Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Überblick

1.1 Einführung

Ein grundlegendes Problem in der Informatik ist die fehlerfreie Nutzung gemeinsamer Daten durch parallele Prozesse[1]. Seit Mitte der Siebziger Jahre steht dem Datenbankprogrammierer ein mächtiger Mechanismus zur Verfügung, mit dem dieses Problem elegant gelöst wird — die Datenbanktransaktion. Das DBMS verfügt über Funktionen, die den Programmierer weitgehend von der Verwaltung der Zugriffssperren auf Datenobjekte befreien. Ein Großteil der eigentlichen Synchronisationsaufgabe wird damit vom DBMS selbst übernommen.

Allerdings ist der Zugriff auf Datenobjekte im herkömmlichen Transaktionsmodell sehr restriktiv. Sperren werden bis zum Ende der Transaktion aufrechterhalten und blockieren parallel arbeitende Prozesse, die auf dieselben Daten zugreifen. Für Standardanwendungen, zum Beispiel bei der Kontenverwaltung der Banken, ist diese Einschränkung ohne weiteres zu verkraften. Bei der Verwaltung von komplexen Objekten, wie CAD–Daten, stößt das herkömmliche Transaktionskonzept jedoch schnell an seine Grenzen [Gray81].

ConTracts sind ein weit allgemeineres Konstrukt als die herkömmliche Datenbanktransaktion. Mit ihnen lassen sich langandauernde, verteilte Abläufe fehlertolerant modellieren und abwickeln.

Die Grundeinheiten der Ausführung eines ConTracts sind die sogenannten *Steps*. Ein Step enthält in sich keine Parallelität, das heißt er läßt sich als linearer Algorithmus beschreiben. Das *ConTract–Skript* hingegen beschreibt den Kontrollfluß des ConTracts — die Parallelität oder die serielle Abfolge mehrerer Steps.

Im Gegensatz zur Transaktion werden bei ConTracts nicht nur die Datenobjekte vor Zerstörung und Inkonsistenz geschützt. Der ConTract–Manager beherrscht auch die robuste Abwicklung des Kontrollflusses. Der gesamte Verarbeitungsprozeß wird vom ConTract–Manager

[1] [Freis87] bietet eine Übersicht und kurze Einführung in die Basismechanismen zur Synchronisation paralleler Prozesse.

kontrolliert und nach Systemausfällen wieder fortgeführt. Das System erfüllt diese Aufgabe weitgehend selbständig, sodaß der Programmierer von der detaillierten Formulierung einer Recovery–Strategie befreit wird.

1.2 Ziele der Arbeit

Am IPVR[2] wird derzeit das APRICOT–System entwickelt. Es handelt sich dabei um einen Prototyp zur Implementierung des ConTract–Modells. Im Rahmen des APRICOT–Projekts soll in dieser Studienarbeit das Modell des ConTract–Managers weiterentwickelt werden. Meine Arbeit baut dabei auf die Studienarbeit von Dieter Laube und Tran Van Dung ([LaDun90]) auf. Waren in dieser Arbeit der Interpreter und der Scheduler noch als eine einzige Komponente geplant, so soll jetzt eine Trennung in zwei Komponenten erfolgen. Der ConTract–Manager erfüllt nun allein die Aufgaben der Interpreters. Der Scheduler ist im ConTract–Processing–Monitor enthalten.

Im Zuge der Umstrukturierung innerhalb des Betriebsmodells werden die Aufgaben des ConTract–Managers neu definiert. Schwerpunkte dieser Arbeit sind dabei:

1. *Die Interpretation des ConTract–Skripts.*

   Der ConTract–Manager steuert den Kontrollfluß des ConTracts. Dazu muß er das ConTract–Skript kennen. Er interpretiert die Ablaufbeschreibung des Skripts und verwaltet die Aufträge für die Ausführungskomponenten des APRICOT–Systems.

2. *Die Bereitstellung einer einfachen Schnittstelle für den ConTract–Client*

   Der ConTract–Client ist eine Art Shell–Programm, über das der Benutzer auf das APRICOT–System zugreifen kann. Die Schnittstelle soll Kommandos bereitstellen, mit denen der Benutzer ConTracts starten und kontrollieren kann.

Neben der Definition der Aufgaben, sollte im Rahmen dieser Arbeit ein Prototyp des ConTract–Managers grob entworfen werden. Einen Teil der entworfenen Funktionen habe ich, parallel zur Studienarbeit, in der Programmiersprache C implementiert. Die Programmierung bezog sich dabei im wesentlichen auf:

- Datenstrukturen und Grundfunktionen zur Interpretation des ConTract–Skripts und
- Funktionen zur Ein- und Auslagerung eines ConTracts aus dem Hauptspeicher und zur externen Repräsentation in einer relationalen Datenbank.

1.3 Gliederung

Im folgenden Kapitel 2 erfolgt eine Einordnung des ConTract–Managers innerhalb des APRICOT–Systems. Die zentralen Aufgaben des ConTract–Managers werden vorgestellt und diskutiert. Kapitel 3 enthält eine detaillierte Besprechung derjenigen Aufgaben des ConTract–

[2] Institut für parallele und verteilte Höchstleistungsrechner, Universität Stuttgart.

Managers, die in dieser Arbeit schwerpunktmäßig behandelt werden (siehe *Ziele der Arbeit*). Der Entwurf des Prototyps für das APRICOT–System wird in Kapitel 4 in seinen wesentlichen Teilen dargelegt. Die wichtigsten Konzepte und Implementierungsentscheidungen werden hier erläutert. Danach wird in Kapitel 5 ein Implementierungskonzept für abstrakte Datentypen in C beschrieben. Kapitel 6 liefert noch einen kurzen Ausblick auf offene und angedachte Probleme im Umfeld des ConTract–Managers.

Kapitel 2

Die Rolle des ConTract–Managers im APRICOT–System

Zunächst einmal möchte ich beschreiben, welche grundlegende Anforderung der ConTract–Manager im APRICOT–System erfüllen soll. Diese Aufgabe wurde in [WäRe90] und [WäRe91] definiert. Wir werden sehen, daß sich einige weitere Aufgabengebiete des ConTract–Managers aus dieser Anfangsbedingung ableiten lassen.

Um eine einfache Modellvorstellung des ConTract–Managers und seiner Einordnung in das APRICOT–System zu entwickeln, wollen wir zuerst die beteiligten Komponenten betrachten.

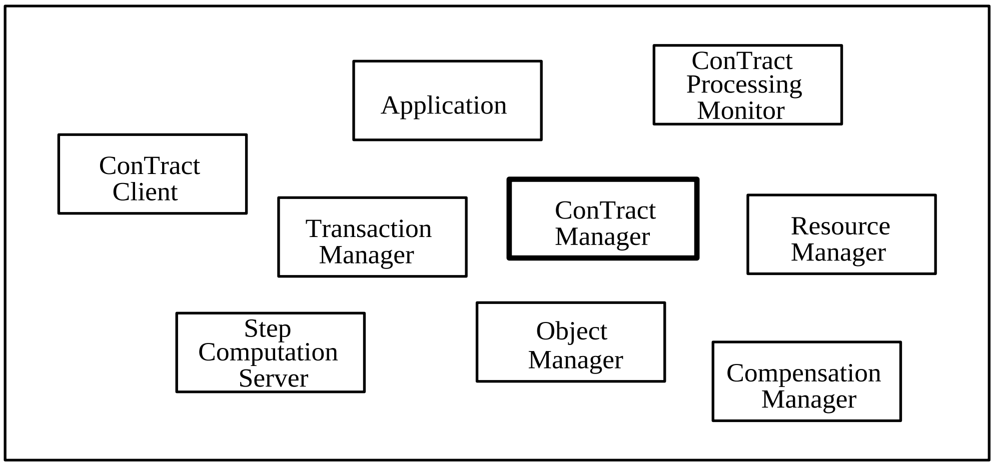


Abbildung 2.1: Die Komponenten des APRICOT–Systems

Die Aufteilung des gesamten APRICOT–Systems in einzelne Systemkomponenten und eine Beschreibung ihrer grundlegenden Aufgaben findet sich in [Burk91]. Abbildung 2.1 stellt die Komponenten zunächst schematisch dar, ohne ihre Beziehungen zu beschreiben.

Die zentrale Anforderung an den ConTract–Manager ist die Überwachung und Steuerung des Kontrollflusses bei der ConTract–Abwicklung. Dies führt in den weiteren Überlegungen auf folgende Fragen:

> Welche Konsequenzen ergeben sich aus dieser Anforderungsdefinition?
> Sind diese Konsequenzen zwingend?
> Welche Freiheitsgrade bleiben bei der Festlegung der Aufgabenbereiche?

