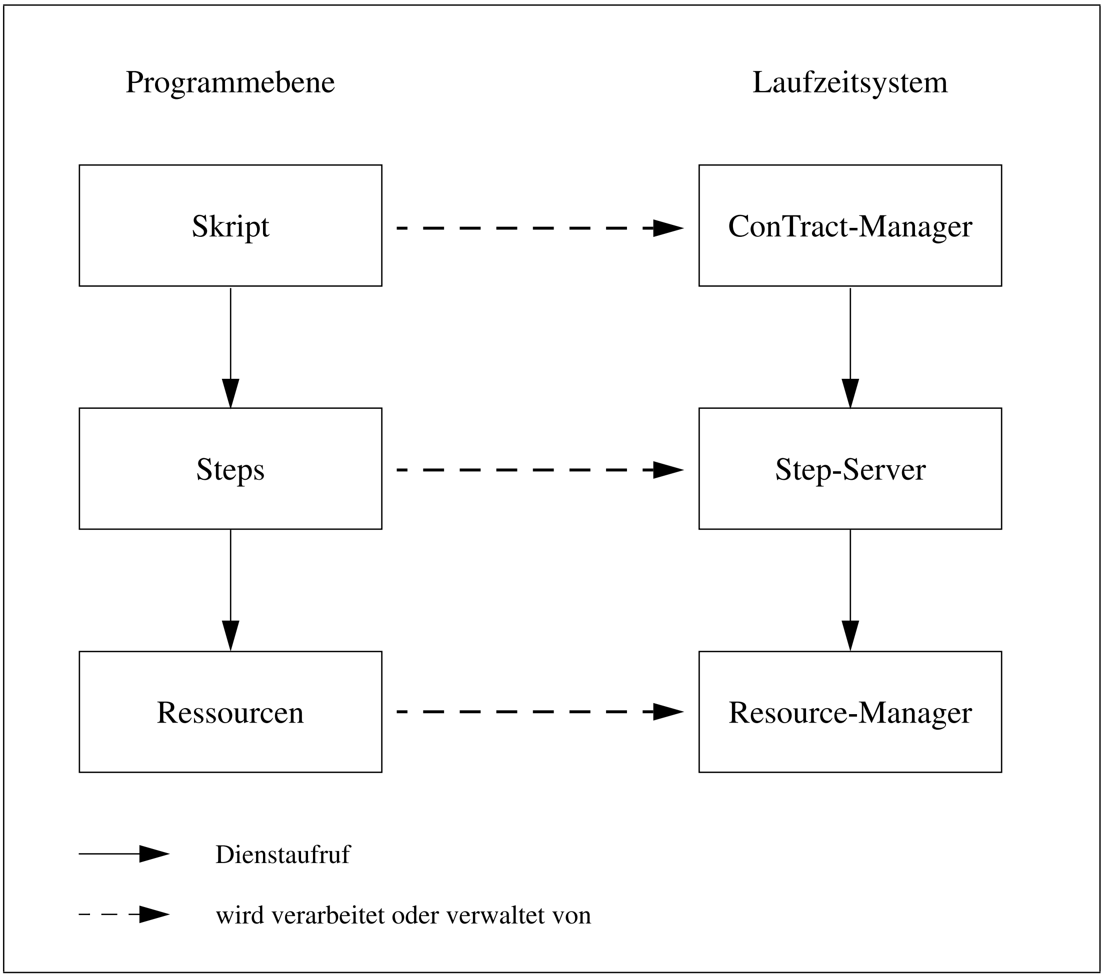

Abb. 12: Programmiermodell von APRICOTS

führte Invariantenklasse 1. Es handelt sich hiebei um Möglichkeiten, einfache Invarianten, wie z.B. WS >= 1 und WS >= 2 zu WS >= 3 zusammenzufassen. Da diesen einfachen Invarianten in der Praxis die größte Bedeutung beizumessen ist, sollte ihre Optimierung in jeder Architekturvariante möglich sein. Darüber hinausgehende Optimierungen sind nur sinnvoll, wenn die Vorteile den jeweils notwendigen Aufwand rechtfertigen. Diese Untersuchung soll unter diesem Stichwort in Abschnitt 3.4.6 erfolgen.

3.4 Vergleich

In diesem Kapitel sollen nun die Architekturvarianten anhand der in Abschnitt 3.3 dargestellten Kriterien verglichen werden. Hierbei werden die Kriterien zunächst separat an jeder Architekturvariante untersucht. Danach folgt eine abschließende Beurteilung, die zeigen soll, welche Architektur sich am besten für eine Implementierung eignet.

3.4.1 Kommunikationsaufwand

Bei der Beurteilung des Kommunikationsaufwands sollen wie in Abschnitt 3.3.1 beschrieben zwei Bereiche getrennt untersucht werden:

- Die Anzahl der Nachrichten, die während der Bearbeitung eines ConTracts für Aufgaben der Synchronisationsverwaltung zwischen verschiedenen APRICOTS-Komponenten ausgetauscht werden.

- Die Zahl der Nachrichten, die pro Komponente während einer Zeiteinheit verarbeitet werden muß. Auch hier werden nur die Nachrichten betrachtet, die für die Synchronisation notwendig sind.

Im folgenden wird lediglich ein Vergleich zwischen zwei der drei Architekturvarianten durchgeführt. Dies ist ausreichend, da die Synchronisationsverwaltung in den Resource-Managern und die Synchronisationsverwaltung bei den Objekten denselben Kommunikationsaufwand zeigen. Dies wurde bereits in Abschnitt 3.2.3.2 gezeigt. Der Vergleich wird mit Hilfe konkreter Zahlenwerte durchgeführt. Diese konnten keinem konkreten ConTract entnommen werden, sondern es wurde versucht, möglichst realistische Werte zu wählen.

Nachrichten pro ConTract

Die Anzahl auszutauschender Nachrichten pro ConTract kann mit Hilfe folgender Angaben berechnet werden:

- Zahl der Nachrichten, die für die Erledigung einer speziellen Aufgabe der Synchronisationsverwaltung ausgetauscht werden müssen. Diese Angaben können aus den in Abschnitt 3.2 aufgeführten Abbildungen entnommen werden. Die konkreten Zahlenwerte sind in Tabelle 2 zusammengestellt.

- Häufigkeit, mit der eine Aufgabe der Synchronisationsverwaltung innerhalb eines ConTracts ausgeführt werden muß. Diese Angaben werden in Tabelle 3 zusammengestellt. Die Zahlen sind so gewählt, da das Ende eines ConTracts nur einmal auftritt und die Änderung von

Aufgaben	zentrale Synchronisationsverwaltung	Synchronisationsverwaltung in den Resource-Managern
Neue Invariante	6	4
Objektänderung	8	2,5[a]
Invariante löschen	2	4
Ende eines ConTracts	2	4

Tab. 2:Nachrichten pro Aufgabe

a. 2 Nachrichten sind notwendig, wenn die Invariante nur Objekte eines Resource-Managers enthalten. Sind zwei Resource-Manager zuständig, so ergeben sich 4 Nachrichten. Andere Fälle sollen hier unberücksichtigt bleiben. 2,5 stellt den Durchschnittswert dar.

Objekten ein wesentlich häufigeres Ereignis ist, als das Etablieren oder Löschen von Invarianten.

Aufgabe	Anzahl pro ConTract
Neue Invariante	10
Objektänderung	50
Invariante löschen	5
Ende des ConTracts	1

Tab. 3:Anzahl der Aufgaben pro ConTract

Multipliziert man die Angaben aus Tabelle 2 und Tabelle 3 getrennt für jede einzelne Aufgabe und addiert anschließend die Ergebnisse, so ergeben sich die Zahlen vonTabelle 4. Als Fazit bleibt festzuhalten, daß bei der globalen Architektur für jeden ConTract der doppelte Kommunikationsaufwand entsteht wie bei den dezentralen Varianten.

	zentrale Synchronisationsverwaltung	Synchronisationsverwaltung in den Resource-Managern
Anzahl der Nachrichten pro ConTract	472	209

Tab. 4: Anzahl der Nachrichten pro ConTract

Nachrichten pro Komponente

Bei dem Versuch, zu ermitteln, wieviele Nachrichten eine APRICOTS-Komponente pro Zeiteinheit verarbeiten muß, sind zusätzlich noch folgende Angaben von Interesse:

- Zahl der ConTracts, die durchschnittlich pro Sekunde vollständig durchgeführt werden. Hier soll von 5 ConTracts ausgegangen werden. Damit ergeben sich die in Tabelle 5 dargestellten Häufigkeiten für die Ausführung einzelner Aufgaben der Synchronisationsverwaltung.

Aufgaben	**Häufigkeiten / Sekunde**
Neue Invariante	50
Objektänderung	250
Invariante löschen	25
Ende eines ConTracts	5

Tab. 5:Häufigkeit der Aufgaben pro Sekunde

- Aus den Abbildungen in Abschnitt 3.2 ergibt sich, wieviele Nachrichten die einzelnen Komponenten für die Ausführung einer Aufgabe der Synchronisationsverwaltung empfangen müssen. Diese Angaben sind inTabelle 7 zusammengefaßt.

	globale Synchronisationsverwaltung				**Synchronisationsverwaltung in den Resource-Managern**		
	CM	**SM**	**SS**	**RM**	**CM**	**SS**	**RM**
Neue Invariante	1	2	2	1	1	2	1
Objektänderung	0	2	2	3	0	0	1-2[a]
Invariante löschen	1	1	0	0	1	2	1
Ende eines ConTracts	1	1	0	0	1	2	1

Tab. 6:Nachrichtenhäufigkeit pro Model

a. 1 gilt, wenn nur ein Resource-Manager beteiligt ist. 2 Nachrichten sind notwendig, wenn zwei Resource-Manager beteiligt sind.

- Zusätzlich muß berücksichtigt werden, daß die Komponenten unterschiedlich oft in einem System vorhanden sind. Hier soll von je einem ConTract-Manager und Synchronisations-

Manager ausgegangen werden. Von Step-Servern und Resource-Managern werden jeweils 5 Exemplare vorausgesetzt.

Mit Hilfe dieser Angaben kann nun die Nachrichtendichte pro Komponente und Sekunde berechnet werden. Hierzu werden die Angaben aus Tabelle 5 und Tabelle 7 für jede einzelne Aufgabe multipliziert und anschließend für jede Komponente separat addiert. Damit ergeben sich die in Tabelle 7 in der grau unterlegten Zeile dargestellten Nachrichten, die jede Komponente pro Sekunde verarbeiten muß. Diese Angaben müssen noch durch die Anzahl der Exemplare der einzelnen Komponententypen dividiert werden, um die tatsächliche Nachrichtendichte pro Komponente zu erhalten. Das Ergebnis zeigt eindeutig, daß der Synchronisations-Manager bei der globalen Synchronisationsverwaltung wesentlich größere Datenmengen pro Zeiteinheit verarbeiten muß, als alle anderen Komponenten. Dies bedeutet, daß diese Komponente für das Gesamtsystem sehr schnell einen Flaschenhals darstellt.

	globale Synchronisationsverwaltung				**Synchronisationsverwaltung in den Resource-Managern**		
	CM	**SM**	**SS**	**RM**	**CM**	**SS**	**RM**
Neue Invariante	50	100	100	50	50	100	50
Objektänderung	0	500	500	750	0	0	313[a]
Invariante löschen	25	25	0	0	25	50	25
Ende eines ConTracts	5	5	0	0	5	10	5
Nachrichten je Sekunde	80	630	600	800	80	160	393
Nachrichten je Sekunde und Exemplar[b]	80	630	120	160	80	32	79

Tab. 7:Nachrichtenhäufigkeit pro Modul und Zeiteinheit

a. Hier wird angenommen, daß bei 100 Änderungen 74 mit einem Resource-Manager durchgeführt werden. Für die restlichen 25 Änderungen wird angenommen, daß 2 Resource-Manager involviert sind. Dies ist dann der Fall, wenn die Objekte einer Invariante von verschiedenen Resource-Managern verwaltet werden.

b. Es wird angenommen, das im System ein ConTract-Manager und ein Synchronisations-Manager, sowie jeweils fünf Resource-Manager und Step-Server arbeiten.

Fazit: Die Betrachtung des Kommunikationsaufwands zeigt klare Vorteile für die dezentralen Architekturvarianten. Ein Vorteil ist die geringere synchronisationsspezifische Nachrichtenzahl, die für einen einzelnen ConTract notwendig ist. Als zweiten Vorteil ergibt sich gegenüber der globalen Variante keine unverhältnismäßig hohe Belastung einer einzelnen Komponente.

3.4.2 Speicheraufwand

Unter dem Aspekt des Speicheraufwands sei hier zunächst die globale Synchronisationsverwaltung betrachtet. Hier werden sämtliche Informationen zentral im Synchronisations-Manager verwaltet. Es ergibt sich aus der Architektur keine Notwendigkeit einen Teil der Information repliziert zu halten. Auch die Wahl der Datenstrukturen kann bei der Implementierung völlig frei erfolgen, da alle Informationen in einer Komponente verwaltet werden.

Anders sieht die Situation bei den übrigen Architekturvarianten aus. Hier wird die Verwaltungsinformation verteilt gespeichert. Die Verteilung der Invarianten wird durch die ConTract-Manager und die Step-Server durchgeführt. Diese müssen hierzu über zusätzliche Verwaltungsinformationen verfügen. Bei Invarianten, die mehrere Objekte beinhalten, ist es meist notwendig, die ganze Invariante in verschiedenen Resource-Managern zu speichern. Dies gilt zunächst für die Synchronisationsverwaltung in den Resource-Managern. Bei der Synchronisationsverwaltung beim Objekt wird davon ausgegangen, daß jedes Objekt nur Prädikate zu sich selbst speichert. Bei zusammengesetzten Invarianten, wie z.B. WS >= 1 || HS >= 1 muß die Information, daß die beiden Prädikate WS >= 1 und HS >= 1 zusammengehören separat verwaltet werden. Dies kann ebenfalls zu replizierter Information führen.

Fazit: Betrachtet man den notwendigen Speicheraufwand, so ergeben sich leichte Vorteile für die globale Synchronisationsverwaltung. Da hier alle Verwaltungsinformationen zentral gespeichert werden, kann die Informationsmenge auf das notwendige Minimum komprimiert werden. Es gibt nicht wie bei den anderen beiden Architekturvarianten eine Replikation der Daten.