Diese Aspekte möchte ich auf den nächsten Seiten eingehender beleuchten. Zum besseren Verständnis der Zusammenhänge, betrachten wir jedoch vorab die Aufgabengebiete des ConTract–Managers im Überblick.

1. *Oberste Instanz der Kontrollflußabwicklung.*

   Der ConTract–Manager steuert den Kontrollfluß eines ConTracts. Dies beinhaltet im wesentlichen die Interpretation des Skripts und damit die Synchronisation der einzelnen Steps.

2. *Fehlertoleranz bei der Kontrollflußabwicklung.*

   Die Kontrollflußabwicklung soll fehlertolerant verlaufen, das heißt der ConTract–Manager muß die Wiederaufsetzbarkeit bei Systemausfällen gewährleisten. Dazu muß er insbesondere alle Daten, die er zur Verwaltung des Kontrollflusses benötigt, ausfallsicher speichern.

3. *Realisierung der Benutzerschnittstelle.*

   Der Benutzer wird gegenüber dem System durch einen Prozeß, den sogenannten ConTract–Client, repräsentiert. Der ConTract–Manager stellt einige Kommandos für den ConTract–Client bereit, mit denen die Beauftragung, Steuerung und Überwachung des Kontrollflusses durch den Benutzer erfolgen kann.

4. *Verwaltung der Aufträge für die Ausführungskomponenten.*

   Als Teilaufgabe der Skriptinterpretation erzeugt, verwaltet und startet der ConTract–Manager die Aufträge für die Ausführungskomponenten. Er nimmt die Ausführungsergebnisse entgegen und wertet sie aus, um den Kontrollfluß zu steuern. Je nach Erfolg oder Mißerfolg der Ausführung, führt er die normale Abwicklung fort oder leitet eine Fehlerbehandlung ein.

5. *Abbildung von Steps und Step–Gruppen auf Transaktionen.*

   Der ConTract–Manager bildet Steps auf Transaktionen ab und definiert Transaktionsbeziehungen. Die Definition von Transaktionsbeziehungen wirkt sich auf die Kontrollflußabwicklung aus. Der ConTract–Manager kennt nun nicht mehr nur einzelne Steps als atomare Einheiten der ConTract–Abwicklung. Er muß Erfolg oder Mißerfolg der Ausführung ganzer Step–Gruppen handhaben.

6. *Migration von ConTracts.*

   Im ConTract–Modell ist die Migration langandauernder Abläufe im Rechnernetz vorgesehen. Dadurch kann der Benutzer zum Beispiel seinen Arbeitsplatz wechseln, ohne den

ConTract abbrechen und neu starten zu müssen. Der ConTract–Manager muß also in der Lage sein, die Verarbeitung eines ConTracts an einen anderen ConTract–Manager im Netz weiterzureichen.

Im nächsten Abschnitt wird die Hauptanforderung, die Kontrollflußabwicklung, eingehend beleuchtet. Außderdem werden die daraus entstehenden Konsequenzen betrachtet.

2.1 Der ConTract–Manager als oberste Instanz der Kontrollflußabwicklung

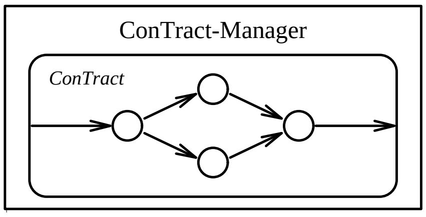


Abbildung 2.2: Der ConTract–Manager kontrolliert die Ausführung eines ConTracts

Der ConTract–Manager ist die oberste Instanz bei der Kontrolle der ConTract–Ausführung. Er startet die Ausführung, überwacht und steuert deren Ablauf und beendet den ConTract.

2.1.1 Welche Aufgaben umfaßt die Kontrollflußabwicklung?

Im *ConTract–Skript* wird bei der Programmierung der Kontrollfluß des ConTracts beschreiben. Die *Steps* sind in dieser Beschreibung die kleinsten Einheiten der Kontrollflußabwicklung. Jeder Step besteht aus einem rein sequentiell ablaufenden Programm.

Um den Kontrollfluß steuern zu können, muß der ConTract–Manager also das Skript kennen und er muß die Interpretation des Skripts beherrschen.

Die Ausführung läuft so lange, aus Sicht des Benutzers automatisch, bis der ConTract vollständig abgearbeitet ist. Ein zweiter Aspekt ergibt sich nun aus der Forderung, den Kontrollfluß *asynchron*, auf Wunsch des Benutzers, beeinflussen zu können. Der Benutzer greift über den ConTract–Client auf das APRICOT–System zu. Der ConTract–Manager muß also eine Schnittstelle zum ConTract–Client zur Verfügung stellen, über die die Ausführung des ConTracts kontrolliert werden kann. In Abschnitt 2.4, werde ich darauf näher eingehen. Zunächst wollen wir das Skript und seine Interpretation betrachten.

2.1.2 Das ConTract–Skript

An dieser Stelle möchte ich, zum besseren Verständnis der Aufgaben bei der Kontrollflußabwicklung, die Struktur des ConTract–Skripts erläutern.

Prädikat–Transitionsnetze

Die Synchronisation der Steps wird im Skript durch ein Prädikat–Transitionsnetz beschrieben. Diese Darstellung lehnt sich eng an die Petri–Netze[1] an.

- Die *Steps* des ConTract–Modells entsprechen den *Stellen* im Petri–Netz.
- Die *Transitionen* des Petri–Netzes werden im Prädikat–Transitionsnetz durch die sogenannten *Events* repräsentiert.
- Steps und Events erscheinen immer abwechselnd im Netz. Einem Step folgen $1 \ldots n$ Events und auf einen Event folgen $1 \ldots m$ Steps.

Als wesentliche Erweiterung bezüglich der Funktionalität der Petri–Netze, enthält das Prädikat–Transitionsnetz einen Mechanismus zur bedingten Steuerung des Kontrollflusses. Jeder Übergang von einem Step zu einem Event ist mit einer Bedingung belegt. Diese Bedingung sperrt einen Übergang oder gibt ihn frei — je nachdem welchen Wert sie während der Interpretation des Skripts annimmt.

Transaktionsbeziehungen

Eine weitere Möglichkeit, Abhängigkeiten zwischen Steps im Skript zu definieren, ist die Einrichtung von Transaktionsbeziehungen. Der ConTract–Manager teilt jedem Step eine eigene Transaktion zu. Durch die Angabe von Transaktionsbeziehungen ist es nun möglich, mehrere Steps aus logischer Sicht unter der Kontrolle einer einzigen Transaktion auszuführen. In diesem Fall ist garantiert, daß alle Steps einer Gruppe erfolgreich ablaufen und ihre Daten dauerhaft in der Datenbank zurücklassen. Scheitert auch nur ein Step, so werden alle anderen ebenfalls abgebrochen.

2.1.3 Weitere Anforderungen

Das Skript enthält Daten zur Synchronisation von Steps. Der ConTract–Manager entscheidet also bei der Interpretation des Skripts über die Reihenfolge, in der die einzelnen Steps gestartet werden sollen. Er bringt Steps zur Ausführung und verarbeitet ihr Ausführungsergebnis. Dieser Aspekt der Auftragsverwaltung wird in Abschnitt 2.5 im Detail besprochen.

[1] Erstmals beschrieben in [Petri62] — [Freis87] bietet eine leicht verständliche Einführung. Eine grundlegende Kenntnis der Petri–Netze möchte ich im weiteren voraussetzen.

Das Skript enthält jedoch außer den Daten, die zur Synchronisation nötig sind, noch weitere Angaben über die Transaktionsbeziehungen zwischen Steps. Der ConTract–Manager ist die einzige Komponente des APRICOT–Systems, die das ConTract–Skript kennt (siehe Abbildung 2.1). Er muß also auch alle notwendigen Maßnahmen zur Wahrung der Transaktionsbeziehungen treffen. In 2.6 wird beschrieben, welche Konsequenzen diese Forderung für das Modell des ConTract–Managers hat.

2.2 Fehlertoleranz und Fehlerbehandlung bei der Kontrollflußabwicklung

Das ConTract–Modell definiert bestimmte Zusicherungen, die für die Ausführung eines ConTracts gelten müssen.