3.4.3 Robustheit

Bei der globalen Synchronisationsverwaltung besteht der folgenreichste Komponentenausfall darin, daß der zentrale Synchronisations-Manager ausfällt. Dies verhindert sehr schnell die weitere Bearbeitung aller ConTracts. Da die vier zentralen Aufgaben der Synchronisationsverwaltung (Invarianten etablieren und löschen, Objekte verändern und ConTracts beenden) alle nur unter Beteiligung des Synchronisations-Managers ausgeführt werden können, diese Dienste von jedem ConTracts aber ständig in Anspruch genommen werden, wird mit einem Ausfall des Synchronisations-Managers die Bearbeitung aller ConTracts praktisch sofort eingestellt. Ihre Fortsetzung ist erst nach dem Restart des Synchronisations-Managers wieder möglich.

Nun zur Betrachtung der beiden dezentralen Architekturvarianten. Hier sind bei jeder Aufgabe der Synchronisationsverwaltung mehrere Komponenten beteiligt. Die Folgen ihres Ausfalls sollen darum getrennt untersucht werden.

Der Ausfall eines Step-Servers hat bei der Synchronisationsverwaltung im Resource-Manager und bei der Synchronisationsverwaltung beim Objekt dieselben Auswirkungen. Zunächst können die vom betroffenen Step-Server verwalteten Steps nicht mehr ausgeführt werden. Hierbei handelt es sich aber um eine generelle Folge und nicht um ein synchronisationsspezifisches Problem. Hinzu kommt, daß der Step-Server nun auch für die Weiterleitung von Invarianten über ihm bekannte Objekte zu den zuständigen Resource-Managern nicht mehr zur Verfügung steht. Alle ConTracts, die für Aufgaben der Synchronisationsverwaltung den ausgefallenen Step-Server in Anspruch nehmen wollen, werden verzögert. Meist wird es sich aber um ConTracts handeln, die auch ohne Berücksichtigung der Synchronisation durch den Ausfall des Step-Servers verzögert würden, da bereits ihr Aufruf eines Steps nicht erfolgreich ist. Dies resultiert aus der Tatsache, daß der Aufruf eines Steps auf einem Step-Server meist bedeutet, daß für die zugehörigen Synchronisationsaufgaben derselbe Step-Server benötigt wird. Als Ergebnis kann festgehalten werden, daß bei der Berücksichtigung der Synchronisation sich für den Ausfall eines Step-Servers kaum zusätzliche Auswirkungen ergeben.

Fehlt noch die Betrachtung dessen, was aus dem Ausfall eines Resource-Managers resultiert. Wie beim Ausfall eines Step-Servers ist hier nur ein Teil der ConTracts betroffen. Zugriffe auf die Objekte oder Veränderungen der Invarianten zu ihnen sind nicht mehr möglich. Zumindest für die Objektzugriffe gilt dies aber auch ohne Berücksichtigung der Synchronisation.

Fazit: Das Kriterium Robustheit führt zu klaren Vorteilen der dezentralen Architekturen. Die Tatsache, daß bei globaler Synchronisationsverwaltung der Ausfall des Synchronisations-Managers unverzüglich alle ConTracts stoppt, ist kaum akzeptabel. Bei den dezentralen Varianten hingegen zeigen Ausfälle der Step-Server oder Resource-Manager kaum größere Auswirkungen als dies ohne Synchronisationsverwaltung schon der Fall wäre.

3.4.4 Invariantenklassen

Betrachtet man zunächst die verschiedenen Invariantenklassen, wie sie in Abbildung 11 dargestellt sind, so zeigt sich, daß die Verarbeitung sämtlicher Klassen bei der globalen Synchronisationsverwaltung möglich ist. Durch die zentrale Speicherung der vollständigen Invarianten spielt es bei der Verarbeitung keine Rolle, ob eine Invariante aus einem oder mehreren Prädikaten zusammengesetzt ist, ob ein oder mehrere Objekte enthalten sind, von welchen Resource-Managern diese verwaltet werden und wie sie verknüpft sind. Alle Aufgaben der Synchronisationsverwaltung laufen immer prinzipiell gleich ab.

Anders stellt sich die Situation bei der Synchronisationsverwaltung beim Objekt dar. Geht man davon aus, daß die Invarianten nur bei den einzelnen Objekten gespeichert sind, so folgt hieraus, daß nur die Invariantenklassen 1 bis 4 ohne zusätzliche Mechanismen verwaltet werden können. Für alle anderen Invariantenklassen ist es bei der Verwaltung und Verarbeitung notwendig, weitere Objekte bzw. Resource-Manager zu berücksichtigen. Dies führt zu verschiedenen Sonderbehandlungen, die eventuell mit einem erhöhten Kommunikationsaufwand zwischen APRICOTS-Komponenten einhergehen.

Eine Mittelposition nimmt bei der Betrachtung der Invariantenklassen die Synchronisationsverwaltung in den Resource-Managern ein. Hier können die Klassen 1 bis 10 ohne weitere Sonderbehandlung verarbeitet werden. Bei allen anderen Klassen sind für die Verarbeitung die Beteiligung mehrerer Resource-Manager notwendig.

Um abzuschätzen, wie stark der Nachteil dieser Sonderbehandlungen bei der Synchronisationsverwaltung in den Resource-Managern gegenüber der globalen Synchronisationsverwaltung wiegt, muß zusätzlich die Häufigkeit beachtet werden, mit der die gesondert zu behandelnden Invariantenklassen auftreten. Obwohl es sich bei den fraglichen Klassen 11 bis 16 um ein Drittel der Invariantenklassen handelt gibt es Grund zu der Annahme, daß der Anteil an real auftretenden Invarianten weit unter einem Drittel liegt. Gründe für diese Annahme sind:

- Die häufigste Anwendung von Invarianten dürfte es sein, sicherzustellen, daß ein einzelnes Objekt einen bestimmten Wertebereich nicht verläßt. Hierzu sind die Invariantenklassen 1 und 2 ausreichend.

- Werden Invarianten zu mehreren Objekten formuliert, so ist es sehr wahrscheinlich, daß alle Objekte in denselben Kontext gehören und damit meist auch von demselben Resource-Manager verwaltet werden. Diese Invarianten fallen damit in die Klassen 5 bis 10.

Die Invarianten der Klassen 11-16 können also eher als seltene Ausnahmen betrachtet werden. Daraus folgt, daß der erhöhte Aufwand bei der Verarbeitung solcher Invarianten bei der Synchronisationsverwaltung im Resource-Manager die Leistung des Gesamtsystems nicht sonderlich stark beeinträchtigt. Im Unterschied hierzu ist bei der Synchronisationsverwaltung beim Objekt eine Sonderbehandlung auch für relativ häufig auftretende Invariantenklassen (Klassen 5 bis 10) notwendig.

Fazit: In allen vorgeschlagenen Architekturvarianten lassen sich sämtliche Invariantenklassen verarbeiten. Der Unterschied besteht lediglich darin, welche Arten von Invarianten einer gesonderten Behandlung bedürfen. Bei der globalen Synchronisationsverwaltung ist dies bei keiner der Klassen notwendig. Bei der Synchronisationsverwaltung in den Resource-

Managern ergibt sich die Notwendigkeit für eine kleine Gruppe selten auftretender Invarianten. Diese Architektur ist hier also nur geringfügig schlechter zu bewerten, als die globale Version. Weitaus mehr Invarianten sind bei der Synchronisationsverwaltung beim Objekt von einer Sonderbehandlung betroffen. Dieser zusätzliche Aufwand schlägt bei dieser Architekturvariante negativ zu Buche.

3.4.5 Programmiermodell

Hier soll nun betrachtet werden, ob sich die drei Architekturvarianten an das Programmiermodell von ConTracts halten. Hierzu müssen die Kommunikationsbeziehungen, wie sie bei der Ausführung von Synchronisationsaufgaben auftreten näher untersucht werden. Dies kann am besten mit Hilfe der Abbildungen aus Abschnitt 3.2 erfolgen. Hier zeigt sich, daß in allen Varianten folgende Bedingungen erfüllt sind:

- Der ConTract-Manager ruft nur Funktionen anderer Komponenten auf und wird im Zusammenhang mit der Synchronisationsverwaltung selbst nicht aufgerufen.
- Die Step-Server werden nur vom ConTract-Manager (in der globalen Variante vom Synchronisations-Manager) aufgerufen.
- Die Resource-Manager werden nur von Step-Servern oder anderen Resource-Managern aufgerufen.

Fazit: Alle drei Architekturvarianten halten sich an das ConTract-Programmiermodell. Sie sind also bezüglich dieses Kriteriums als gleichwertig anzusehen.

3.4.6 Optimierungsmöglichkeiten

Als letztes Kriterium folgt hier die genauere Betrachtung der Optimierungsmöglichkeiten, deren Bedeutung bereits in Abschnitt 3.3.6 erläutert wurde. Für alle Architekturansätze gilt, daß nur solche Optimierungen von Invarianten berücksichtigt werden sollten, die innerhalb einer Komponente durchgeführt werden können. Die Kommunikation verschiedener Komponenten mit dem Ziel Optimierungsmöglichkeiten zu finden, stellt sicherlich einen Aufwand dar, der auch durch die Vorteile bei der Evaluierung von Invarianten nicht gerechtfertigt ist. Dies soll anhand der Abbildung 7 erläutert werden. Sie stellt den Ablauf bei der Etablierung einer neuen Invariante dar, wenn die Synchronisationsverwaltung in den Resource-Managern erfolgt. Während dieser Aufgabe können bereits Optimierungen durchgeführt werden. Allein die Einbeziehung eines zusätzlichen Resource-Managers würde den Kommunikationsaufwand um 50 Prozent erhöhen. Sind noch weitere Komponenten an der Optimierung beteiligt, sorgt dies zu einer erheblichen Verzögerung bis eine Invariante etabliert ist.

Es sollten also nur Optimierungen betrachtet werden, die innerhalb einer Komponente ausgeführt werden können. Dies ist notwendigerweise immer die Komponente, der die Verwaltung der Invarianten obliegt. Für die einzelnen Architekturvarianten also der Synchronisations-Manager, die Resource-Manager oder die Objekte selbst. Für die globale Synchronisationsverwaltung ergibt sich sofort, daß die zentrale Verwaltung der Invarianten Optimierungen für sämtliche Invariantenklassen möglich macht. Bei der Synchronisationsverwaltung in den Resource-Managern sind Optimierungen nur für die Invariantenklassen 1 bis 10 und bei der Synchronisationsverwaltung beim Objekt nur für die Klassen 1 bis 4 möglich.

Fazit: Die wichtigsten Optimierungen für Invarianten der Klassen 1 bis 4 sind in allen Architekturvarianten möglich. Die volle Flexibilität bei der Optimierung besteht allerdings nur bei der globalen Variante. Bei den dezentralen Architekturen gibt es jeweils eine Reihe von Invariantenklassen, für die sich der Optimierungsaufwand nicht rechtfertigen läßt. Hier besteht bei der Synchronisationsverwaltung in den Resource-Managern eine etwas größere Flexibilität, als dies bei der Synchronisationsverwaltung beim Objekt der Fall ist.

3.4.7 Abschließende Bewertung

Nachdem die Kriterien Kommunikations- und Speicheraufwand, Robustheit, Invariantenklassen, Programmiermodell und Optimierungsmöglichkeiten nun für jede der Architekturvarianten genauer betrachtet wurden, kann die abschließende Bewertung erfolgen. Die einzelnen Bewertungen sind inTabelle 8 zusammengestellt. Die Einträge dort haben folgende Bedeutungen:

+ bedeutet, daß die Architekturvariante das Kriterium optimal erfüllt.
0 bedeutet, daß die Architekturvariante das Kriterium ausreichend erfüllt.
- bedeutet, daß die Architekturvariante das Kriterium kaum erfüllt.

Nach dieser Tabelle stellt die Synchronisationsverwaltung durch die Resource-Manager die geeignetste der betrachteten Architekturvarianten dar. Wichtige Gründe hierfür sind:

- Da es sich um eine dezentrale Architektur handelt, ist sie, was die notwendige Kommunikation zwischen Komponenten und die Robustheit anbetrifft, der globalen Variante klar über-

	globale Synchronisationsverwaltung	Synchronisationsverwaltung in den Resource-Managern	Synchronisationsverwaltung beim Objekt
Kommunikationsaufwand	-	+	+
Speicheraufwand	+	0	0
Robustheit	-	+	+
Invariantenklassen	+	0	-
Programmiermodell	+	+	+
Optimierungsmöglichkeiten	+	0	-

Tab. 8: Gesamtbewertung der Architekturvarianten

legen. Dies ist für die Gesamtbewertung besonders wichtig, da es sich hier um die wichtigsten Kriterien handelt.

- Die Synchronisationsverwaltung durch die Resource-Manager hat gegenüber der Synchronisationsverwaltung im Objekt den Vorteil, daß viele Invariantenklassen ohne zusätzliche Mechanismen verarbeitet und optimiert werden können.

- Die Architektur erreicht als einzige bei allen Kriterien zumindest durchschnittliche Werte.

Die aufgeführten Argumente führen schließlich dazu, daß in der weiteren Arbeit nur noch die Synchronisationsverwaltung in den Resource-Managern betrachtet wird. Zunächst müssen hierzu weitere Details geklärt werden, bevor dann auf die erfolgte Implementierung eines Prototyps näher eingegangen wird.

Kapitel 4

Details der gewählten Variante

Dieses Kapitel beschäftigt sich mit Details der Synchronisationsverwaltung in den Resource-Managern, die über die kurze Beschreibung aus Abschnitt 3.2.2 hinausgeht. Hiermit sollen die Grundlagen für die prototypische Implementierung, wie sie in Kapitel 4 beschrieben wird, gelegt werden. Zunächst werden die Aufgaben, die die verschiedenen Komponenten im Rahmen der Synchronisationsverwaltung haben, näher beschrieben. Hierauf werden spezielle Mechanismen für einzelne Invariantenklassen eingeführt. Schließlich wird die, für die einzelnen Komponenten notwendige Verwaltungsinformation zusammengestellt.

4.1 Die Komponenten und ihre Aufgaben

In Abschnitt 3.2.2 wurde lediglich skizziert, welche Aufgaben die einzelnen Komponenten bei der Synchronisationsverwaltung zu erfüllen haben. Hier sollen nun diese Aufgaben weiter untergliedert und im Detail betrachtet werden. Eine Zuordnung der einzelnen Aufgaben zu den von den Komponenten angebotenen Diensten erfolgt dann erst im nächsten Kapitel bei der Beschreibung der Implementierung. Im folgenden werden Step-Server und Resource-Manager separat betrachtet. Für jede der Komponenten wird weiter nach auszuführenden Aufgaben unterschieden.