1. Ein ConTract befindet sich nie in einem undefinierten Zustand. Der Anwendungsprogrammierer wird also davon entlastet, den Ablauf selbst vor Systemausfällen zu schützen und die Wiederaufsetzbarkeit seiner Anwendung zu gewährleisten.
   Im Abschnitt 2.8 wird gezeigt, welche Daten diesen Zustand des ConTracts beschreiben.
2. Ein ConTract läuft entweder vollständig und korrekt durch, oder er wird unterbrochen und alle nach außen hin sichtbaren Wirkungen werden rückgängig gemacht.

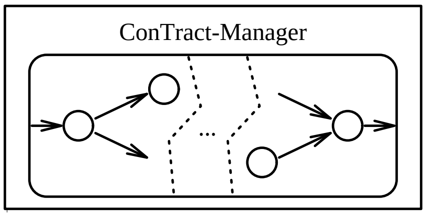


Abbildung 2.3: Ausfallsicherheit eines ConTracts

Ein wichtiger Aspekt der Datensicherheit ist die Speicherung der Ein- und Ausgabedaten der Steps. Nach außen hin erscheinen Steps wie Unterprogramme einer prozeduralen Programmiersprache. Ihre Aufrufschnittstelle enthält Ein- und Ausgabeparameter, die ihrerseits wieder sicher gespeichert werden müssen. Diese Speicherung erfolgt im *Kontextsystem*, einer Komponente des APRICOT–Systems, die Ausprägungen von Datenobjekten sicher speichert und verwaltet. Eine detaillierte Beschreibung befindet sich in [BuRai90].

2.2.1 Recovery

Aus der 1. Zusicherung für ConTracts ergibt sich die Anforderung, Systemausfälle zu jedem Zeitpunkt so handhaben zu können, daß die Abwicklung des ConTracts weitergeführt werden kann. Der ConTract–Manager muß dazu in der Lage sein alle Daten, die den Zustand eines ConTracts beschreiben, sicher zu speichern.

Der ConTract–Manager selbst muß, nach einem Systemausfall, vom APRICOT–System automatisch wieder gestartet werden. Hat der ConTract–Manager einmal einen Auftrag zur Bearbeitung eines ConTracts übernommen, so führt er diesen auch zu Ende — selbst wenn er dabei durch Systemausfälle unterbrochen wird. Den Neustart des ConTract–Managers nach einem Systemausfall übernimmt eine zentrale Prozeßverwaltungseinheit des APRICOT–Systems. Die Beschreibung der dazu notwendigen Verfahren ist Teil der Arbeit von Holger Burkert ([Burk91]).

Zur Realisierung der Recovery wird die Ausführung eines ConTracts in atomare Einheiten aufgegliedert. Die Größe dieser Einheiten bemißt sich daran, wieviele Aktionen bei einem Systemausfall maximal verlorengehen dürfen. Im Normalbetrieb muß der ConTract–Manager, jeweils am Ende einer solchen Ausführungseinheit, folgende Maßnahmen durchführen:

1. Alle Veränderungen der internen Zustandsdaten werden in einem Logging–System protokolliert.
2. Die eingelesenen und bearbeiteten Nachrichten anderer Komponenten werden quittiert.
3. Alle Nachrichten, die der ConTract–Manager während der Verarbeitung erzeugt hat, werden an die anderen Komponenten verschickt.

Solange der ConTract–Manager keine Daten ausgibt, bleiben seine internen Zustandsänderungen vor den anderen Komponenten verborgen. Sobald er jedoch Ausgaben durchführt, müssen diese Daten dauerhaft protokolliert werden. Die Ausgabe und ihre Protokollierung müssen zusammen und ununterbrechbar durchgeführt werden. Schickt der ConTract–Manager zum Beispiel eine Nachricht ab, ohne dies zu protokollieren, so würde er die Ausgabe nach einem Systemausfall unter Umständen wiederholen. Die angesprochene Komponente würde die Nachricht dann in doppelter Ausführung empfangen.

Nach einem Systemausfall kann der ConTract–Manager, anhand des Protokolls, seinen internen Zustand vor dem Ausfall wiederherstellen. Der wiederhergestellte Zustand des ConTract–Managers entspricht dabei demjenigen Zustand, der zuletzt protokolliert wurde — dies ist genau das Ende der zuletzt vollständig durchgeführten Ausführungseinheit. Da die Ausgaben des ConTract–Managers und die Protokollierung ununterbrechbar gemeinsam durchgeführt werden, ist außerdem garantiert, daß der interne Zustand des ConTract–Managers nun dem aktuellen Zustand des APRICOT–Systems entspricht.

2.2.2 Konfliktbehandlung

Herkömmliche Datenbanktransaktionen halten Sperren auf benutzten Datenobjekten bis zum COMMIT oder ROLLBACK, aufrecht. Transaktionen können sich deshalb gegenseitig bis zum Ende ihrer Durchführung verzögern, wenn sie auf gemeinsamen Daten arbeiten. Gerade diese Eigenschaft macht sie ungeeignet für lang andauernde Abläufe. Im ConTract–Modell wurde ein anderer Mechanismus für die Verwaltung konkurrierender Zugriffe gewählt. Einzelne Steps oder Step–Gruppen, die in ihrer Ausführung relativ kurz sind, laufen unter Transaktionsschutz. Zwischen diesen Einzeltransaktionen eines ConTracts werden keine Sperren auf den verwendeten Datenobjekten gehalten. Andere ConTracts oder auch Anwendungsprogramme, die außerhalb des APRICOT–Systems laufen, können auf die Daten zugreifen und sie beliebig verändern[2]. Die Parallelität der Zugriffe wird so wesentlich erhöht. Konflikte beim Zugriff auf Datenobjekte werden durch das *Invariantenkonzept* kontrolliert.

Die *Ausgangsinvariante* eines Steps beschreibt den Zustand der bearbeiteten Daten, sofern er für den weiteren, korrekten Ablauf des ConTracts benötigt wird. Nachfolgende Steps können in ihrer *Eingangsinvariante* auf solche Zustandbeschreibungen bezug nehmen. Die Eingangsinvariante definiert Anforderungen an den Datenbestand, die für die korrekte Ausführung des Steps erfüllt sein müssen.

Betrachten wir zur Veranschaulichung das Beispiel einer Flugbuchung. In einer Abteilung eines Unternehmens steht ein bestimmtes Reisebudget zur Verfügung. Ein Step würde hier vor einer Abbuchung in seiner Eingangsinvarianten prüfen, ob das Budget für die Buchung ausreicht. Scheitert die Eingangsinvariante, so wird eine Konfliktbehandlung durchgeführt, die im Skript für den jeweiligen Step definiert ist. Die Konfliktbehandlung führt zur Ausführung eines speziellen Fehlerbehandlungszweigs. Im Beispiel könnte dies ein Step sein, der das Reisebudget erhöht, um die Flugbuchung doch noch durchführen zu können.

Die Auswertung der Invarianten ist nicht Aufgabe des ConTract–Managers. Sie wird von tieferliegenden Schichten des APRICOT–Systems übernommen. Scheitert die Eingangsinvariante, so erhält der ConTract–Manager eine entsprechende Mitteilung bei der Rückkehr des Steps. Er muß dann die Konfliktbehandlung auslösen.

Unter Umständen ist für einen Step keine Konfliktbehandlung definiert, oder die Konfliktbehandlung scheitert. In beiden Fällen kann der ConTract nicht erfolgreich zu Ende gebracht werden. Er wird vom ConTract–Manager abgebrochen und alle nach außen hin sichtbaren Wirkungen werden rückgängig gemacht (Zusicherung 2) — der ConTract wird kompensiert.

2.3 Kompensation

Mit der 2. Zusicherung für ConTracts sind noch weitere Anforderungen an den ConTract–Manager verbunden. In bestimmten Fällen kann es vorkommen, daß die Ausführung eines ConTracts nicht fortgesetzt werden kann. Wie wir im vorangegangenen Abschnitt gesehen haben tritt dieser Fall unter anderem beim Scheitern einer Konfliktbehandlung ein. Die Ausführung eines ConTracts scheitert auch dann, wenn ein Step nicht ausgeführt werden kann. Der ConTract–Processing–Monitor, der als Scheduler im APRICOT–System fungiert, könnte zum Beispiel feststellen, daß eine benötigte Ausführungskomponente nicht verfügbar ist.

Betrachten wir zum Vergleich die herkömmliche Datenbanktransaktion. Das Anwendungsprogramm würde im vorliegenden Fall einfach einen ROLLBACK veranlassen um alle Ände-

[2] Allerdings dürfen Datenobjekte nicht gelöscht werden ([WäRe91]).

rungen des Datenbestandes zurückzunehmen. Im ConTract–Modell ist dies nicht ohne weiteres möglich, da die Datenobjekte nicht dauerhaft gesperrt waren. Der ConTract–Manager muß davon ausgehen, daß parallel arbeitende Prozesse die Daten ebenfalls im Zugriff hatten. Ist die Transaktion eines einzelnen Steps oder einer Step–Gruppe einmal durch COMMIT beendet, so sind alle Änderungen im Sinne des ACID–Prinzips tatsächlich dauerhaft durchgeführt. Der Datenbestand kann nicht einfach durch Überschreiben mit *before images* rekonstruiert werden. Um alle Wirkungen des ConTracts rückgängig zu machen, sind Kompensationsaktionen durchzuführen. Für jeden abgearbeiteten Step wird quasi eine Umkehroperation gestartet. Die Zuordnung der Kompensationsaktionen zu den Steps ist im Skript beschrieben.

2.3.1 Verfahren zur Kompensation von ConTract–Skripten

Im folgenden möchte ich drei Verfahren zur Kompensation von ConTracts vorstellen und diskutieren. Allen drei Verfahren ist gemeinsam, daß für jeden Step ein zugehöriger Kompensations–Step existiert. Dieser Ansatz wird in [WäRe91] ausführlich besprochen.

1. *Sequentielle Kompensation von Step–Gruppen.*

   Während der normalen Ausführung des ConTracts wird die zeitliche Reihenfolge der Step–Ausführungen[3] in einem History–Log protokolliert. Gruppen von Steps, die gleichzeitig (parallel) zur Ausführung gebracht werden, haben dieselbe Startzeit. Bei der Kompensation können diese Gruppen anhand der Startzeit wieder rekonstruiert und die einzelnen Steps parallel kompensiert werden.

   Wichtig ist dabei, daß alle Steps einer Gruppe vollständig kompensiert sein müssen bevor die Kompensation der nächsten Gruppe angestoßen wird. Wird diese Einschränkung nicht beachtet, so werden unter Umständen Steps parallel ausgeführt, die im Skript als zwingend sequentiell vermerkt sind.

2. *Echte Rückwärts–Interpretation des Skripts.*

   Die Kompensation orientiert sich an der Struktur des Skripts, das heißt das Skript wird in umgekehrter Reihenfolge durchlaufen.

3. *Kompensation aufgrund zeitlicher Abhängigkeiten.*

   Während der normalen ConTract–Ausführung werden alle zeitlichen Abhängigkeiten der Step–Ausführungen aufgezeichnet. Löst das Ende einer Step–Ausführung die Ausführung eines anderen Steps aus, so wird diese Abhängigkeit in einem History–Log vermerkt. Damit sind alle notwendigen Synchronisationsbedingungen für die Kompensation gegeben. Bei der Kompensation müssen die aufgezeichneten Abhängigkeiten nur in umgekehrter Reihenfolge durchlaufen werden.

[3] In verteilten Systemen existiert (meist) keine einheitliche Systemzeit. Da jedoch nur die Reihenfolge der Ausführungen (relative Startzeit) und nicht ihre absolute Startzeit von Interesse ist, reicht ein fortlaufender Zähler aus.

2.3.2 Bewertung der Verfahren

Die sequentielle Kompensation von Step–Gruppen ist durch ein Paket einfacher Kompensationsfunktionen realisierbar, da keine aufwendigen Auswertungen durchgeführt werden müssen. Allerdings ist der Parallelitätsgrad nicht optimal: Die Step–Gruppen ergeben sich recht zufällig aufgrund der Synchronisationsbedingungen im Skript und des Laufzeitverhaltens der vorangegangenen Steps. Die erreichbare Parallelität der Kompensation von Step–Gruppen ist deshalb meist weit geringer als die, aufgrund des Skripts, maximal zulässige.

Die echte Rückwärts–Interpretation des Skripts ereicht bei der Kompensation optimale Parallelität, da alle Synchronisationsbedingungen bekannt sind. Im Vergleich zu den anderen Verfahren ist sie jedoch sehr komplex. Das Verfahren ist deshalb für eine prototypische Realisierung ungeeignet. Außerdem wäre noch zu klären welche Daten im History–Log gespeichert werden müßten, um die Rückwärts–Interpretation zu ermöglichen.

Bei der Kompensation aufgrund zeitlicher Abhängigkeiten ist die erreichbare Parallelität optimal — alle Synchronisationsbedingungen für die Kompensation sind bekannt. Der Aufwand ist außerdem kaum größer als bei der Kompensation von Step–Gruppen. Die notwendigen Abfragen und Auswertungen des History–Logs können in SQL schnell und einfach implementiert werden. Ein Einsatz dieses Verfahrens zu Kompensation im Prototypsystem APRICOTS erscheint deshalb sinnvoll.

2.3.3 Der Compensation–Manager

Der Compensation–Manager übernimmt die Aufgabe der Kompensation im APRICOT–System. Die Entscheidung darüber, ob und wann die Kompensation angestoßen wird trifft jedoch der ConTract–Manager, da er die oberste Instanz der Kontrollflußabwicklung ist.

Zur Durchführung der Kompensation benötig der Compensation–Manager, je nach gewähltem Kompensationsverfahren, folgende Daten:

- Den Kontext der abgearbeiteten Steps, um die Eingabedaten für die Kompensations–Steps zur Verfügung stellen zu können.
- Den History–Log zur Rekonstruktion der zeitlichen Folge der Step–Ausführungen.
- Das ConTract–Skript, um die Daten der Kompensations–Steps zu erhalten.

ConTract–Manager und Compensation–Manager müssen nicht strikt voneinander getrennt werden. Im Prototyp wäre eine einheitliche Komponente zur Abdeckung beider Aufgabenbereiche durchaus denkbar.

2.4 Die Schnittstelle zum ConTract–Client

Das APRICOT–System ist aus Sicht des ConTract–Client ein Betriebssystemdienst. Er wird verwendet, um ConTracts zu starten und zuverlässig abzuwickeln. Wie wir in Abschnitt 2.2 gesehen haben, übernimmt das APRICOT–System jedoch sehr viel weitreichendere Zusicherungen, als die Prozeßverwaltung herkömmlicher Betriebssysteme. Die Überlegungen zur Kontrollflußabwicklung haben gezeigt, daß auch die Schnittstelle des ConTract–Clients zum System in den Aufgabenbereich des ConTract–Managers fällt. Betrachten wir nun die einzelnen Kommandos, die der ConTract–Client zur Kontrolle der Ausführung eines ConTracts benötigt.

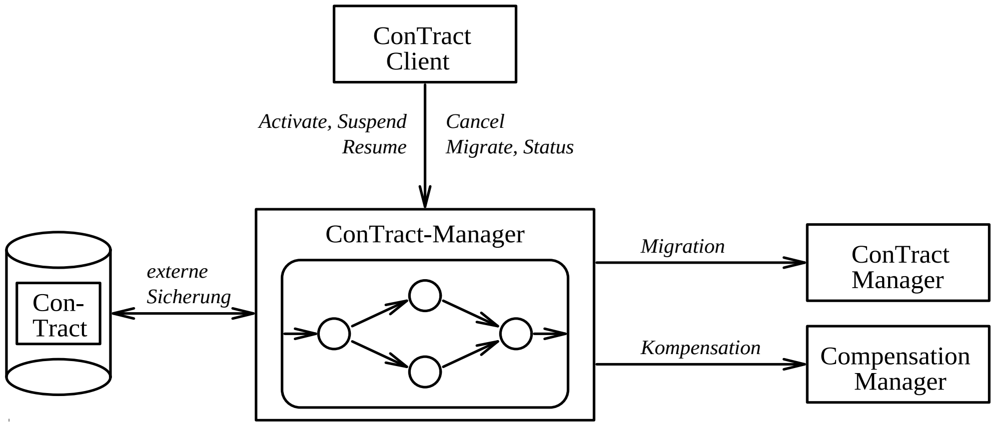


Abbildung 2.4: Die Schnittstelle zum ConTract–Client

1. ACTIVATE — Starten eines ConTracts.

   Der ConTract–Client kann mit dieser Funktion einen neuen ConTract starten. Er übergibt dazu den symbolischen Namen des ConTracts, der bei der Programmierung des Skripts definiert wurde.

2. SUSPEND — Unterbrechung eines ConTracts.

   Mit diesem Kommando kann der ConTract–Client die Ausführung eines ConTracts unterbrechen. Die Unterbrechung kann beliebig lange andauern. Der ConTract–Client kann die Ausführung des ConTracts jederzeit wieder mit dem RESUME–Kommando fortsetzen.