4.1.1 Step-Server

Neue Invariante

Der Step-Server nimmt vom ConTract-Manager eine neue Invariante entgegen. Dies Invariante muß nun durchsucht werden, um festzustellen, welche Objekte in ihr enthalten sind. Für diese Objekte muß der zuständige Resource-Manager ermittelt werden. Dies ist natürlich nur für diejenigen Objekte möglich, die dem Step-Server auch bekannt sind. Die Invariante wird nun dem ermittelten Resource-

Manager zur weiteren Verarbeitung übergeben. Der Step-Server hat hier also lediglich eine Art Verteilerfunktion. Hat der Step-Server die Invariante an den Resource-Manager weitergegeben, so kann er eine erfolgreiche Erledigung erst dann an den ConTract-Manager zurückmelden, wenn vom Resource-Manager bestätigt wurde, daß die Invariante etabliert werden konnte.

In Abschnitt 4.2 werden spezielle Mechanismen für bestimmte Invariantenklassen beschrieben. Diese führen teilweise zu einer anderen Verarbeitung der Invarianten. Genaueres hierzu soll erst dort erläutert werden.

Invariante löschen

Das Verhalten eines Step-Servers, wenn er vom ConTract-Manager den Auftrag erhält, eine Invariante zu löschen, unterscheidet sich kaum von dem bei der Etablierung einer neuen Invariante. Auch hier muß die Invariante nach den enthaltenen Objekten durchsucht werden. Dem jeweils zuständigen Resource-Manager wird dann die Invariante übergeben. Meldet der Resource-Manager einen erfolgreichen Löschvorgang, so kann die Erfolgsmeldung an den ConTract-Manager weitergegeben werden.

Ende eines ConTracts

Bekommt ein Step-Server von einem ConTracts-Manager das Ende eines ConTracts mitgeteilt, so besteht seine Aufgabe darin, dies an die betroffenen Resource-Manager weiterzuleiten. Hierzu ist es notwendig, daß der Step-Server protokolliert, welche Resource-Manager für welchen ConTract gearbeitet haben. Wurde von allen Resource-Managern als Ergebnis übermittelt, daß die zum ConTract-Ende notwendigen Aufgaben durchgeführt werden konnten, so kann der Step-Server auch dem ConTract-Manager eine erfolgreiche Durchführung melden.

4.1.2 Resource-Manager

Neue Invariante

Ein Resource-Manager erhält neue Invarianten von einem Step-Server. Diese Invariante kann nur akzeptiert werden, falls:

- Sie bei der aktuellen Wertebelegung der Objekte erfüllt ist.
- Sie keinen Widerspruch zu vorhandenen Invarianten darstellt.

Sind diese beiden Bedingungen erfüllt, so kann dem Step-Server die erfolgreiche Etablierung gemeldet werden. Dann können die mit der neuen Invariante möglichen Optimierungen versucht werden. Die Invariante und die daraus hervorgegangenen Zusammenfassungen werden zusammen mit der notwendigen Verwaltungsinformation im Resource-Manager gespeichert.

Invariante löschen

Erhält der Resource-Manager den Auftrag eine Invariante zu löschen, so muß er zusätzlich alle Invarianten löschen, die durch Zusammenfassungen oder sonstige Optimierungen aus dieser Invariante hervorgegangen sind. Dann kann dem Step-Server das erfolgreiche Löschen der Invariante zurückgemeldet werden.

Ende eines ConTracts

Wird dem Resource-Manager das Ende eines ConTracts gemeldet, so müssen alle Invarianten, die für diesen ConTract etabliert wurden, ebenfalls gelöscht werden. Zusätzlich werden auch alle Invarianten ungültig, die als Zusammenfassung aus denjenigen hervorgegangen sind, die nun gelöscht werden müssen.

Objektwert ändern

Die Änderung von Objektwerten wird entweder durch einen Step-Server oder durch andere Anwendungen, die nicht als ConTracts ablaufen, angestoßen. Bevor die Änderung erfolgen kann, muß der Resource-Manager überprüfen, ob der neue Wert eines Objekts nicht zu einer Verletzung einer Invarianten führt. Nur wenn alle Invarianten für das Objekt weiterhin gültig bleiben, wird der neue Wert dem Objekt zugewiesen und dem Auftraggeber die erfolgreiche Ausführung gemeldet.

4.2 Mechanismen für spezielle Invariantenklassen

Der vorangegangene Abschnitt hat beschrieben, welche Aufgaben die an der Synchronisationsverwaltung beteiligten Komponenten zu erfüllen haben, und wie die grundsätzliche Vorgehensweise aussieht. Nun gibt es aber Invariantenklassen, deren Verwaltung nach dieser "einfachen" Methode nicht möglich ist. Es handelt sich hierbei um alldiejenigen Klassen, deren Invarianten sich aus mehreren Objekten zusammensetzen, für deren Verwaltung mindestens zwei verschiedene Resource-Manager zuständig sind. Betroffen sind alle Invarianten der Klassen 11 bis 16 (Vgl. hierzu Abbildung 11). In diesem Abschnitt soll beschrieben werden, mit Hilfe welcher Mechanismen diese trotzdem verarbeitet werden können.

Für einige andere Invariantenklassen werden in diesem Abschnitt ebenfalls vereinfachende Mechanismen vorgestellt, obwohl diese nicht zwingend notwendig sind. Dies ist besonders für solche Invarianten der Fall, die sich aus UND-verknüpften Prädikaten zusammensetzen.

4.2.1 Virtuelle Objekte

Problembeschreibung

Invarianten der Klassen 11 und 13 bestehen aus einem Prädikat und können durch arithmetische Operatoren verknüpfte Objekte enthalten. Von den Klassen 5 und 7 heben sie sich dadurch ab, daß die verknüpften Objekte von verschiedenen Resource-Managern verwaltet werden. Das bedeutet, daß es nicht ausreicht, einem einzigen Resource-Manager die Invariante zu übergeben, wie dies in Abschnitt 4.1.1 beschrieben ist. Hier gibt es unterschiedliche Methoden, wie die Invarianten verarbeitet werden können.

Invariante nicht verändern

Bei dieser Methode wird der einfache Mechanismus möglichst weitgehend beibehalten. Dies bedeutet, daß die vollständige Invariante an alle Resource-Manager weitergegeben wird, die für eines der in der Invariante auftretenden Objekte verantwortlich sind. Die Resource-Manager kennen dann jeweils die vollständige Invariante. Aus der Sicht eines speziellen Resource-Managers enthält eine solche Invariante Objekte, die bei ihm lokal vorhanden sind und solche, die von entfernten Resource-Managern verwaltet werden. Für die Auswertung einer Invariante muß er dann lediglich mit diesen entfernten Resource-Managern Kontakt aufnehmen, um die aktuellen Werte der Objekte zu erfahren. Diese Methode, die Invarianten der Klassen 11 und 13 zu verarbeiten, kann folgendermaßen bewertet werden:

- Es wird so wenig wie möglich an der grundsätzlichen Vorgehensweise verändert.
- Jeder Resource-Manager muß die vollständige Invariante speichern.

- Jedem Resource-Manager muß bei jeder Invariante die Information über entfernte Objekte mitgegeben werden. Dies gilt auch dann, wenn dieselben Objekte, auf dieselbe Art und Weise verknüpft, bereits mehrfach aufgetreten sind.

Virtuelle Objekte

Hier werden die Invarianten vom Step-Server nicht im Original an die Resource-Manager weitergegeben, sondern es findet vorab eine Verarbeitung statt. Hierbei werden Ausdrücke, die mehrere arithmetisch Verknüpfte Objekte enthalten durch einen neuen Objektbezeichner ersetzt. So kann z.B. die Invariante WR + WS >= 3 zu virt_obj_1 >= 3 werden. Bevor die veränderte Invariante an die zuständigen Resource-Manager weitergegeben wird, muß diesen das neue virtuelle Objekt bekannt gemacht werden. Das bedeutet, ihnen muß der Name und der Aufbau des Objekts mitgeteilt werden. Außerdem müssen sie wissen, welche Resource-Manager für die im virtuellen Objekt enthaltenen Einzelobjekte zuständig sind. Das ist notwendig, da jeder Resource-Manager bei der Bestimmung des Wertes eines virtuellen Objekts die für die enthaltenen Objekte zuständigen Resource-Manager nach dem aktuellen Wert befragen muß. Bei der Einführung virtueller Objekte gibt es folgende Vor- und Nachteile:

- Die Invarianten müssen bereits im Step-Server vorverarbeitet werden.

- An die Resource-Manager werden kürzere Invarianten übergeben, als dies ohne Veränderung der Invarianten der Fall ist. Die einfacher aufgebauten Invarianten lassen sich auch einfacher verwalten und auswerten.

- Die Information über den Aufbau virtueller Objekte und die zuständigen Resource-Manager für Teilobjekte müssen an die Resource-Manager nur einmal weitergegeben werden.

- Virtuelle Objekte müssen in den Resource-Managern zusätzlich verwaltet werden.

Bewertung

Die Einführung virtueller Objekte hat gegenüber der einfachen Methode mit unveränderten Invarianten klare Vorteile. Diese liegen einmal in der Vereinfachung der Kommunikation zwischen Step-Servern und Resource-Managern. Diese können einfachere Invarianten austauschen und die Information über einem Resource-Manager nicht lokal bekannte Objekte muß diesem nur einmal mitgeteilt werden. Dann ist das virtuelle Objekt bei ihm etabliert und er kann darauf zugreifen. Die einfacheren Invarianten machen sich auch bei der Speicherung und Verarbeitung in den Resource-Managern positiv bemerkbar. Anstelle vieler Invarianten, die immer denselben komplexen Ausdruck enthalten, können

Invarianten mit einem einzigen virtuellen Objekt verwaltet werden. Der komplexe Ausdruck wird dann nur an einer Stelle, als Aufbau des virtuellen Objekts gespeichert. Im Beispiel heißt dies: In allen Invarianten, die WR + WS enthalten, wird dies durch virt_obj_1 ersetzt. Bereits dies spart jeweils das Speichern eines Operators und eines Objektbezeichners. Bei komplexeren Ausdrücken ist die Ersparnis natürlich erheblich größer.

Die hier aufgeführten Vorteile virtueller Objekte führen dazu, daß diese für die in dieser Arbeit beschriebene prototypische Implementierung vorgesehen werden.

4.2.2 UND-Verknüpfung

Problembeschreibung

Invarianten der Klassen 2,3,4,8,9,10,14,15 und 16 bestehen aus mehreren mit logischen Operatoren verknüpften Prädikaten. Hier sollen davon nur diejenigen betrachtet werden, bei denen es sich auf oberster Ebene um eine UND-Verknüpfung handelt. Für die Verarbeitung dieser Invarianten sollen hier zwei Alternativen verglichen werden.

Invarianten nicht verändern

Hier wird grundsätzlich die komplette Invariante an den oder die zuständigen Resource-Manager übergeben. Für die Invariantenklassen 14 und 16 muß die Weitergabe an mehrere Resource-Manager erfolgen. Jedem Resource-Manager muß dann zusätzlich die Information darüber gegeben werden, welche Resource-Manager für ihm lokal nicht bekannte Objekte zuständig sind. Bei diesen entfernten Resource-Managern muß er sich die aktuellen Werte der in einer Invariante enthaltenen Objekte beschaffen, sobald die Invariante ausgewertet werden soll. Folgende Punkte sind bei der Bewertung dieser Vorgehensweise wichtig:

- Es wird so wenig wie möglich an der grundsätzlichen Vorgehensweise verändert.

- Jeder Resource-Manager muß die vollständige Invariante speichern und auswerten, auch wenn dies für einzelne Teilprädikate durch einen anderen Resource-Manager komplett erledigt werden könnte.

Invarianten zerlegen

Hier werden die Invarianten vor ihrer Weitergabe an den oder die Resource-Manager zerlegt. Dies ist möglich aufgrund der UND-Verknüpfung der Prädikate bei den hier betrachteten Invarianten. Es zeigt sich, daß es gleichwertig zur Weitergabe der vollständigen Invarianten ist, wenn die durch UND verknüpften Prädikate einzeln betrachtet werden. Für jedes Teilprädikat müssen dann der oder die zuständigen Resource-Manager ermittelt werden. Diesen wird nur das Teilprädikat übergeben. Achten alle Resource-Manager gleichermaßen auf die Einhaltung der ihnen übergebenen Teilprädikate, so ergibt sich insgesamt die durch die UND-Verknüpfung beabsichtigte Semantik. Im Beispiel heißt dies: Die Invariante VR >= 2 && WS >= 1 wird in die Prädikate VR >= 2 und WS >= 1 zerlegt. VR >= 2 wird an den für VR zuständigen Resource-Manager übergeben. Ebenso erhält der für WS zuständige Resource-Manager das Prädikat WS >= 1. Hier die Bewertung für diese Vorgehensweise:

- Komplexe Invarianten werden durch einfache ersetzt. Deren Verarbeitung ist für die einzelnen Resource-Manager einfacher.

- Invarianten, die mehrere von verschiedenen Resource-Managern verwalteten Objekte enthalten, können eventuell so aufgeteilt werden, daß jedem Resource-Manager genau ein Teilprädikat übergeben werden kann. Hierdurch benötigen weniger Resource-Manager Informationen über entfernte Objekte.

- Für die Zerlegung ist eine zusätzliche Verarbeitung der Invarianten in den Step-Servern notwendig.

Bewertung

Die Zerlegung der Invarianten in einzelne Prädikate hat klare Vorteile. Die Resource-Manager erhalten einfachere Prädikate. Deren Speicherung und Auswertung gestalten sich ebenfalls einfacher. Außerdem kann die Zerlegung in vielen Fällen so erfolgen, daß jeder Resource-Manager Teilprädikate erhält, die nur bei ihm lokal vorhandene Objekte beinhalten. Hierdurch kann der Kommunikationsaufwand bei der Auswertung von Invarianten gesenkt werden. Diese Gründe führen dazu, daß in der prototypischen Implementierung der Step-Server eine Zerlegung im dargestellten Stil angestrebt wird.

4.3 Notwendige Verwaltungsinformationen

Die in Abschnitt 4.1 beschriebenen Aufgaben der einzelnen Komponenten setzen jeweils das Vorhandensein einer Reihe von Verwaltungsinformationen voraus. Welche das sind soll in diesem Abschnitt näher untersucht werden. Die Resultate dieser Betrachtung bilden eine wichtige Grundlage für die Festlegung der Datenstrukturen. Hier wird zunächst nur festgelegt, welche Daten gespeichert werden müssen, damit die einzelnen Komponenten ihre Aufgaben auch erfüllen können. In Kapitel 5 werden dann die Datenstrukturen für die Implementierung entworfen. Hierbei können noch verschiedentlich Verwaltungsinformationen hinzukommen, deren Notwendigkeit sich an dieser Stelle noch nicht ergibt.