3. RESUME — Wiederanfahren der ConTract–Verarbeitung nach einer Unterbrechung.

   Das RESUME–Kommando setzt voraus, daß der ConTract zuvor mit SUSPEND unterbrochen wurde. Die Ausführung des ConTracts wird an der unterbrochenen Stelle fortgesetzt.

4. CANCEL — Abbruch eines ConTracts.

   Die Ausführung des ConTracts wird abgebrochen und der ConTract–Manager leitet die Kompensation ein. Alle nach außen hin sichtbaren Wirkungen des ConTracts werden dadurch rückgängig gemacht.

5. MIGRATE — Wanderung eines ConTracts zu einem anderen ConTract–Manager.

   Die Migration, das heißt die Übertragung der Verarbeitung an einen anderen Rechnerknoten, ist für langandauernde Abläufe, wie sie durch ConTracts realisiert werden, sehr wichtig. Die Wartung eines Rechners oder auch der Wechsel des Arbeitsplatzes eines Benutzers, erfordern einen Mechanismus, mit dem ConTracts auf andere Rechnerknoten übertragen werden können, ohne daß der Ablauf neu gestartet werden muß.

   Der Migrationsbefehl veranlaßt den ConTract–Manager den ConTract an den Empfänger der Migration zu übertragen. Dieser Empfänger ist ebenfalls ein ConTract–Manager, er befindet sich jedoch auf einem anderen Knoten des Rechnernetzes.

6. STATUS — Abfrage des ConTract–Zustands.

   Der ConTract–Client kann beim ConTract–Manager jederzeit Daten zum aktuellen Zustand eines ConTracts erfragen. Dazu gehören zum Beispiel die gerade aktiven Steps und statistische Daten zum bisherigen Ablauf des ConTracts.

2.5 Verwaltung der Aufträge für die Ausführungskomponenten

Der ConTract–Manager verwaltet und interpretiert das ConTract–Skript. Das beinhaltet auch die Aufgabe, Step–Ausführungen anzustoßen und deren Ergebnis auszuwerten. Ausführungsergebnisse der Steps lassen sich dabei in zwei Klassen einteilen:

1. *Ausgabedaten des Steps, die im Kontextsystem gespeichert werden.*

   Die Übergänge zwischen Steps und Events im Prädikat–Transitionsnetz sind mit Bedingungen belegt[4]. Diese Bedingungen beziehen sich auf die Ausgabedaten der Steps. Der ConTract–Manager muß diese Bedingungen auswerten. Er kann so den Kontrollfluß, in Abhängigkeit von den Ausführungsergebnissen der Steps, steuern.

2. *Ein Returncode, der über Erfolg oder Mißerfolg der Stepausführung Auskunft gibt.*

   Konnte die Ausführung eines Steps erfolgreich durchgeführt werden, so durchwandert der ConTract–Manager das Prädikat–Transitionsnetz, ausgehend vom beendeten Step. Er sammelt dabei alle Steps, die aufgrund der Skript–Information direkte Nachfolger des beendeten Steps sind. Diese Steps werden dann zur Ausführung gebracht.

   Tritt bei der Ausführung eines Steps ein Fehler auf, so leitet der ConTract–Manager eine entsprechende Fehlerbehandlung ein. Der Returncode gibt zum Beispiel Auskunft darüber, ob die Eingangsinvariante des Steps gescheitert ist. In diesem Fall wird eine Konfliktbehandlung durchgeführt. Es ist auch möglich, daß ein Step gar nicht ausgeführt werden konnte, weil die entsprechende Ausführungskomponente nicht verfügbar war. Tritt dieser Fall ein, so muß der ConTract–Manager die Kompensation einleiten.

[4]Siehe Beschreibung der Prädikat–Transitionsnetze auf Seite 7.

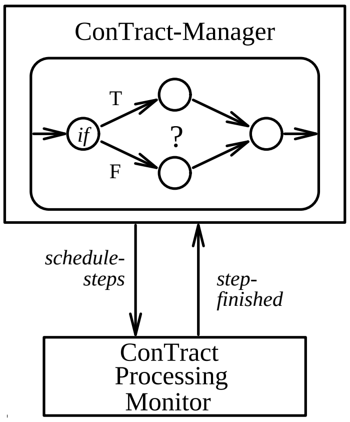


Abbildung 2.5: Auftragsverwaltung für die Ausführungskomponenten — Entscheidungen über den weiteren Ablauf des ConTracts hängen vom Ergebnis der Step–Ausführung ab.

Im APRICOT–System gibt es verschiedene Komponenten, die für die eigentliche Ausführung der Steps zuständig sind. Im einzelnen sind dies *Step–Computation–Server*, *Resource–Manager* und *Object–Manager*. Die Ausführungskomponenten stellen Systemdienste, wie zum Beispiel einen Flugbuchungsdienst, zur Verfügung. Eine genaue Definition ihrer Aufgaben ist in [Burk91] zu finden.

Der ConTract–Manager entscheidet aufgrund des ConTract–Skripts, welche Steps gestartet werden sollen. Er erstellt Aufträge für die Ausführungskomponenten und reicht sie an den ConTract–Processing–Monitor weiter. Der ConTract–Processing–Monitor fungiert im APRICOT–System als Scheduler. Er teilt die Aufträge an die Ausführungskomponenten zu und nimmt die Ergebnisse ihrer Ausführung entgegen, um sie an den ConTract–Manager weiterzuleiten. Erhält nun der ConTract–Manager vom ConTract–Processing–Monitor eine solche Rückmeldung, so prüft er, welche Steps nun, aufgrund der Synchronisationsdaten im Skript, gestartet werden können. Für jeden dieser Steps wird dann erneut ein Ausführungsauftrag an den ConTract–Processing–Monitor geschickt.

2.6 Abbildung von Steps und Step–Gruppen auf Transaktionen

Jeder Step im APRICOT–System läuft innerhalb einer Transaktion. Im Skript sind darüber explizit keine Angaben gemacht. Das Skript enthält nur Informationen über die Beziehungen zwischen den Transaktionen der einzelnen Steps.

Bevor der ConTract–Manager eine Gruppe von Steps zur Ausführung an den ConTract–Processing–Monitor schickt, teilt er jedem Step einen eigenen *Transaktions–ID*, auch kurz *gtrid*[5] genannt, zu. Die entsprechenden Transaktions–IDs werden vom Transaction–Manager erzeugt

[5]Global TRansaktion ID.

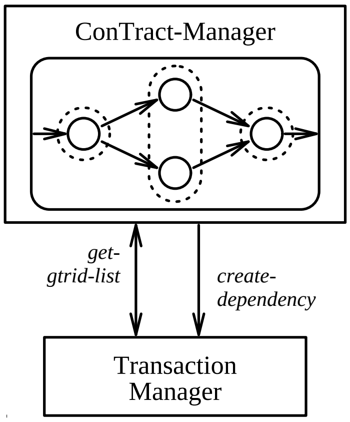


Abbildung 2.6: Die Schnittstelle zum Transaktionssystem

und können vom ConTract–Manager angefordert werden. Die gtrid wird in der Datenstruktur des Ausführungsauftrags abgelegt und gelangt so zu den Ausführungskomponenten. Diese wiederum starten die Ausführung der Steps unter dieser Transaktions–ID.

Nachdem der ConTract–Manager jedem Ausführungsauftrag eine eigene Transaktions–ID zugeteilt hat, prüft er, ob im Skript explizite Beziehungen zwischen Transaktionen einzelner Steps definiert sind. Transaktionsbeziehungen werden dann, unter Angabe der beteiligten Transaktions–IDs, beim Transaction–Manager vereinbart.

Betrachten wir als Beispiel die Vereinbarung einer COMMIT–Beziehung zwischen den Transaktionen zweier Steps A und B. Die Beziehung führt dazu, daß entweder beide Transaktionen mit COMMIT beendet oder mit ROLLBACK abgebrochen werden. Angenommen, Step A möchte einen COMMIT durchführen, Step B jedoch einen ROLLBACK. In diesem Fall kommt der Transaction–Manager dem COMMIT–Wunsch von A nicht nach; die Transaktion wird aufgrund der COMMIT–Beziehung zurückgesetzt. Die Auswertung der Transaktionsbeziehungen, das heißt die Auswirkungen zum Beispiel eines Abbruchs einer Transaktion, die mit anderen Transaktionen in Beziehung steht, sind nicht mehr Teil dieser Arbeit (siehe dazu [Lutz91] und [Schü91]).

Der ConTract–Manager muß zusätzlich zur Rückmeldung des ConTract–Processing–Monitors auch den Zustand der Transaktion eines beendeten Steps auswerten, um den Kontrollfluß zu steuern.

2.7 Migration von ConTracts

Wie die Besprechung der Schnittstelle zum ConTract–Client gezeigt hat, benötigt der Benutzer einen Mechanismus, der ihm erlaubt, die Verarbeitung eines ConTracts auf einen anderen Rechnerknoten zu übertragen. Der Ablauf soll dabei nicht abgebrochen, sondern auf dem Zielrechner weitergeführt werden.

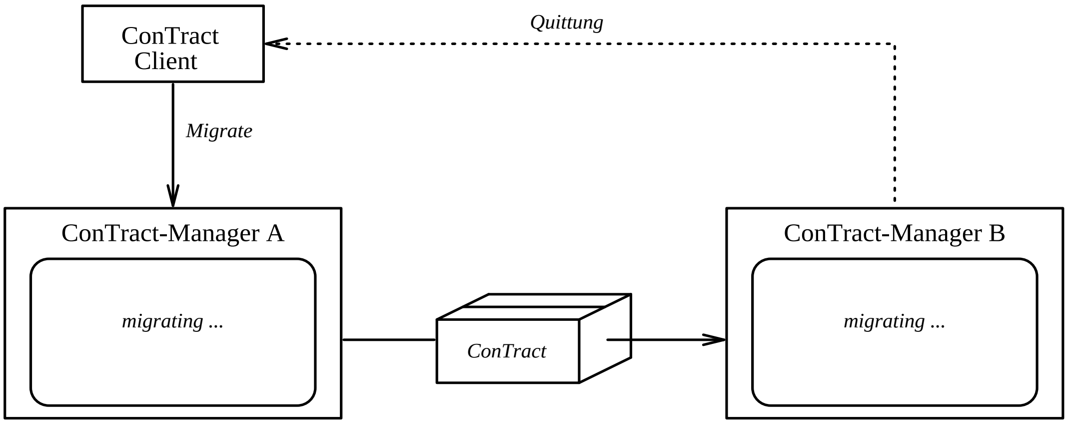


Abbildung 2.7: Migration eines ConTracts

An der Migration sind immer zwei ConTract–Manager beteiligt, jeweils einer als Sender und Empfänger. Der ConTract–Client teilt der Prozeßverwaltungseinheit seinen Migrationswunsch mit. Er erhält daraufhin den Komponenten–ID[6] des ConTract–Managers, der bei der Migration als Empfänger fungiert.

Der ConTract–Manager muß nun in der Lage sein, beide Rollen einzunehmen. Er muß die Verarbeitung eines ConTracts unterbrechen, alle seine Daten in einem geeigneten Datenformat verpacken und an den Empfänger der Migration übertragen. Außerdem muß er — falls er bei der Migration als Empfänger fungiert — ein solches Datenpaket empfangen und denjenigen Zustand des ConTracts wiederherstellen, in dem sich dieser vor seiner Migration befand. Die eigentliche Übertragung der Daten übernimmt im APRICOT–Prototyp das Kommunikationssystem[7]. Die Kommunikationssystem garantiert dabei auch, daß Nachrichten bei Systemausfällen nicht verlorengehen ([Burk91]).

2.8 Welche Daten beschreiben den Zustand eines ConTracts?

In den bisherigen Überlegung wurde immer wieder von den Zustandsdaten eines ConTracts gesprochen. Diese Daten enthalten eine vollständige Beschreibung des Ablaufs und seines momentanen Zustand. Bei der Migration werden die Zustandsdaten auf einen anderen Rechnerknoten übertragen. Auch bei der Suspendierung benötigt der ConTract–Manager die gesamten Zustandsdaten — in diesem Fall, um eine dauerhafte Repräsentation des ConTracts auf einem externen Speichermedium zu erzeugen. Betrachten wird nun diese Zustandsdaten aus logischer Sicht.

1. *Cid*

[6]Jeder Prozeß im APRICOT–System erhält eine Komponenten–ID, die global eindeutig ist.

[7]Siehe Abschnitt 2.9 auf Seite 19

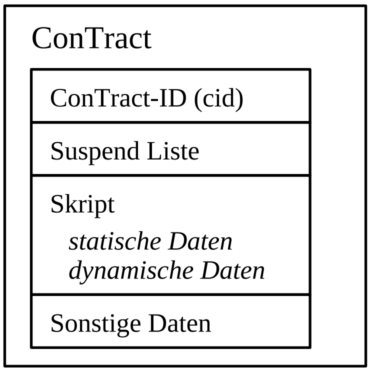


Abbildung 2.8: Die Daten eines ConTracts

Der cid[8] dient zur Identifikation des ConTracts im APRICOT–System. Er ist global eindeutig.

2. *Suspend Liste*

   Bei der Unterbrechung des ConTracts fallen Daten an, die zur späteren Weiterverarbeitung benötigt werden. Es handelt sich dabei um die Aufträge, die bei der Unterbrechung gerade aktiv waren. Eine anschauliche Beschreibung dieser Daten bietet der Begriff *Aktivitätslinie*. Damit ist die Menge derjenigen Punkte im Prädikat–Transitionsnetz gemeint, die gerade aktiv sind; das sind gerade die im Moment aktiven Steps.

3. *Statische Daten des Skripts*

   Alle Daten des Skripts, die die Struktur des Prädikat–Transitionsnetzes beschreiben, sind statisch — sie verändern sich während der Ausführung des ConTracts nicht. Darunter fallen Steps, Events, die Kanten zwischen Steps und Events[9], Informationen zur Kompensation des ConTracts und die Definition der Transaktionsbeziehungen.

4. *Dynamische Daten des Skripts*

   Hierunter fallen alle Daten, die während der Ausführung des ConTracts erzeugt oder verändert werden. Dies sind zum Beispiel Daten zur Verwaltung der Aufträge für die Ausführungskomponenten. Nur die im Moment aktiven Aufträge werden in den Zustandsdaten gespeichert. Bereits erfolgreich ausgeführte Aufträge müssen in den Zustandsdaten des ConTracts nicht mehr gespeichert werden.

5. *Sonstige ConTract–Daten*

   Hierunter fallen Daten, die auf der Ebene der Anforderungen nicht interessant sind, wie statistische Daten, interne Zustandsvariablen usw.

[8] ConTract–ID

[9] Die Kanten des Prädikat–Transitionsnetzes.

2.9 Die Kommunikation zwischen den Komponenten des APRICOT–Systems

Für die Abwicklung des Datenaustauschs zwischen den Komponenten ist im APRICOT–System ein spezielles Kommunikationssystem zuständig [Burk91]. Das Kommunikationssystem übernimmt dabei drei wesentliche Aufgaben:

1. *Die Pufferung von Nachrichten*

   Die Komponenten können unter Umständen viele Nachrichten innerhalb kurzer Zeit erhalten. Dabei darf keine der eingetroffenen Nachrichten verlorengehen. Das Kommunikationssystem entbindet die Komponenten von der Aufgabe die empfangenen Nachrichten zu puffern. Jeder Komponente wird eine Warteschlange zur Verfügung gestellt, aus der die Nachrichten zur Verarbeitung entnommen werden können.

2. *Die fehlertolerante Übertragung von Nachrichten.*

   Hat das Kommunikationssystem eine Nachricht entgegengenommen, so garaniert es auch die Ablieferung der Nachricht beim Empfänger. Insbesondere ist gewährleistet, daß Nachrichten bei Systemausfällen nicht verlorengehen. Dies befreit die einzelnen Komponenten davon, ihren Nachrichtenaustausch selbst vor Systemausfällen schützen zu müssen.

3. *Die Zuordnung der Komponenten zueinander.*

   Die einzelnen Komponenten kommunizieren, aus ihrer Sicht, mit *Klassen* von Komponenten, nicht mit einzelnen Prozessen. Fordert der ConTract–Manager zum Beispiel den ConTract–Processing–Monitor auf, Aufträge an Ausführungskomponenten zuzuteilen, so schickt er diese Nachricht an die *Klasse der ConTract–Processing–Monitore*. Existieren mehrere ConTract–Processing–Monitore im System, so interessiert es aus Sicht des ConTract–Managers nicht, welcher ConTract–Processing–Monitor diese Nachricht nun tatsächlich erhält. Allerdings muß gewährleistet sein, daß die Rückmeldungen des ConTract–Processing–Monitors wieder zu demjenigen ConTract–Manager gelangen, der den zugehörigen Auftrag erteilt hat.