4.3.1 Step-Server

Jeder Step-Server muß folgende Daten verwalten:

- Zu jedem Objekt, das dem Step-Server bekannt ist, muß der zuständige Resource-Manager gespeichert sein. Dies ist notwendig, damit die Invarianten an die richtigen Resource-Manager weitergeleitet werden können.
- Für jedes virtuelle Objekt muß gespeichert sein:
  - Der Bezeichner für das Objekt
  - Der Ausdruck, der das Objekt beschreibt.

    Dies ermöglicht es dem Step-Server in später auftretenden Invarianten bereits etablierte virtuelle Objekte zu identifizieren.
- Für jeden ConTract muß der Step-Server festhalten, welche Resource-Manager Aufgaben der Synchronisationsverwaltung für diesen ConTract übernommen haben. Damit soll das Ende eines ConTracts gezielt allen beteiligten Resource-Managern mitgeteilt werden können.

4.3.2 Resource-Manager

Hier die Liste der in jedem Resource-Manager notwendigen Verwaltungsinformationen:

- Für jede Invariante muß folgendes festgehalten werden:
  - Der Aufbau der Invariante.
  - Der ConTract, zu dem die Invariante gehört. Dies ist beim Ende eines ConTracts wichtig, wenn alle zugehörigen Invarianten gelöscht werden sollen.
  - Handelt es sich um eine Zusammenfassung oder Optimierung?
  - Aus welchen anderen Invarianten ging die Invariante durch Zusammenfassung hervor? Beim Löschen einer Zusammenfassung werden die ursprünglichen Invarianten wieder gültig. Darum ist diese Information wichtig.
  - Für welche Zusammenfassung wurde die Invariante verwendet? Beim Löschen der Invariante werden automatisch alle Zusammenfassungen ungültig, in die sie eingegangen ist. Darum muß diese Information gespeichert werden.
- Für jedes Objekt sind folgende Informationen wichtig:
  - Handelt es sich um ein lokales, ein virtuelles oder ein entferntes Objekt?
  - Bei lokalen Objekten interessiert ihr Wert bei der Auswertung von Invarianten.
  - Bei entfernten Objekten muß der zuständige Resource-Manager festgehalten werden. Dies ist ebenfalls bei der Auswertung der Invarianten wichtig.
  - Bei jedem virtuellen Objekt muß der Ausdruck festgehalten werden, der beschreibt, wie es sich zusammensetzt.

4.4 Optimierungsmöglichkeiten

Die bisherigen Abschnitte dieses Kapitels haben sich mit den Standardabläufen der Synchronisationsverwaltung beschäftigt. Hierbei sind eine Reihe von Optimierungen möglich. Dieser Abschnitt führt einige der Möglichkeiten auf. Hierbei geht es vor allem um eine möglichst effiziente Auswertung von Invarianten. Da diese Aufgabe der Synchronisationsverwaltung bei Objektänderungen sehr häufig ausgeführt werden muß haben Optimierungen an dieser Stelle einen großen Einfluß auf die Performance des Systems.

Zusammenfassung von Invarianten

[Schw95] beschreibt eine Reihe von Möglichkeiten, wie mehrere Invarianten zu demselben Objekt zusammengefaßt werden können. Hieraus ergibt sich als Vorteil, daß die Anzahl der zu überprüfenden Invarianten reduziert werden kann. Einige dieser Möglichkeiten sind in der vorliegenden Implementierung realisiert. Im Gegensatz hierzu sind die folgenden Möglichkeiten der Optimierung hier nur kurz erwähnt und werden in der weiteren Arbeit nicht mehr berücksichtigt. Dies hängt nicht mit deren Bedeutung zusammen, sondern hat seine Ursache lediglich im Zwang, die prototypische Implementierung auf einige wichtige Mechanismen zu beschränken.

Änderung der Auswertungsreihenfolge

Bei Invarianten, die aus mehreren logisch verknüpften Prädikaten bestehen, kann durch eine Veränderung der Auswertungsreihenfolge eine erhebliche Effizienzsteigerung erreicht werden. Der Effekt läßt sich dadurch erreichen, daß das Ergebnis der Invariante bereits nach der Auswertung eines Teils der Prädikate bekannt ist. So ist bei der Invariante `VR > 1 || HR > 1` bereits klar, daß sie erfüllt ist, wenn `HR>1` zum Ergebnis TRUE führt. Eine Änderung der Auswertungsreihenfolge ist in diesem Fall sinnvoll, wenn VR von einem entfernten Resource-Manager verwaltet wird. Dann kann das Ergebnis der gesamten Invariante lokal bestimmt werden, und die Kommunikation mit dem für VR zuständigen Resource-Manager kann entfallen.

Nutzung von Zusatzinformation

Die Auswertung von Invarianten kann eventuell durch die Nutzung von Zusatzinformationen zu Objekten beschleunigt werden. Als Beispiel sei die Invariante `VR + HR >= 2` genannt. Wieder soll `HR` lokal und `VR` entfernt verwaltet werden. Die Zusatzinformation zum entfernten Objekt VR könnte z.B. lauten: `VR >=0`. Dies hat zur Folge, daß lokal nur noch die Invariante `HR >= 2` ausgewertet werden muß. Ist diese bereits erfüllt, so gilt dies in jedem Fall auch für die ganze Invariante. Wieder wird die Kommunikation mit einem entfernten Resource-Manager eingespart.

Kapitel 5

Implementierung

Die bisherige Arbeit hat sich mit der Architektur für die Synchronisationsverwaltung in APRICOTS beschäftigt. Es wurde eine geeignete Variante ausgewählt und näher untersucht. Auf der Basis dieser Architektur soll nun eine prototypische Implementierung beschrieben werden. Diese wurde in C unter UNIX erstellt. Da es den Rahmen dieser Arbeit bei weitem sprengen würde, die volle Funktionalität der Synchronisationsverwaltung zu implementieren, sind folgende generelle Einschränkungen zu beachten:

- Ziel der Arbeit ist es, die Konzepte und Methoden, wie sie in [Schw95] und in dieser Arbeit entwickelt wurden, prototypisch zu implementieren. Besonderes Augenmerk soll hierbei vor allem auf die Abläufe in den Step-Servern und Resource-Managern gerichtet werden. Diese Komponenten sind in der Hauptsache für die Synchronisationsverwaltung nach der gewählten Architekturvariante zuständig. Aufgaben, die andere Komponenten haben, werden im Rahmen dieser Arbeit nicht implementiert.

- Die hier beschriebenen Komponenten werden nicht in den bestehenden Prototyp integriert. Dies soll erst nach dieser Arbeit erfolgen. Ein Grund hierfür ist, daß der ConTract-Manager parallel zu der hier beschriebenen Implementierung überarbeitet wurde, so daß dieser nicht in der aktuellen Version einsetzbar war. Hieraus ergibt sich als Konsequenz auch, daß für die Kommunikation zwischen Step-Servern und Resource-Managern einfache Mechanismen der Interprozeßkommunikation verwendet werden und keine Integration des APRICOTS-Kommunikationssystems erfolgt.

- In [Schw95] wird eine ganze Reihe von Datentypen bei der Synchronisationsverwaltung betrachtet. Hier werden lediglich zwei davon ausgewählt. Als einfacher Datentyp wird Integer gewählt. Arrays von Integerwerten werden exemplarisch für zusammengesetzte Datentypen betrachtet. Diese Einschränkung bezieht sich allerdings nur auf die Funktionalität der Komponenten. Die Schnittstellen und Datenstrukturen sind so gestaltet, daß eine Verwaltung von Invarianten zu Objekten aller in [Schw95] betrachteten Typen möglich ist.

- Die Menge der möglichen Prädikate wird gegenüber [Schw95] ebenfalls eingeschränkt. Hier werden nur Vergleiche und einfache arithmetische Operatoren berücksichtigt. Darüber hinausgehende Operatoren werden zwar in den implementierten Datenstrukturen berücksichtigt, nicht aber bei der Verarbeitung innerhalb der Step-Server und Resource-Manager.

- Objekte, über die innerhalb der prototypischen Implementierung Invarianten formuliert werden können sind zunächst nur innerhalb der Datenstrukturen der Resource-Manager gespeichert. Ein Zugriff auf Datenbankobjekte wird hier nicht vorgesehen.

Alle diese Einschränkungen erlauben es, in dieser Arbeit als wesentlichen Aspekt die Implementierung der internen Verwaltung der Synchronisationsinformationen in Step-Servern und Resource-Managern zu betrachten. Für eine vollständige Implementierung, die in APRICOTS integriert werden soll, müssen die Einschränkungen weitgehend aufgehoben werden.

Die sich anschließende Betrachtung erfolgt getrennt nach Step-Servern und Resource-Managern und den Kommunikationsaspekten zwischen beiden. Für jede Komponente werden die Datenstrukturen, Schnittstellen und gewählte Algorithmen näher betrachtet. Außerdem wird jeweils aufgeführt, welche Ergänzungen für eine vollständige Implementierung notwendig sind.

5.1 Resource-Manager

Die Resource-Manager übernehmen den Hauptteil der Aufgaben für die Synchronisationsverwaltung. Dieser Abschnitt stellt die wichtigsten Aspekte einer Implementierung dar. Zunächst werden die Module, die die Synchronisationsverwaltung implementieren, aufgeführt. Hierauf folgt eine Beschreibung der Schnittstellen für die Dienste, die ein Resource-Manager anbieten muß. Anschließend werden die notwendigen Datenstrukturen und die verwendeten Algorithmen erklärt. Schließlich sollen auch die für eine vollständige Implementierung notwendigen Ergänzungen genannt werden.

5.1.1 Module

Die Synchronisationsverwaltung in einem Resource-Manager setzt sich aus mehreren Modulen zusammen. Deren Aufgaben werden in diesem Abschnitt jeweils kurz skizziert. Abbildung 13 stellt die Abhängigkeiten zwischen den einzelnen Modulen dar.

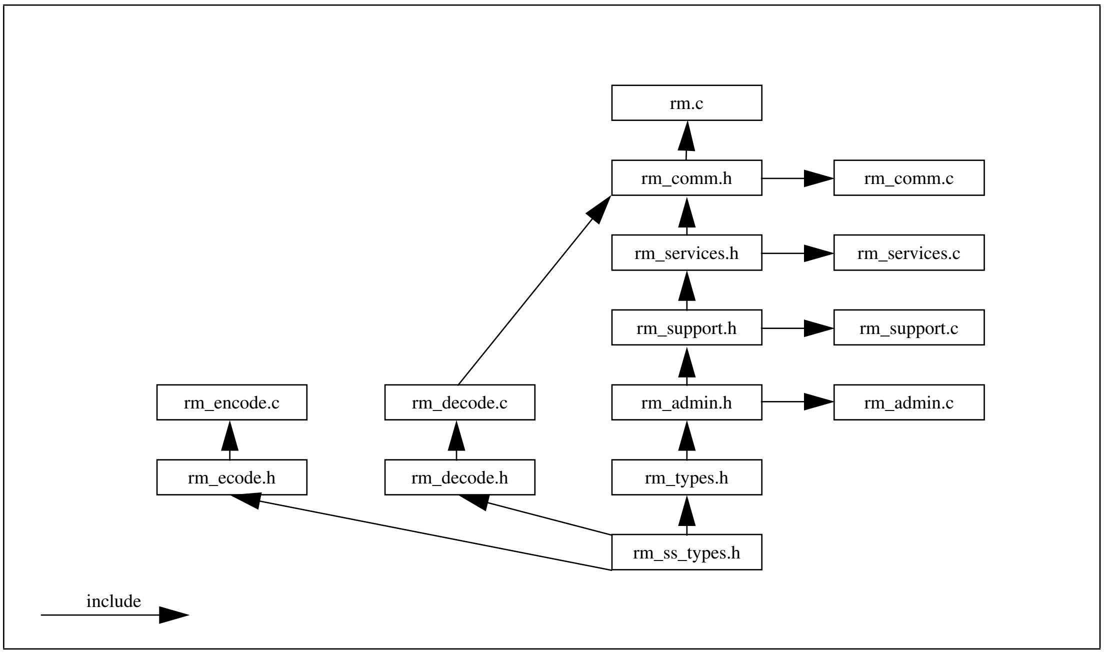

Abb. 13: Module der Resource-Manager

rm_ss_types.h: Hier werden die Datentypen definiert, die sowohl in den Step-Servern, als auch in den Resource-Managern benötigt werden. Hierunter fallen insbesondere diejenigen Typen, die für die Aufrufparameter der angebotenen Dienste genutzt werden.

rm_types.h: Dieses Modul enthält alle Datentypen für die Verwaltungsstrukturen, die die Synchronisationsinformationen aufnehmen sollen. Hierzu gehören Informationen zu den von einem Resource-Manager verwalteten Objekten und zu den hierzu existierenden Invarianten.

rm_admin.h: Die Synchronisationsinformation wird teilweise in Zeigerstrukturen verwaltet. Für die Verwaltung der Knoten dieser Strukturen werden separate Funktionen bereitgestellt. Dieses Modul deklariert die Schnittstellen hierzu.

rm_admin.c: Hier wird die interne Verwaltungsstruktur der Knoten dynamischer Strukturen definiert und die in rm_admin.h vereinbarten Funktionen werden implementiert.

rm_support.h: Dieses Modul stellt alle Schnittstellen für Funktionen zur Verfügung, die Teilaufgaben bei der Implementierung der Dienste übernehmen.

rm_support.c: Die in rm_support.h deklarierten Funktionen werden hier implementiert.

rm_services.h: Ein Resource-Manager muß für die Synchronisationsverwaltung Dienste bereitstellen. Deren Schnittstellen werden in diesem Modul eingeführt.

rm_services.c: Dieses Modul beinhaltet die Implementierung der Dienste. Hierzu wird auf Funktionen von rm_support zurückgegriffen.

rm_comm.h: Hierin werden alle Funktionen vereinbart, die Aufgaben bei der Kommunikation mit Step-Servern übernehmen.

rm_comm.c: Aufgabe dieses Moduls ist die Implementierung der Kommunikationsfunktionen.

rm_encode.h: Die Parameter der Dienste müssen für die Kommunikation mit den Step-Servern codiert werden. Dieses Modul stellt die Schnittstellen zu den hierfür zuständigen Funktionen zur Verfügung.

rm_encode.c: Dieses Modul enthält die Implementierung der in rm_encode.h vereinbarten Kodierungsfunktionen.

rm_decode.h, rm_decode.c: Hierin sind die Schnittstellen bzw. die Implementierung der Funktionen zur Dekodierung der Parameter enthalten.