2.10 Zusammenfassung

Wir haben nun alle wesentlichen Aufgaben des ConTract–Managers betrachtet. Wie wir gesehen haben hängen diese Aufgaben teilweise miteinander zusammen; sie bedingen sich oft gegenseitig. Abbildung 2.9 stellt die Zusammenhänge noch einmal graphisch dar. Wenn für den ConTract–Manager als zentrale Anforderung vorausgesetzt wird, daß er die fehlertolerante Kontrollflußabwicklung beherrscht, so führen weitergehende Überlegungen zu den in Abbildung 2.9 genannten Aufgabengebieten.

Aufgrund der beschriebenen Anforderungen ist es nun möglich, die Rolle des ConTract–Managers im APRICOT–System zu definieren. Im folgenden möchte ich nochmals die Beziehungen des ConTract–Managers zu den anderen Komponenten im Überblick darstellen.

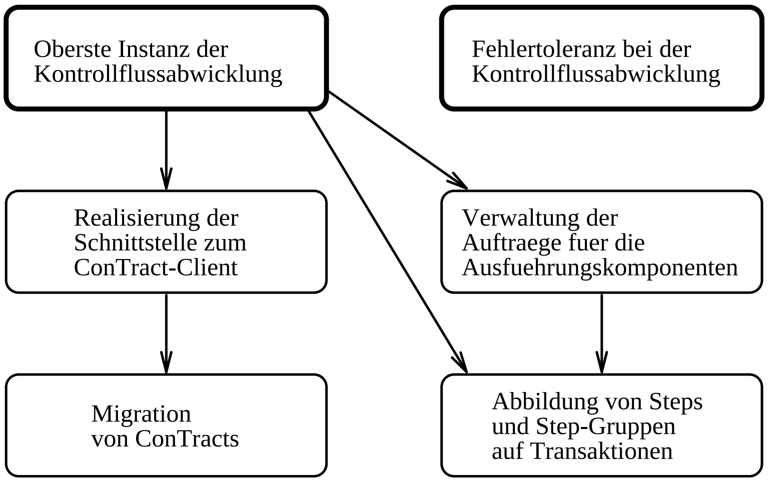


Abbildung 2.9: Die Aufgaben des ConTract–Managers

2.10.1 Die Beziehungen des ConTract–Managers zu den anderen Komponenten

In Abbildung 2.10 sind die Beziehungen des ConTract–Managers zu den anderen Komponenten des APRICOT–Systems dargestellt. Im Sinne der Betrachtungen zu den Aufgaben des ConTract–Managers ist sie als Weiterentwicklung von Abbildung 2.1 zu verstehen.

Der ConTract–Manager nimmt in diesem Beziehungsgeflecht unterschiedliche Rollen ein. Teilweise agiert er als Auftraggeber, teilweise als Auftragnehmer. Die einzelnen Beziehungen sehen dabei wie folgt aus:

1. *ConTract–Manager — ConTract–Client*

   Der ConTract–Manager nimmt in dieser Beziehung immer die Rolle des Auftragnehmers ein. Er erhält vom ConTract–Client Kommandos, die er berarbeitet und dann quittiert.

2. *ConTract–Manager — ConTract–Processing–Monitor*

   In dieser Beziehung nimmt der ConTract–Manager sowohl die Rolle des Auftraggebers, als auch die des Auftragnehmers ein. Er erzeugt für auszuführende Steps Ausführungsaufträge und schickt sie an den ConTract–Processing–Monitor. Der ConTract–Processing–Monitor verarbeitet diese Aufträge, indem er sie an die Ausführungskomponenten zuteilt.

   Gleichzeitig ist der ConTract–Manager jedoch Auftragnehmer des ConTract–Processing–Monitors. Der ConTract–Processing–Monitor schickt Ergebnisse von Stepausführungen an den ConTract–Manager zurück. Er erteilt dem ConTract–Manager damit den Auftrag, diese Ergebnisse zu verarbeiten.

3. *ConTract–Manager — Transaction–Manager*

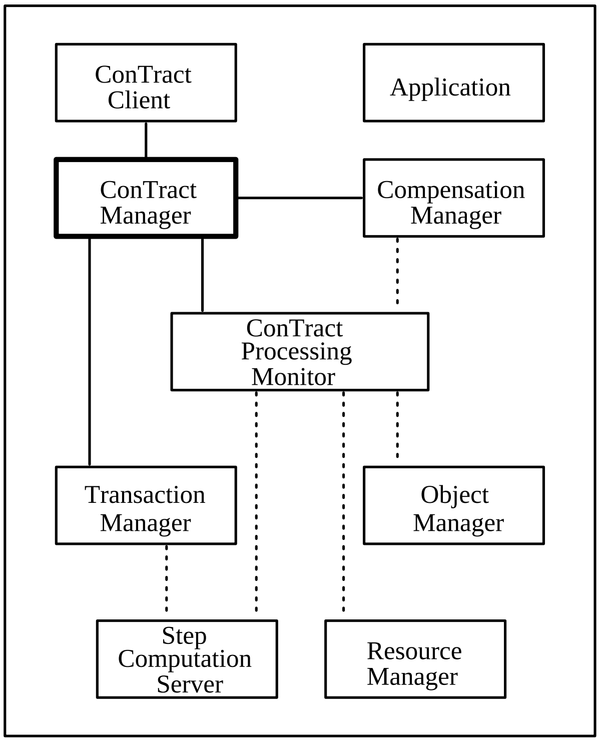


Abbildung 2.10: Einordnung des ConTract–Managers ins APRICOT–System

In der Beziehung zum Transaction–Manager ist der ConTract–Manager wiederum nur Auftraggeber. Er veranlaßt die Erzeugung von Transaktions–IDs und vereinbart Transaktionsbeziehungen.

4. *ConTract–Manager — Compensation–Manager*

   Hier ist der ConTract–Manager ebenfalls nur Auftraggeber. Er entscheidet, wann die Kompensation eines ConTracts eingeleitet werden soll. Daraufhin beauftragt er den Compensation–Manager die Kompensation durchzuführen.

Der ConTract–Manager verarbeitet Nachrichten von verschiedenen Komponenten. Dabei kann es vorkommen, daß mehrere Nachrichten gleichzeitig eintreffen. Um die Antwortzeiten auf Nachrichten anderer Komponenten möglichst gering zu halten, muß der ConTract–Manager viele Nachrichten innerhalb kurzer Zeit bearbeiten können. Ein hoher Durchsatz an Nachrichten ist deshalb eine wichtige Anforderung an Entwurf und Implementierung des ConTract–Managers.

Kapitel 3

Detaillierte Darstellung zentraler Aufgaben

In Kapitel 2 wurden die Aufgaben des ConTract–Managers im Überblick dargestellt. Hier werden nun diejenigen Aufgaben detailliert beschrieben, die den Schwerpunkt der Studienarbeit bilden.

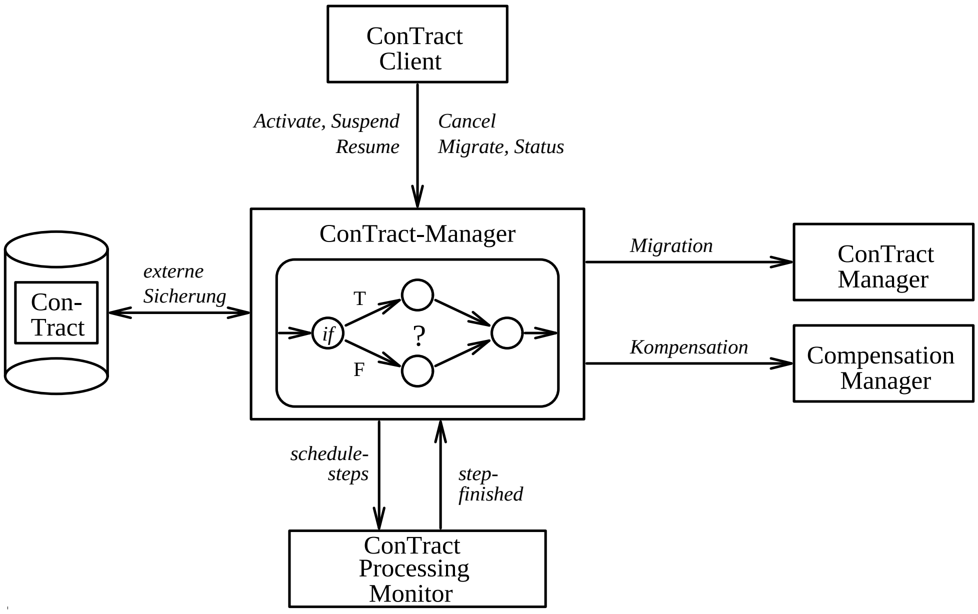


Abbildung 3.1: Zentrale Aufgaben des ConTract–Managers

Im einzelnen sind dies:

1. *Die Interpretation des ConTract–Skripts*

2. *Die Verwaltung der Aufträge für die Ausführungskomponenten*
3. *Die Schnittstelle zum ConTract–Client*

Abbildung 3.1 stellt diese Aufgabenbereiche schematisch dar.