5.1.2 Schnittstellen der Dienste

Für Aufgaben der Synchronisationsverwaltung bietet ein Resource-Manager unterschiedliche Dienste an. In diesem Abschnitt werden deren Schnittstellen angegeben. Zusätzlich wird jeweils die Aufgabe des Dienstes kurz erklärt und die möglichen Fehlercodes angegeben. Die Datentypen der Parameter sind in Abschnitt 5.1.3 näher erläutert. Hier wird nur dargestellt, welche Informationen mit Hilfe der Parameter an den jeweiligen Dienst übergeben werden.

5.1.2.1 RM_establish_virtual_object

Dieser Dienst erlaubt es beim Resource-Manager ein neues virtuelles Objekt zu vereinbaren.

Schnittstelle

```
RM_error RM_establish_virtual_object(object_name_t,
                                     struct object_type_t *,
                                     struct expression_node_t *);
```

Als Parameter wird der Name des virtuellen Objekts, ein Zeiger auf die Typbeschreibung und ein Zeiger auf einen Ausdruck übergeben, der den Aufbau des virtuellen Objekts beschreibt.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt.
NO_MEMORY	:	Der Speicherplatz reicht für die Aufnahme des neuen Objekts nicht aus.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.1.2.2 RM_delete_virtual_object

Virtuelle Objekte existieren oft nur vorübergehend während eines ConTracts. Mit Hilfe dieses Dienstes wird das Löschen eines solchen Objekts ermöglicht.

Schnittstelle

```
RM_error RM_delete_virtual_object( object_name_t );
```

Als Parameter ist nur der Name des zu löschenden Objekts notwendig.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt
OBJECT_NOT_VIRTUAL	:	Beim angegebenen Objekt handelt es sich nicht um ein virtuelles Objekt.
OBJECT_NOT_FOUND	:	Das angegebene Objekt ist dem Resource-Manager nicht bekannt.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.1.2.3 RM_establish_remote_object

In Ausdrücken, die virtuelle Objekte beschreiben, können Objekte vorkommen, die einem Resource-Manager lokal nicht bekannt sind. Mit diesem Dienst kann ein Step-Server einem Resource-Manager ein solches Objekt bekannt machen und ihm den zuständigen Resource-Manager mitteilen. Dies versetzt den Resource-Manager in die Lage über den entfernten Resource-Manager den Wert des Objekts zu erfahren.

Schnittstelle

```
RM_error RM_establish_remote_object(object_name_t,
                                    rm_name_t );
```

Die Parameter geben den Namen des Objekts und den Namen des zuständigen Resource-Managers an.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt.
NO_MEMORY	:	Der Speicherplatz reicht für die Aufnahme des neuen Objekts nicht aus.

5.1.2.4 RM_get_object_value

Für die Evaluierung von Invarianten über virtuelle Objekte muß ein Resource-Manager oft auf entfernte Objekte zugreifen. Hierfür stellt jeder Resource-Manager diesen Dienst zur Verfügung.

Schnittstelle

```
RM_error RM_get_object_value(object_name_t,
                             expression_string_t * );
```

Als Parameter wird der Name des Objekts übergeben, dessen Wert benötigt wird. Außerdem wird ein Zeiger auf den String, in dem das Ergebnis abgelegt werden soll, mitgeteilt.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt.

OBJECT_NOT_FOUND	:	Das angegebene Objekt ist dem Resource-Manager nicht bekannt.

5.1.2.5 RM_establish_predicate

Dieser Dienst nimmt neue Prädikate von Step-Servern entgegen und verarbeitet diese.

Schnittstelle

```
RM_error RM_establish_predicate(expression_string_t,
                                 contract_id_t );
```

Dem Dienst wird ein Ausdruck übergeben, der das Prädikat enthält. Zusätzlich wird mitgeteilt, für welchen ConTract die Invariante etabliert werden soll.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt.
PREDICATE_VIOLATION	:	Das Prädikat kann nicht angenommen werden, da es in der aktuellen Situation nicht erfüllt ist.
PREDICATE_CONFLICT	:	Das Prädikat kann nicht angenommen werden, da es einem bereits existierenden Prädikat widerspricht.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.1.2.6 RM_delete_predicate

Wird ein Prädikat nicht mehr benötigt, so kann es mit Hilfe dieses Dienstes gelöscht werden.

Schnittstelle

```
RM_error RM_delete_predicate(  expression_string_t,
                               contract_id_t );
```

Wie bei `RM_establish_predicate` wird der Prädikatausdruck und der zugehörige ConTract übergeben.

Fehlercodes

TRUE	:	Dienst wurde erfolgreich ausgeführt
PREDICATE_NOT_FOUND	:	Das angegebene Prädikat existiert nicht.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.1.2.7 RM_end_of_contract

Dem Resource-Manager muß das Ende eines ConTracts angezeigt werden können. Hierzu dient dieser Dienst. Er sorgt u.a. für das Löschen aller zum ConTract gehörenden Invarianten.

Schnittstelle

```
RM_error RM_end_of_contract( contract_id_t );
```

Als Parameter ist nur der Identifikator für den ConTract notwendig.

Fehlercodes

TRUE	:	Dienst erfolgreich ausgeführt.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.1.2.8 RM_change_object_value

Dieser Dienst zeigt den Änderungswunsch für ein Objekt an. Es wird überprüft, ob durch den neuen Objektwert Prädikate verletzt werden oder nicht. Nur wenn dies nicht der Fall ist wird dem Objekt der neue Wert tatsächlich zugewiesen.

Schnittstelle

```
RM_error RM_change_object_value(object_name_t,
                                expression_string_t);
```

Als Parameter wird der Name des zu ändernden Objekts und der neue Objektwert als Zeichenkette übergeben.

Fehlercodes

TRUE	:	Dienst erfolgreich ausgeführt.
NO_MEMORY	:	Der Speicherplatz reicht für die Aufnahme des neuen Objektwerts nicht aus.
OBJECT_NOT_FOUND	:	Das angegebene Objekt existiert lokal nicht.
FALSE	:	Sonstige Fehler machen die korrekte Ausführung des Dienstes unmöglich.

5.1.2.9 RM_check_predicate

Mit Hilfe dieses Dienstes kann ein Resource-Manager unabhängig von der Objektänderung dazu aufgefordert werden, ein Prädikat zu überprüfen. Dies ist z.B. bei der Prüfung von Eingangsinvarianten der Steps der Fall.

Schnittstelle

```
RM_error RM_check_predicate(expression_string_t);
```

Dem Dienst muß lediglich der Prädikatausdruck übergeben werden. Die Überprüfung findet anders als bei RM_establish_predicate ohne Kenntnis des aufrufenden ConTracts statt.

Fehlercodes

TRUE	:	Der Dient wurde erfolgreich ausgeführt und das Prädikat gilt.
PREDICATE_VIOLATION	:	Der Dienst wurde erfolgreich ausgeführt, das Prädikat ist aber nicht erfüllt.
FALSE	:	Sonstige Fehler machten eine korrekte Ausführung des Dienstes unmöglich.

5.1.3 Datenstrukturen

Dieser Abschnitt beschreibt die Datentypen, die für die Synchronisationsverwaltung in den Resource-Managern benötigt werden. Zunächst sollen hier diejenigen Datentypen betrachtet werden, die sowohl in Resource-Managern als auch in Step-Servern notwendig sind. Dies sind die Datentypen für die Parameter der von Resource-Managern angebotenen Dienste, die von Step-Servern aufgerufen werden.

Identifikatoren für ConTracts

```
#define CONTRACT_ID_LEN 60
typedef char contract_id_t[CONTRACT_ID_LEN];
```

Dieser Typ ermöglicht die Angabe von Identifikatoren für ConTracts. Da jeder ConTract in mehreren Versionen parallel ablaufen kann, handelt es sich um den Identifikator einer Instanz eines ConTracts. Dieser setzt sich aus mehreren Teilen zusammen[7]. Die interne Struktur interessiert in dieser Arbeit nicht, so daß die Identifikatoren hier als eine Zeichenkette mit 60 Zeichen betrachtet werden.

Bezeichner für Objekte

```
#define OBJECT_NAME_LEN 60
typedef char object_name_t[OBJECT_NAME_LEN];
```

Mit Hilfe dieses Typs werden Bezeichner für Objekte dargestellt.

Bezeichner für Resource-Manager

```
#define RM_NAME_LEN 60
typedef char rm_name_t[RM_NAME_LEN];
```

Mit Hilfe dieses Typs werden Bezeichner für Resource-Manager dargestellt.

Ausdrücke

```
#define EXPRESSION_LEN 500
typedef char expression_string_t[EXPRESSION_LEN];
```

Ausdrücke beschreiben den Aufbau von Invarianten und den virtueller Objekte. Die Ausdrücke werden an den Schnittstellen als Zeichenketten übergeben und intern dann aufgespalten.

7. Vgl. [Seif95], S. 50

Typausdrücke

```
#define TYPE_EXPRESSION_LEN 500
typedef char tpye_expression_string_t[TYPE_EXPRESSION_LEN];
```

Mit Typausdrücken wird der Typ von Objekten angegeben.

Nachrichtenpuffer

```
#define MESSAGE_LEN 1000
typedef char message_buffer_t[MESSAGE_LEN];
```

Nachrichten, die Step-Server und Resource-Manager austauschen werden in Nachrichtenpuffer geschrieben und aus diesen gelesen. Dieser Typ ermöglicht die Bereitstellung dieser Puffer.

Fehlercodes

```
typedef enum { FALSE, TRUE, TOO_MANY_OBJECT_INFO,
          TOO_MANY_OBJECT_VALUE, TOO_MANY_OBJECT_TYPE,
          TOO_MANY_CONSTANT_TYPE, TOO_MANY_EXPRESSION_NODE,
          TOO_MANY_PREDICATE_INFO, TOO_MANY_PREDICATE_LIST_NODE,
          TOO_MANY_VIRT_OBJ_LIST_NODE, OBJECT_NOT_FOUND,
          OBJECT_NOT_VIRTUAL, UNKNOWN_OBJECT_CLASS,
          UNKNOWN_OBJECT_VALUE, UNKNOWN_OPERAND,
          UNKNOWN_PREDICATE_TYPE, UNKNOWN_EXPRESSION_NODE_TYPE,
          TYPE_MISMATCH, USE_SINGLE_PREDICATES, PREDICATE_CONFLICT,
          PREDICATE_VIOLATION, EXPRESSION_ERROR, NO_MEMORY }
          RM_error;
```

Dieser Aufzählungstyp enthält sowohl die Returncodes der Resource-Manager-Dienste, als auch die der internen Funktionen.

Objektklasse

Jeder Resource-Manager verwaltet Informationen zu Objekten, die lokal bei ihm selbst vorliegen. Hinzu kommen Angaben zu Objekten, die von entfernten Resource-Managern verwaltet werden und Angaben zu virtuellen Objekten. Jedem Objekt kann mit Hilfe des Typs `object_class_t` die eigene Objektklasse zugeordnet werden.

```
typedef enum { LOCAL, REMOTE, VIRTUAL } object_class_t;
```

Objekttypen

Für die einzelnen Objekte aller Klassen muß der Typ festgehalten werden können. Obwohl in dieser Arbeit lediglich die Typen INTEGER und ARRAY OF INTEGER näher betrachtet werden sollen, ist die Datenstruktur so ausgelegt, daß alle in [Schw95] aufgeführten Datentypen dargestellt werden können. Damit müssen auch geschachtelte Strukturen, wie z.B. Arrays mit Strukturen als Komponenten berücksichtigt werden. Darum muß die Typbeschreibung mit Hilfe einer dynamischen Struktur erfolgen. Die einzelnen Knoten der Struktur sind vom Typ `object_type_t`. Sie enthalten jeweils einen Typbezeichner vom Typ `object_type_name_t` und Angaben zu den Komponenten. Ist der Knoten selbst eine Komponente eines übergeordneten Typs, so wird eine Komponentennummer `component_number` und der Zeiger `next` auf die nächste Komponente angegeben. Hat der Knoten Unterkomponenten, so ist deren Anzahl in `number_of_components` festgehalten und `list_of_components` zeigt auf die erste dieser Komponenten. Mit Hilfe dieser Datentypen aufgebaute Strukturen erlauben auch die Beschreibung komplexer Typstrukturen.

```
typedef enum { INTEGER, FLOATING, CHARACTER, BOOLEAN, ARRAY, STRUCT,
         SET, TIME } object_type_name_t;
typedef struct {    int component_number;
                    object_type_t *next;
                    object_type_name_t type;
                    int number_of_components;
                    object_type_t *list_of_components;
                    } object_type_t;
```

Objektwerte

Die Beschreibung von Objektwerten ähnelt in ihrer Struktur der von Objekttypen. Wieder wird eine dynamische Struktur mit object_value_t als Knoten verwendet. Ein solcher Knoten enthält eventuell einen einfachen Objektwert in object_value, sowie den Verweis `next` auf die nächste Komponente in einem Array oder einer Struktur. Außerdem enthält `list_of_components` den Verweis auf die erste Unterkomponente.

```
typedef union { long integer_value;
                double floating_value;
                char character_value;
                boolean boolean_value;
```

```
} simple_object_value_t;

typedef struct { simple_object_value_t object_value;
                 object_value_t *next;
                 object_value_t *list_of_components;
                 } object_value_t;
```

Ausdrucksstrukturen

Die Beschreibung von Invarianten und virtuellen Objekten wird den Diensten der Resource-Manager als Strings übergeben. Intern jedoch ist die Speicherung als String nicht sinnvoll. Die Gründe hierfür sollen kurz erläutert werden.

Die Speicherung einer Invariante als String bietet den Vorteil, daß die Information vergleichsweise komprimiert gehalten werden kann. Beim Etablieren einer neuen Invariante entsteht kein großer Aufwand durch deren Verarbeitung. Dem steht als Nachteil gegenüber, daß bei jeder Auswertung der Invariante das Parsen derselben notwendig wird. Da bei jeder Änderung von Objekten alle betroffenen Invarianten ausgewertet werden müssen sollte gerade bei dieser häufigen Aufgabe der Synchronisationsverwaltung ein möglichst geringer Aufwand notwendig sein.

Werden die Invarianten in den Resource-Managern als dynamische Struktur gespeichert, so bedeutet dies auf der einen Seite einen größeren Speicherbedarf und einen erhöhten Aufwand beim Einbringen einer neuen Invariante. Schließlich muß der übergebene String zuerst in die Struktur umgewandelt werden. Als Vorteil steht hier ein geringerer Aufwand bei der Auswertung der Invariante gegenüber. Da die Struktur der Invariante bereits erfaßt wurde müssen hier nur noch die aktuellen Objektwerte beschafft und die Anwendung der Operatoren auf dieselben berücksichtigt werden.

Der zuletzt genannte Vorteil der Speicherung von Invarianten in einer Struktur überwiegt alle Nachteile, da es das Hauptziel bei dem Entwurf der Datenstruktur sein muß, die Auswertung von Invarianten möglichst effektiv zu gestalten. Es wird darum hier eine Struktur eingeführt, mit `expression_node_t` als Typ der Knoten. Jeder Knoten kann sowohl einen Operator, ein Objekt oder eine Konstante beinhalten. Dies wird in `node_type` festgehalten. Steht der Knoten für einen Operator, so wird dieser in `op` vermerkt. Die Verweise auf die beiden Operanden werden durch `operand_l` und `operand_r` repräsentiert. Für den Fall, daß der Knoten für ein Objekt steht, ist dessen Bezeichner in `object_name` gespeichert. Bleibt noch der dritte Knotentyp. Für Konstanten wird ein Verweis auf deren Typbeschreibung in `constant_type` und ein Verweis auf ihren Wert in `constant_value` eingetragen.

```
typedef enum { OPERATOR, OBJECT, CONSTANT } expression_node_type_t;

typedef enum { NONE, EQUAL, NOT_EQUAL, SMALLER, SMALLER_EQUAL BIGGER,
         BIGGER_EQUAL, PLUS, MINUS, MULT, DIV, NEGATION, BIT_AND,
         BIT_OR, BIT_XOR, SHIFT_LEFT, SHIFT_RIGHT, NOT, AND, OR, AT,
         BEFORE, AFTER, IN, COUNT, EXIST, LENGTH } operator_t;

typedef struct { expression_node_type_t node_type;
                 expression_node_t *next;
                 operator_t op;
                 expression_node_t *operand_l;
                 expression_node_t *operand_r;
                 object_name_t object_name;
                 object_type_t *constant_type;
                 object_value_t *constant_value;
                 } expression_node_t;
```

Prädikate

Alle Prädikate werden innerhalb einer Liste gespeichert. Damit muß keine Obergrenze für die Anzahl der von einem Resource-Manager zu verwaltenden Prädikate festgelegt werden. Zunächst müssen zwei verschiedene Prädikattypen unterschieden werden. Es kann sich entweder um Prädikate handeln, die direkt von einem Step-Server übergeben wurden, oder um solche, die eine Vereinfachung oder Zusammenfassung anderer Prädikate darstellen. Dies wird mit `predicate_type_t` realisiert.

Für jedes Prädikat wird festgehalten, ob es gerade gültig ist oder nicht. Gültig heißt hier, daß es bei anstehenden Objektänderungen überprüft werden muß. Prädikate, die gerade nicht gültig sind können nicht einfach entfernt werden, wenn sie mit anderen Prädikaten zusammengefaßt werden. Dann werden beim Löschen der Zusammenfassung wieder die ursprünglichen Prädikate gültig. Der aktuelle Status wird mit `predicate_state_t` beschrieben.

Auch die Prädikatklasse, zu der ein Prädikat gehört, soll jeweils vermerkt werden. Dies erfolgt mit dem Typ `predicate_class_t`. Ein Vorteil dieser Information macht sich bei der Zusammenfassung von Invarianten bemerkbar. Die in Frage kommenden Zusammenfassungen sind oft abhängig von den Invariantenklassen. Wurde die Klasse für jedes Prädikat bei dessen Etablierung einmal gespeichert, so kann die Suche nach möglichen Optimierungen stark eingeschränkt werden.

Folgende Angaben werden nun für jedes Prädikat vermerkt. `predicate` enthält einen Verweis auf den Ausdruck, der das Prädikat enthält. Die Prädikatklasse wird in `predicate_class`, der Zustand des Prädikats in predicate_state und der Typ des Prädikats in `predicate_type` festgehalten. Der Verweis auf das nächste Prädikat in der Liste der genutzten Prädikate steht in `next`. Für jedes

Prädikat muß auch bekannt sein, von welchem ConTract es stammt. Diese Information wird in `contract_id` festgehalten. Handelt es sich bei dem Prädikat um eine Zusammenfassung aus anderen Prädikanten, so enthalten `from_l` und `from_r` die Verweise auf diese. Wurde das Prädikat selbst Teil einer Zusammenfassung, so ist `to` ein Verweis auf diese. Diese Verweise sind alle notwendig, um beim Löschen von Prädikaten die davon abhängigen schnell zu finden. Vom Löschen sind sowohl die aus einem Prädikat hervorgegangen Zusammenfassungen, als auch die Ausgangsprädikate, falls es sich um eine Zusammenfassung handelt, betroffen. Die ersteren müssen ebenfalls gelöscht werden, die zweiteren werden wieder gültig.

```
typedef enum { CLASS_1, ... , CLASS_16 } predicate_class_t;

typedef enum { VALID, NOT_VALID } predicate_state_t;;

typedef enum { ORIGINAL, COMBINATION } predicate_type_t;

typedef struct {  expression_node_t *predicate;
                  predicate_class_t predicate_class;
                  predicate_state_t predicate_state;
                  predicate_type_t predicate_type;
                  predicate_info_t *next;
                  contract_id_t contract_id;
                  predicate_info_t *from_l;
                  predicate_info_t *from_r;
                  predicate_info_t *to;
                  } predicate_info_t;
```

Objekte

Die Zahl der Objekte, über die ein Resource-Manager Informationen halten muß, ist nicht von vornherein beschränkt. Darum werden die Objektinformationen in einer Liste gehalten, so daß Objekte hinzukommen und wegfallen können. Für jedes Objekt sind folgende Informationen notwendig: Der Name des Objekts wird in `object_name` festgehalten. Klasse und Typ des Objekts stehen in `object_class` bzw. `object_type`. `next` enthält den Verweis auf das nächste Objekt in der Liste. Für lokale Objekte enthält `object_value` den Verweis auf den Objektwert. In der vorliegenden Implementierung werden auch lokale Objekte nur im Hauptspeicher verwaltet. Bei einer Erweiterung auf persistente Objekte kann die Speicherung von Objektwerten in dieser Struktur entfallen. Für entfernte oder virtuelle Objekte muß im Normalfall kein Wert gespeichert werden. Dies kann aber sinnvoll sein, wenn über verschiedene Optimierungen versucht wird, daß nicht bei jedem Zugriff auf ein solches Objekt der entfernte Resource-Manager befragt oder der das virtuelle Objekt beschreibende Ausdruck ausgewertet werden muß. Die Speicherung von Objektwerten wird darum für alle Objektklassen vorgesehen. Bei entfernten Objekten muß `rm_name` den Namen des zuständigen Resource-Managers enthalten. Für virtuelle Objekte wird deren Aufbau in expression vermerkt. Zusätzlich kommen die beiden Zeiger predicates und `virtual_objects` für jedes Objekt hinzu. Der erste zeigt auf eine Verweisliste, die für jedes Prädikat, das das Objekt enthält einen Verweis bereitstellt. Der zweite Zeiger führt zu einer ähnlichen Liste, die Verweise auf alle virtuellen Objekte enthält, in die das vorliegende Objekt eingeht. Warum diese zusätzlichen Verweise? Sie helfen, das Auffinden der für ein bestimmtes Objekt zuständigen Prädikate effizienter zu gestalten. Ohne sie müßte bei einem Änderungswunsch für ein Objekt folgendes durchgeführt werden:

- Alle Prädikate müssen durchsucht werden, ob sie etwas über das zu ändernde Objekt aussagen. Die Prädikate, für die das zutrifft, sind auszuwerten.

- Alle virtuellen Objekte müssen daraufhin untersucht werden, ob das zu löschende Objekt in ihnen verwendet wird. Für jedes gefundene virtuelle Objekt sind die Prädikate zu suchen, die etwas über das virtuelle Objekt aussagen. Diese Prädikate müssen schließlich ausgewertet werden.

Um diesen Aufwand der Suche nach zuständigen Prädikaten zu vermeiden, werden die oben genannten Verweislisten eingeführt. Mit deren Hilfe kann dann bei jeder Objektänderung leicht ermittelt werden, welche Prädikate vor dem Zulassen der Änderung überprüft werden müssen.

```
typedef struct { predicate_info_t *predicate;
                 predicate_list_node_t *next;
                 } predicate_list_node_t;

typedef struct { object_info_t *object;
                 virt_obj_list_node_t *next;
                 } virt_obj_list_node_t;

typedef struct { object_name_t object_name;
                 object_class_t object_class;
                 object_type_t *object_type;
                 object_value_t *object_value
                 rm_name_t rm_name;
                 expression_node_t *expression;
                 predicate_list_node_t *predicates;
```

```
virt_obj_list_node_t *virtual_objects;
} object_info_t;
```

5.1.4 Algorithmen

RM_establish_virtual_object

Zunächst wird ein neuer Objektinformationsblock beschafft. In diesen wird der Name des neuen virtuellen Objekts eingetragen. Der als String übergebene Ausdruck zur Beschreibung des Objektaufbaus wird mit Hilfe der Funktion `parse_expression` in eine Ausdrucksstruktur überführt. Der so erhaltene Ausdruck wird mit Hilfe von `get_expression_value` ausgewertet, so daß der Typ und der aktuelle Wert des Objekts im Objektinformationsblock gespeichert werden kann. Anschließend wird mit `appoint_virtual_object_to_objects` dafür gesorgt, daß das virtuelle Objekt bei alldenjenigen Objekten in der Objektliste vermerkt wird, die im virtuellen Objekt enthalten sind. Sind alle diese Funktionen erfolgreich ausgeführt, so kann das virtuelle Objekt akzeptiert werden.

RM_delete_virtual_object

Für die Ausführung dieser Funktion muß zunächst der Zeiger auf den Objektinformationsblock bereitgestellt werden. Dies erfolgt mit `get_object_ptr`. Anschließend wird, falls es sich tatsächlich um ein virtuelles Objekt handelt, `delete_virtual_object_from_objects` ausgeführt. Dies stellt sicher, daß das virtuelle Objekt aus allen Objektverweislisten bei Objekten gelöscht wird. Anschließend kann der das virtuelle Objekt beschreibende Ausdruck und der Objektinformationsblock selbst gelöscht werden. War dies alles erfolgreich, so ist das Löschen des virtuellen Objekts geglückt.

RM_establish_remote_object

Dieser Dienst legt lediglich einen neuen Objektinformationsblock an und hält in diesem den Namen des Objekts, den zuständigen Resource-Manager und die Objektklasse fest.

RM_establish_predicate

Für das Etablieren eines neuen Prädikats muß zunächst ein neuer Prädikatinformationsblock bereitgestellt werden. Danach kann der als String übergebene Prädikatausdruck in eine Ausdrucksstruktur verwandelt werden (`parse_expression`), sowie der Prädikatinformationsblock mit Werten belegt werden. Es folgt die Überprüfung des Prädikats. Falls dies in der aktuellen Situation nicht erfüllt ist wird es wieder gelöscht.

Vor der weiteren Ausführung wird das Prädikat mit `appoint_predicate_to_objects` bei allen betroffenen Objekten in die Prädikatverweisliste eingetragen. Danach kann der Reihe nach versucht werden, das neue Prädikat mit bereits existierenden Prädikaten zusammenzufassen. Dies erfolgt mit Hilfe der Funktion `simplify`. Bei deren Aufruf muß sichergestellt sein, daß ein Prädikat nicht mit sich selbst zusammengefaßt wird, und daß die Kandidaten für die Zusammenfassung auch gültig sind. `Simplify` liefert als Ergebnis entweder die Angabe, daß die Prädikate getrennt verwaltet werden müssen, oder den Hinweis auf einen Konflikt der Prädikate oder aber das sich ergebende zusammengefaßte Prädikat. Bei einem Prädikatkonflikt wird das neue Prädikat gelöscht und der entsprechende Rückgabewert geliefert. Bei geglückter Zusammenfassung wird das zusammengefaßte Prädikat in die Liste der vorhandenen Prädikate aufgenommen, die ursprünglichen Prädikate werden ungültig und der Verweis beim neuen Prädikat auf seine Ursprünge muß angepaßt werden. Danach gilt die Zusammenfassung als neues Prädikat und es kann erneut versucht werden Zusammenfassungen zu erreichen.

RM_delete_predicate

Damit das zu löschende Prädikat in der Liste der existierenden Prädikate gefunden werden kann, muß zunächst der Prädikatausdruck in eine Ausdrucksstruktur verwandelt werden (`parse_expression`). Danach wird das Prädikat in der Liste vorhandener Prädikat gesucht. Hierzu wird jeweils der Prädikatausdruck und der eingetragene ConTract verglichen. Wurde das Prädikat gefunden, so wird mit `unget_expression` die Ausdrucksstruktur und mit `erase_predicate` das komplette Prädikat und alle davon abhängigen gelöscht.

RM_end_of_contract

Die Realisierung dieses Dienstes gestaltet sich recht einfach. Die Liste der Prädikate wird sequentiell durchsucht und bei jedem geprüft, ob es vom beendeten ConTract stammt. Alle Prädikate, für die dies zutrifft, werden mit Hilfe von `erase_predicate` gelöscht. Diese Funktion entfernt auch alle abhängigen Prädikate.

RM_check_predicate

Diesem Dienst wird lediglich das zu überprüfende Prädikat als String übergeben. Die entsprechende Ausdrucksstruktur wird zunächst mit `parse_expression` erstellt. Danach wird dieser Ausdruck in einen neuen Prädikatinformationsblock eingetragen. Da das Prädikat nicht dauerhaft gespeichert werden soll ist dieser nur vorrübergehend notwendig und wird am Ende dieses Dienstes wieder gelöscht. Mit dem Prädikatinformationsblock als Argument wird nun `test_predicate` aufgerufen. Hier findet die eigentliche Überprüfung des Prädikats statt. Entsprechend dem Ergebnis dieser Funktion muß der Dienst eine Prädikatverletzung signalisieren oder nicht.

RM_change_object_value

Zunächst wir der zum angegebenen Objekt gehörende Objektinformationsblock beschafft. Falls dieses Objekt nicht existiert oder es sich nicht um ein lokales Objekt handelt, kann der Dienst nicht fortgesetzt werden. Ist das Objekt lokal vorhanden, so wird zunächst sein alter Wert gerettet und der neue Wert sowie der neue Typ aus dem angegebenen Ausdrucksstring bestimmt. War dies alles erfolgreich, so kann das Objekt vorrübergehend den neuen Wert erhalten. Jetzt muß überprüft werden, ob alle betroffenen Prädikate trotzdem gelten. Diese müssen dank der gewählten Datenstrukturen nicht in der Menge aller Prädikate gesucht werden, sondern mit Hilfe der den Objekten zugeordneten Verweislisten ist ein Zugriff direkt möglich. Zunächst wird die Prädikatverweisliste des Objekts durchlaufen und jedes der dort eingetragenen Prädikate überprüft. Haben diese alle ihre Gültigkeit behalten, so kann mit der Verweisliste auf virtuelle Objekte fortgesetzt werden. Diese enthält alle virtuellen Objekte, in die das zu ändernde Objekt eingeht. Bei jedem dieser virtuellen Objekte ist wiederum eine Liste zugeordneter Prädikate vorhanden, die alle geprüft werden müssen. Sind dann alle Prädikate für virtuelle Objekte, die dem Objekt zugeordnet sind, überprüft, so steht als Ergebnis fest, ob die Änderung zulässig ist oder nicht. Falls nicht wird dem Objekt wieder der alte Wert zugewiesen.

5.1.5 Ergänzungen für eine vollständige Implementierung

Wie in der Einleitung zu Kapitel 5 bereits beschrieben wurde ist in dieser Arbeit keine vollständige Implementierung der Synchronisationsverwaltung angestrebt. In diesem Abschnitt soll kurz dargestellt werden, welche Erweiterungen der vorliegenden Implementierung auf dem Weg dorthin notwendig sind.

Integration aller Komponenten

Die Synchronisationsverwaltung wie sie hier implementiert wurde, wird zum größten Teil von den Step-Servern und Resource-Managern ausgeführt. Aber auch andere APRICOTS-Komponenten müssen für die vollständige Integration Dienste anbieten. So ist es z.B. Aufgabe des ConTract-Managers gelesene Invarianten an die Step-Server weiterzuleiten. Zu diesen für die Integration in APRICOTS notwendigen Schritten gehört auch, daß die Kommunikation der Komponenten nicht mit

Hilfe von Interprozeßkommunikation erfolgt. Anzustreben ist die Verwendung des Kommunikationssystems für APRICOTS[8].

Zusätzliche Datentypen und Prädikate

Bei den in Abschnitt 5.1.3 beschriebenen Datenstrukturen sind alle in [Schw95] aufgeführten Datentypen und Prädikate berücksichtigt. Für die vollständige Implementierung ist es notwendig, die Dienste so zu erweitern, daß sie zusätzliche Typen und Prädikate verarbeiten können. Dies beginnt bei der Umwandlung von Ausdruckszeichenketten in die internen Verwaltungsstrukturen und endet bei weiteren Möglichkeiten für die Zusammenfassung von Prädikaten.

Zugriff auf Objekte

Bei der vorliegenden Implementierung werden die Objekte lediglich im Hauptspeicher verwaltet. Die eigentliche Aufgabe von Resource-Managern ist es aber, die Zugriffe auf Objekte zu synchronisieren, die z.B. in einer Datenbank abgelegt sind. Der Resource-Manager muß dann jeweils dort, wo auf den aktuellen Wert eines Objekts zugegriffen wird um entsprechende Datenbankanfragebefehle erweitert werden. Dies bedeutet aber nicht, daß die vorrübergehende Speicherung der Objektwerte im Hauptspeicher nach einer solchen Erweiterung entfallen sollte. Wird nämlich auf ein Objekt z.B. während der Auswertung einer Invarianten mehrmals zugegriffen, so ist dies mit Hilfe der beschriebenen Datenstrukturen effektiver zu bewerkstelligen, als mit mehrfachen Datenbankzugriffen. Da der Resource-Manager für die Kontrolle der Datenbankobjekte zuständig ist, weiß er auch, wann sich ein Objekt ändert und der im Hauptspeicher vorliegende Wert damit nicht mehr aktuell ist.

Speicherung der Verwaltungsinformationen

Für die Integration der Synchronisationsverwaltung in APRICOTS ist es notwendig die Verwaltungsinformation der Resource-Manager persistent zu speichern. Dies resultiert aus den Anforderungen an das System, eine zuverlässige Abwicklung von ConTracts zu ermöglichen. Da Komponenten nach ihrem Ausfall wieder dort aufsetzen sollen, wo sie beendet wurden, ist die sichere Speicherung der hierfür notwendigen Informationen vorzusehen. Solche Informationen, die im Rahmen der Synchronisationsverwaltung in jedem Resource-Manager anfallen, sind alle Prädikat- und Objektinformationen. Für deren Speicherung in einer relationalen Datenbank soll an dieser Stelle ein geeignetes Datenbankschema vorgeschlagen werden. Es werden dabei lediglich die notwendigen Tabellen mit den Typen der einzelnen Spalten aufgeführt. Die Typangabe orientiert sich hierbei nicht an den Möglichkeiten eines

8. Vgl. [Schü93]

speziellen DBMS, sondern wird allgemein gehalten. Die Namen der Spalten stimmen soweit wie möglich mit den in den beschriebenen Datenstrukturen verwendeten Namen überein.

objects

ONR	object_name	object_class	object_type	object_value	rm_name	expression
INTEGER	STRING	INTEGER	STRING	STRING	STRING	STRING

predicates

PNR	predicate	predicate_ class	predicate_ state	predicate_ type	contract_i d	from_l	from_r	to
INTEGER	STRING	INTEGER	INTEGER	INTEGER	STRING	INTEGER	INTEGER	INTEGER

predicates_to_object

ONR	PNR
INTEGER	INTEGER

virtual_objects_to_object

ONR	virt_ONR
INTEGER	INTEGER

Die Tabelle predicates wird für die Speicherung der Prädikatinformationen benötigt. Ihr Inhalt entspricht weitgehend dem Prädikatinformationsblock (`predicate_info_t`), wie er in Abschnitt 5.1.3 beschrieben ist. Um Verweise auf Prädikate möglich zu machen wird zusätzlich der Schlüssel PNR vorgeschlagen.

Eine ähnliche Tabelle gibt es auch für die Objektinformationen. Diese beinhaltet die Angaben, wie sie in `object_info_t` stehen. Auch hier wird als Schlüssel eine zusätzliche Spalte eingeführt, nämlich ONR. Gegenüber der C-Datenstruktur verändert sich allerdings die Realisierung der Verweislisten auf Prädikate und virtuelle Objekte zu einem Objekt. Diese müssen jeweils durch separate Tabellen aufgebaut werden. Die Tabelle predicates_to_object weist hierbei jedem Objekt eine Anzahl von Prädikaten zu, wobei jeder einzelne Verweis eine einzelne Reihe der Tabelle darstellt. Die Tabelle virtual_objects_to_object stellt die entsprechenden Verweise von Objekten zu virtuellen Objekten dar.

5.2 Step-Server

Aufgabe der Step-Server im Rahmen der Synchronisationsverwaltung ist im Wesentlichen die Verteilung von Invarianten oder Teilprädikaten auf die einzelnen Resource-Manager. Von der prototypischen Implementierung werden hier zunächst die beteiligten Module beschrieben. Hierauf folgt eine Aufstellung der Dienst-Schnittstellen, die Auflistung der verwendeten Datenstrukturen und Algorithmen sowie die Erläuterung, welche Erweiterungen notwendig sind, um eine vollständige Implementierung zu erhalten.

5.2.1 Module

Die Dienste, die Step-Server für die Synchronisationsverwaltung bereitstellen werden ebenfalls in verschiedenen Modulen realisiert. Diese sind in Abbildung 13 dargestellt

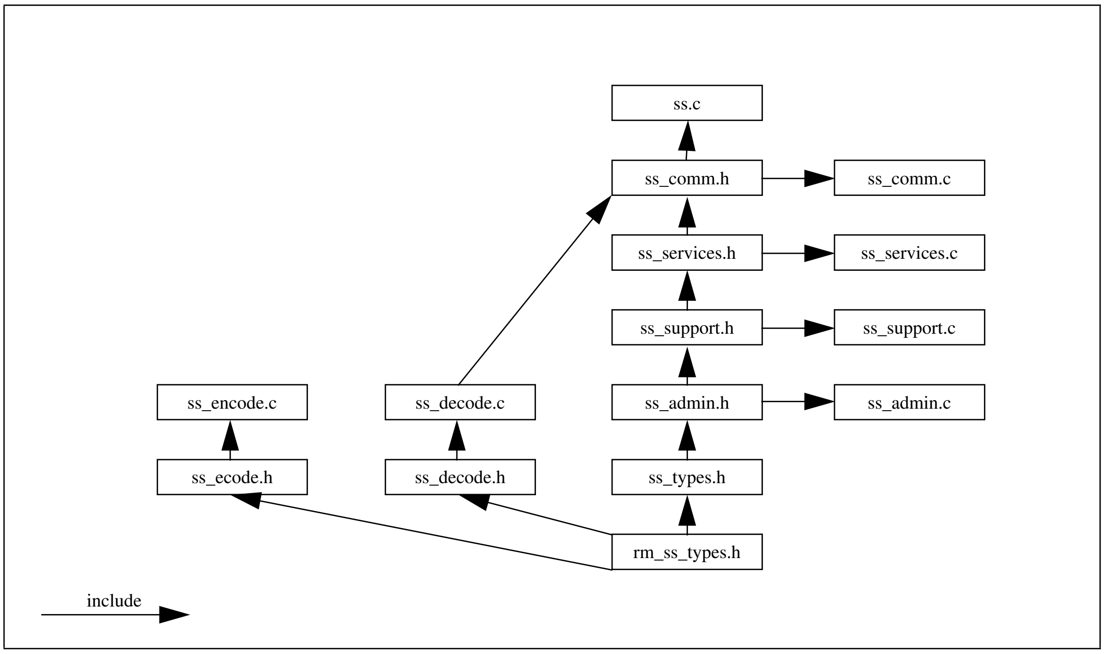

Abb. 14: Module der Step-Server

rm_ss_types.h: Hier werden die Datentypen definiert, die sowohl in den Step-Servern, als auch in den Resource-Managern benötigt werden. Hierunter fallen insbesondere diejenigen Typen, die für die Aufrufparameter der angebotenen Dienste genutzt werden.

ss_types.h: Dieses Modul enthält alle Datentypen für die Verwaltungsstrukturen, die die Synchronisationsinformationen aufnehmen sollen. Hierzu gehören Informationen zu den dem Step-Server bekannten Objekten, zu virtuellen Objekten und zu den an einem ConTract beteiligten Resource-Managern.

ss_admin.h: Die Synchronisationsinformation wird teilweise in Zeigerstrukturen verwaltet. Für die Verwaltung der Knoten dieser Strukturen werden separate Funktionen bereitgestellt. Dieses Modul deklariert die Schnittstellen hierzu.

ss_admin.c: Hier wird die interne Verwaltungsstuktur der Knoten dynamischer Strukturen definiert und die in ss_admin.h vereinbarten Funktionen werden implementiert.

ss_support.h: Dieses Modul stellt alle Schnittstellen für Funktionen zur Verfügung, die Teilaufgaben bei der Implementierung der Dienste übernehmen.

ss_support.c: Die in ss_support.h deklarierten Funktionen werden hier implementiert.

ss_services.h: Ein Resource-Manager muß für die Synchronisationsverwaltung Dienste bereitstellen. Deren Schnittstellen werden in diesem Modul eingeführt.

ss_services.c: Dieses Modul beinhaltet die Implementierung der Dienste. Hierzu wird auf Funktionen von ss_support zurückgegriffen.

ss_comm.h: Hierin werden alle Funktionen vereinbart, die Aufgaben bei der Kommunikation mit Step-Servern übernehmen.

ss_comm.c: Aufgabe dieses Moduls ist die Implementierung der Kommunikationsfunktionen.

ss_encode.h: Die Parameter der Dienste müssen für die Kommunikation mit den Step-Servern codiert werden. Dieses Modul stellt die Schnittstellen zu den hierfür zuständigen Funktionen zur Verfügung.

rm_encode.c: Dieses Modul enthält die Implementierung der in rm_encode.h vereinbarten Kodierungsfunktionen.

rm_decode.h, rm_decode.c: Hierin sind die Schnittstellen bzw. die Implementierung der Funktionen zur Dekodierung der Parameter enthalten.

5.2.2 Schnittstellen

Dieser Abschnitt beschreibt kurz die Schnittstellen der Step-Server-Dienste. Es wird jeweils die Aufgabe des Dienstes, die Parameter und die Fehlercodes angegeben.

5.2.2.1 SS_establish_invariant

Dieser Dienst wird aufgerufen, wenn eine neue Invariante etabliert werden soll.

Schnittstelle

```
SS_error SS_establish_invariant(  expression_string_t,
                                  contract_id_t );
```

Als Parameter wird ein Ausdrucksstring, der die Invariante beschreibt, sowie der Identifikator des ConTracts, von dem die Invariante stammt, übergeben.

Fehlercode

TRUE	:	Dienst erfolgreich ausgeführt.
FALSE	:	Fehler machen eine erfolgreiche Ausführung des Dienstes unmöglich.

5.2.2.2 SS_delete_invariant

Soll eine Invariante gelöscht werden, so ist dieser Dienst zuständig.

```
SS_delete_invariant(  expression_string_t,
                      contract_id_t );
```

Auch hier wird ein die Invariante beschreibende Ausdrucksstruktur und der Identifikator des ConTracts übergeben.

Fehlercodes

TRUE	:	Dienst erfolgreich ausgeführt.

FALSE	:	Fehler machen eine erfolgreiche Ausführung des Dienstes unmöglich.

5.2.2.3 SS_end_of_contract

Dem Step-Server muß das Ende eines ConTracts angezeigt werden, was mit Hilfe dieses Dienstes geschehen kann. Der Step-Server muß dafür sorgen, daß das ConTract-Ende an alle betroffenen Resource-Manager weitergeleitet wird.

Schnittstelle

```
SS_error SS_end_of_contract(contract_id_t);
```

Als Parameter ist nur der Identifikator für den ConTract notwendig.

Fehlercodes

TRUE	:	Dienst erfolgreich ausgeführt.
FALSE	:	Sonstige Fehler machen das korrekte Ausführen des Dienstes unmöglich.

5.2.3 Datenstrukturen

Für die Verwaltung der Synchronisationsinformationen in den Step-Servern sind andere Datenstrukturen notwendig, als in den Resource-Managern. Diese werden im folgenden beschrieben. Neben diesen hier aufgeführten Strukturen gibt es die in rm_ss_types.h festgehaltenen Datentypen, die von Resource-Managern und Step-Servern gleichermaßen genutzt werden. Dies sind Typen für ConTract-Identifikatoren, Objektbezeichner, Resource-Manager-Bezeichner, Ausdrücke, Typausdrücke und Nachrichtenpuffer.

Fehlercodes

```
typedef enum { FALSE, TRUE, TOO_MANY_OBJECT_INFO,
          TOO_MANY_VIRT_OBJECT_INFO, TOO_MANY_RM_LIST_NODE,
```

```
TOO_MANY_CONTRACT_INFO, CONTRACT_NOT_FOUND, NO_MEMORY,
OBJECT_NOT_FOUND } SS_error;
```

Dieser Aufzählungstyp beschreibt die Returncodes der Resource-Manager-Dienste und der internen Funktionen.

Objektinformationen

```
typedef struct {  object_info_t *next;
                  object_name_t object_name;
                  rm_name_t rm_name;
                  } object_info_t;
```

Ein Step-Server muß zu jedem ihm bekannten Objekt den zuständigen Resource-Manager wissen. Hierzu wird für jedes Objekt ein Objektinformationsblock bereitgestellt, der diese Informationen enthält. Die Verwaltung der Blöcke erfolgt innerhalb einer Liste.

Informationen zu virtuellen Objekten

Für jedes virtuelle Objekt gibt es im Step-Server ebenfalls einen Informationsblock vom Typ virt_object_info_t. Hierin wird jedem virtuellen Objekt der Ausdruck zugeordnet, der seinen Aufbau beschreibt. In Invarianten kann dieser String dann durch den Namen des virtuellen Objekts ersetzt werden, sofern es dem Resource-Manager bekannt gemacht wurde.

```
typedef struct { virt_object_info_t *next;
                 object_name_t object_name;
                 expression_string_t expression;
                 } virt_object_info_t;
```

ConTract-Informationen

Die Informationen, die ein Step-Server zu jedem ConTract wissen muß, bestehen aus dem Identifikator des ConTracts und den Namen der Resource-Manager, die für das jeweilige ConTract Aufgaben der Synchronisationsverwaltung übernommen haben. Die ConTract-Informationen werden in einer Liste von Informationsblöcken vom Typ `contract_info_t` gehalten. Hierin ist jeweils der Identifikator eines ConTracts sowie der Verweis auf eine Liste enthalten. Diese besteht aus Knoten des Typs `rm_list_node_t`. Jeder enthält den Bezeichner für einen Resource-Manager. Die ganze Liste ordnet somit einem ConTract alle beteiligten Resource-Manager zu.

```
typedef struct { rm_list_node_t *next;
                 rm_name_t rm_name;
                 } rm_list_node_t;

typedef struct { contract_info_t *next;
                 contract_id_t contract_id;
                 rm_list_node_t *rm_list;
                 } contract_info_t;
```

5.2.4 Algorithmen

SS_establish_invariant

SS_delete_invariant

Die Algorithmen dieser beiden Dienste sollen aufgrund ihrer Ähnlichkeit an dieser Stelle gemeinsam beschrieben werden. Zunächst wird die übergebenen Invariante in einzelne Prädikate zerlegt, die alle durch UND verknüpft sind. Jedes dieser Prädikate wird separat weiterverarbeitet. Diese Verarbeitung sieht folgendermaßen aus:

Zunächst wird überprüft, ob vorhandene virtuelle Objekte in einem Prädikat auftreten. Ist dies der Fall, so werden die beschreibenden Ausdrücke jeweils durch die Bezeichner für die virtuellen Objekte ersetzt. Anschließend wird festgestellt, ob in dem verbleibenden Prädikat Ausdrücke vorhanden sind, für die ein neues virtuelles Objekt etabliert werden sollte. Falls solche Ausdrücke auftreten muß das neue virtuelle Objekt bei den zuständigen Resource-Managern bekannt gemacht werden. Als letzte Aufgabe bleibt dem Dienst in den Prädikaten jeweils die enthaltenen Objekte festzustellen. Für jedes Objekt kann aus der Verwaltungsinformation der Step-Server der zuständige Resource-Manager entnommen werden. Diesen so ermittelten Resource-Manager müssen dann die Prädikate jeweils in den Diensten `RM_establish_predicate` oder `RM_delete_predicate` übergeben werden.

SS_end_of_contract

Dieser Dienst durchsucht zunächst die Liste der ConTract-Informationsblöcke. Wurde ein Block zu dem beendeten ConTract gefunden, so kann die Verweisliste auf Resource-Manager durchlaufen werden. Bei jedem der eingetragenen Resource-Manager wird dann der Dienst `RM_end_of_contract` aufgerufen.

5.2.5 Ergänzungen für eine vollständige Implementierung

Ähnlich wie in Abschnitt 5.1.5 beschrieben sind auch bei Step-Servern noch eine Reihe von Ergänzungen notwendig, um eine vollständige Implementierung der Synchronisationsverwaltung zu erreichen. Diese werden hier aufgeführt, wobei teilweise ein Verweis auf die entsprechenden Ausführungen für Resource-Manager ausreichend ist.

Integration aller Komponenten

Hier gilt wie bei den Resource-Managern, daß die Aufgaben weiterer APRICOTS-Komponenten berücksichtigt werden müssen. Außerdem ist das Kommunikationssystem von APRICOTS zu integrieren.

Zusätzliche Datentypen und Prädikate

Ein Step-Server operiert nur auf Ausdrücken über Objektnamen. Die Typen der Objekte sind hier ohne Bedeutung. Somit bedeutet auch die Ausweitung der Synchronisationsverwaltung auf andere Datentypen keinen zusätzlichen Implementierungsaufwand in den Step-Servern. Allerdings müssen hinzukommende Prädikate berücksichtigt werden. Sollen diese ebenfalls verarbeitet werden können, so ist eine Erweiterung der Dienste, die Invariantenausdrücke entgegennehmen, notwendig.

Speicherung der Verwaltungsinformation

Auch die Verwaltungsinformation der Step-Server muß für den Fall, daß die Komponente erneut gestartet werden soll, auf stabilem Speicher gesichert werden. Dies kann mit Hilfe folgender Tabellen geschehen:

object_info

ONR	object_name	rm_name
INTEGER	STRING	STRING

virtual_object_info

VONR	object_name	expression
INTEGER	STRING	STRING

contract_info

contract_id	rm_name
STRING	STRING

Wie in Abschnitt 5.1.5 lehnen sich die angegebenen Tabellen möglichst weitgehend an die Datenstrukturen der Step-Server an. Die Tabelle object_info hält für die einzelnen Objekte die zuständigen Resource-Manager fest. Tabelle virtual_object_info speichert zu den virtuellen Objekten, die ein Step-Server kennt, deren Aufbau. Die Tabelle contract_info schließlich kann zu jedem ConTract mehrere Einträge enthalten. Hierdurch werden jedem ConTract alle beteiligten Resource-Manager zugeordnet.

5.3 Kommunikation

Im Rahmen dieser Arbeit war eine Integration des Kommunikationssystems von APRICOTS nicht vorgesehen. Trotzdem soll die Kommunikation zwischen Step-Servern und Resource-Managern auch in der vorliegenden Implementierung möglich sein. Die verschiedenen Komponenten sollen hierbei als einzelne Prozesse auf demselben Rechner realisiert werden. Notwendig ist also die Auswahl der geeigneten Interprozeßkommunikation. Dies soll in diesem Abschnitt erfolgen.

5.3.1 Interprozeßkommunikation

Kommunikation zwischen Prozessen kann unter UNIX auf unterschiedliche Arten realisiert werden. Einige der Möglichkeiten sollen hier kurz skizziert werden, um die Basis für die bei der vorliegenden Implementierung getroffenen Entscheidungen bereitzustellen.

Shared memory

Mit Shared Memory wird ein Datenbereich bezeichnet, der von verschiedenen Prozessen gemeinsam genutzt werden kann. Mit entsprechenden System-Calls kann ein solcher Bereich eingerichtet bzw. auf einen eingerichteten Bereich Bezug genommen werden. Es handelt sich hierbei um eine effiziente Methode der Interprozeßkommunikation. Die Schwierigkeit liegt bei dieser Methode in der Festlegung einer geeigneten Synchronisierung, da alle Prozesse in der Regel Schreib- und Leseberechtigung für den gemeinsam genutzten Bereich erhalten.

Pipelines

Bei der Verwendung von Pipelines wird die Ausgabe eines Prozesses in einen Puffer geschoben, der nach dem FIFO-Prinzip von einem anderen Prozeß gelesen und als Eingabe verwendet wird. Soll dabei in beiden Richtungen kommuniziert werden, so müssen mehrere Pipelines aufgebaut werden.

Sockets

Sockets sind Kommunikationsendpunkte. Mit deren Hilfe wird einem Prozeß ein Puffer für die Kommunikation zugeordnet. Dieser kann von anderen Prozessen unter Angabe spezieller Adressen angesprochen werden. Es kann lokal auf einem Rechner oder innerhalb eines Netzes mit Hilfe von Netzwerksoftware mit Sockets kommuniziert werden. Hierbei ist auch bidirektionale Kommunikation möglich.

Für die Implementierung der Kommunikation zwischen Step-Servern und Resource-Managern werden Sockets gewählt. Die Gründe hierfür sind:

- Synchronisation muß nicht wie bei shared memory berücksichtigt werden.
- Es ist bidirektionale Kommunikation möglich.
- Zunächst kann die Kommunikation rechnerlokal gestaltet werden und dann später mit Hilfe derselben Mechanismen auf rechnerübergreifende Kommunikation ausgedehnt werden.

5.3.2 Realisierte Kommunikation

Die für die Kommunikation zwischen Step-Servern und Resource-Managern notwendigen Funktionen werden in ss_comm.c bzw. rm_comm.c realisiert. Hierin ist jeweils die Funktion prepare_comm enthalten, die die notwendigen Sockets einrichtet. Außerdem werden hier die Funktionen implementiert, die das eigentliche Senden und Empfangen von Daten ermöglichen. Details zu deren Aufbau sollen hier nicht diskutiert werden. In den Modulen rm_encode.c und rm_decode.c sowie den entsprechenden für die Step-Server sind Funktionen enthalten, die ebenfalls für die Kommunikation benötigt werden. Sie sorgen für eine Kodierung der Dienstparameter in Zeichenketten. Zwei Parameter werden jeweils durch ein Byte mit dem Dezimalwert 255 getrennt. Die so erhaltenen Zeichenketten können in Nachrichtenpuffer geschrieben oder von dort gelesen werden. Die Kodier- und Dekodierfunktionen können auch nach der Integration des APRICOTS-Kommunikationssystems weiterverwendet werden.

Kapitel 6

Ausblick

Diese Arbeit hat sich mit möglichen Architekturen für ein Laufzeitsystem beschäftigt, das die Synchronisation von ConTracts in APRICOTS übernimmt. Die verschiedenen Varianten wurden verglichen und die dezentrale Synchronisationsverwaltung in den einzelnen Resource-Managern als die geeignetste ausgewählt. Hauptargumente, die für diese Architektur sprechen sind deren Robustheit und der relativ geringe Kommunikationsaufwand zwischen den einzelnen Komponenten.

In der weiteren Arbeit wurde die prototypische Implementierung der Synchronisationsverwaltung beschrieben. Hier konnte gezeigt werden, wie die Verwaltung von Synchronisationsinformationen aussehen kann und wie die Dienste der verschiedenen Komponenten implementiert werden können.

Der erstellte Prototyp ist noch nicht in APRICOTS integriert. Bevor dies geschieht sollte zunächst die Leistung der vorliegenden Implementierung gemessen werden, die notwendigen Ergänzungen im Bezug auf weitere Datentypen und Prädikate erstellt werden, sowie das Kommunikationssystem von APRICOTS an die Stelle der Kommunikation via Sockets gestellt werden. Diese Aufgaben waren nicht Ziel dieser Arbeit und bleiben darum nachfolgenden Arbeiten überlassen.

Kapitel 7

Literaturverzeichnis

[BuRa90]: Burkert, Holger; Raichle, Bernd: Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext. Universität Stuttgart, Fakultät Informatik, IPVR, Studienarbeit Nr. 859, 1990.

[Eitl92]: Eitler, Markus: Eine graphische Oberfläche für die interaktive Steuerung langlebiger, verteilter Abläufe. Universität Stuttgart, IPVR, Diplomarbeit, 1992.

[Gray81]: Gray, Jim: The Transaction Concept: Virtues and Limitations. IN: Proc. VLDB, Cannes, 1981.

[GrRe93]: Gray, Jim; Reuter, Andreas: Transaction Processing: Concepts and Techniques. San Mateo: Morgan Kaufmann, 1993.

[HaSt91]: Harbison, Samuel P.; Steele, Guy L.: C: Ein Referenzhandbuch, Wolfram's, 1991.

[Laub92]: Laube, Dieter: Transaktionsorientierte Kommunikationsmechanismen für das verteilte Transaktionssystem APRICOTS. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 865, 1992.

[Reut89]: Reuter, Andreas: ConTracts: A Means for Extending Control beyond Transaction Boundaries. IN: Proc. 2nd Int. Workshop on High Performance Transaction Systems, Asimolar, 1989.

[Schü93]: Schützle, Alexander: Portierung / Reimplementierung eines transaktionsorientierten Kommunikationssystems. Universität Stuttgart, IPVR, Diplomarbeit, 1993.

[Schw95]: Schwarz, Holger: Synchronisationsprädikate in APRICOTS. Universität Stuttgart, Studienarbeit Nr. 1421, 1995.

[Seif95]: Seifert, Jens: Ein Programmierwerkzeug für APRICOTS. Universität Stuttgart, Studienarbeit Nr. 1422, 1995.

[Wäch]: Wächter, Helmut: ConTracts: A Means for Improving Reliability in Distributed Computing. IN: IEEE Spring CompCon, 1991.

[WäRe90]: Wächter, Helmut; Reuter, Andreas: Grundkonzepte und Realisierungsstrategien-des ConTract-Modells. In: Informatik Forschung und Entwicklung. (1990) Nr. 5, S. 202-212.

[WäRe91]: Wächter, Helmut; Reuter, Andreas: The ConTract Model. Universität Stuttgart, Fakultät Informatik, IPVR, Computer Science Report Nr. 8/91

Erklärung

Ich versichere, daß ich diese Arbeit selbständig verfaßt und nur die angegebenen Hilfsmittel verwendet habe.