3.1 Interpretation des ConTract–Skripts

Die Sprache zur Programmierung des Skripts, die bislang im APRICOT–System verwendet wird, ist in [Mas90] beschrieben. Es handelt sich dabei um eine rein textorientierte Notation, die sich an *Concurrent Pascal* anlehnt. Zur Beschreibung von parallelen Abläufen mit Prädikat–Transitionsnetzen würde sich ein graphische Notation weit besser eignen. Eine solche anwenderfreundliche Programmierschnittstelle könnte die bisherige Notation in späteren Versionen des Systems ersetzen. Durch die Definition einer *Zwischensprache*, im weiteren *Zwischen–Code* genannt, wurde die Programmierschnittstelle von den anderen Komponenten entkoppelt. Der Zwischen–Code wird vom Skript–Compiler in einer relationalen Datenbank abgelegt. Der ConTract–Manager ließt den Zwischen–Code beim Start eines ConTracts ein und beginnt dann mit seiner Interpretation.

Die folgenden Abschnitte erläutern die Mechanismen und die Basiskonstrukte des Zwischen–Codes.

3.1.1 Der Event–Trigger–Mechanismus

In Abschnitt 2.1.2 wurde die Struktur der Prädikat–Transitionsnetze in ihrer Analogie zu den Petri–Netzen einführend beschrieben. Im folgenden möchte ich nun die Prädikat–Transitionsnetze näher erläutern. Insbesondere die Grundregeln für ihre Interpretation sollen hier besprochen werden.

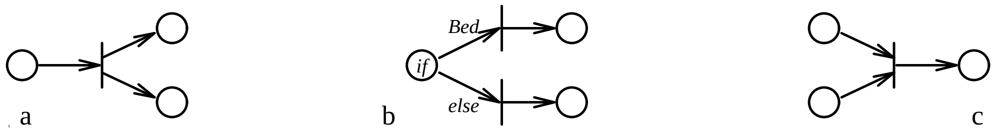


Abbildung 3.2: Beispielhafte Basiskonstrukte des Prädikat–Transitionsnetzes

Wie bei Petri enthält ein aktives Netz Markierungen, sogenannte Token, die die Aktivitätspunkte[1] kennzeichnen. Das Weiterreichen der Token wird im Prädikat–Transitionsnetz *triggern* genannt. Triggert ein Event, so schickt er an jeden Nachfolge–Step jeweils ein Token. Erhält ein Step ein Token, so erzeugt der ConTract–Manager einen Ausführungsauftrag für diesen Step und

[1] Bei paralleler Ausführung verschiedener Zweige hat ein Skript mehrere Aktivitätspunkte.

reicht ihn an den ConTract–Processing–Monitor weiter. Der Step wird ausgeführt und läuft als atomare Einheit der ConTract–Abwicklung ab. Eine Verzweigung auf parallel arbeitende Zweige ist leicht möglich (Abbildung 3.2a).

Ist die Ausführung beendet, so triggert der Step seine Nachfolge–Events. Die Triggerung der Steps erfolgt jedoch nach Regeln, die gegenüber dem Petri–Netz um einen wesentlichen Punkt erweitert sind. Wie in Abschnitt 2.1.2 bereits erläutert wurde, ist jedem Event eine Bedingung vorgeschaltet. Diese Bedingung wird auch Trigger–Bedingung des Events genannt. Ihre Auswertung entscheidet darüber, ob der Event ein Token erhält oder nicht. Hat nun ein Step seine Ausführung beendet, so prüft der ConTract–Manager alle Trigger–Bedingungen seiner Nachfolge–Events. Liefert eine Bedingung den Wahrheitswert *true*, so erhält der zugehörige Event ein Token. Mit diesem Mechanismus ist die bedingte Ausführung verschiedener Pfade des Skripts möglich (Abbildung 3.2b). Genutz wird diese Möglichkeit unter anderem auch zur Behandlung bestimmter Fehlerfälle[2] bei der ConTract–Ausführung. Auch die Kontrollflußkonstrukte des Prädikat–Transitionsnetzes, die in 3.1.3 besprochen werden, machen davon gebrauch.

In Petri–Netzen wird die Synchronisation paralleler Pfade dadurch erreicht, daß Transitionen erst dann zünden, wenn jede Eingabestelle markiert ist. Derselbe Effekt entsteht beim Prädikat–Transitionsnetz durch die Definition einer bestimmten Anzahl Token, die ein Event zur Triggerung benötigt. Steps können jederzeit Token an ihre Nachfolge–Events senden. Die Events speichern die empfangenen Token und triggern erst, wenn sie genug Token empfangen haben. Der Event in Abbildung 3.2c) benötigt zum Beispiel zwei Token um zu triggern. Er synchronisiert damit die beiden einlaufenden Pfade.

3.1.2 Steps und Events im Zwischen–Code

Der Skript–Compiler ([Mas90]) erweitert die Beschreibung des Prädikat–Transitionsnetzes im Zwischen–Code um spezielle Steps *(Pseudosteps)*, die im Skript nicht enthalten sind. Außerdem sind bestimmte Events in ihrer Funktion vordefiniert. Dieser Aspekt zählt im Gesamtbild des APRICOT–Systems zwar zum Entwurf bzw. zur Realisierung. Aus Sicht des ConTract–Managers ist diese Schnittstelle jedoch vorgegeben. Sie ist deshalb Teil seiner Spezifikation.

Pseudosteps

Diese Steps enthalten keinen ausführbaren Programm–Code. Sie werden vom Skript–Compiler eingeführt um die Struktur bestimmter Kontrollflußkonstrukte im Prädikat–Transitionsnetz zu realisieren. Der ConTract–Manager erzeugt für Pseudosteps keine Aufträge an die Ausführungskomponenten, sondern wertet sie direkt aus. Der Pseudostep *sendtoken* reicht zum Beispiel nur ein empfangenes Token an seinen Nachfolge–Event weiter.

In Abschnitt 3.1.3 werden die verschiedenen Pseudosteps bei der Besprechung der Kontrollflußkonstrukte vorgestellt.

[2]Hier insbesondere Konfliktbehandlung bei Invariantenfehlern (siehe Abschnitt 3.1.4, [WäRe90], [WäRe91] und [Mas90]).

Vordefinierte Events

Jedem Event im Zwischen–Code wird vom Skript–Compiler ein, innerhalb des Skripts eindeutiger, Event–ID zugewiesen. Dieser ID besteht aus einer ganzen Zahl. Für jeden ConTract werden mindestens folgende drei Events vordefiniert:

1. *start*

   Der Start–Event definiert den Beginn der ConTract–Ausführung. Der ConTract wird durch Triggerung dieses Events gestartet. Sein ID hat den Wert 0 (*null*).

2. *termination*

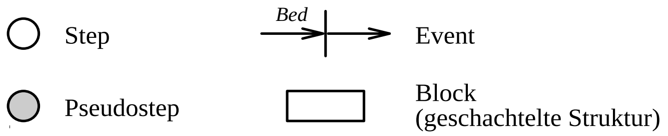
   An der Triggerung dieses Events erkennt der ConTract–Manager das Ende der ConTract–Ausführung. Der Wert des IDs ist 1 (*eins*).

3. *conflict resolution failed*

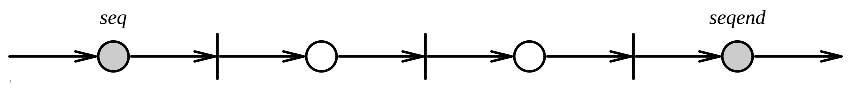
   Scheitert eine Konfliktbehandlung[3], so wird dieser Event getriggert. Der ConTract–Manager muß dann die Kompensation einleiten. Der Wert seines IDs ist 2 (*zwei*).

3.1.3 Kontrollflußkonstrukte

Der Zwischen–Code verfügt über eine Reihe von Sprachkonstrukten, mit denen der Kontrollfluß des ConTracts beschrieben wird. Die Abbildungen zeigen jeweils Ausschnitte aus Prädikat–Transitionsnetzen, die diese Sprachkonstrukte graphisch darstellen. Dabei werden folgende Darstellungsmittel verwendet:



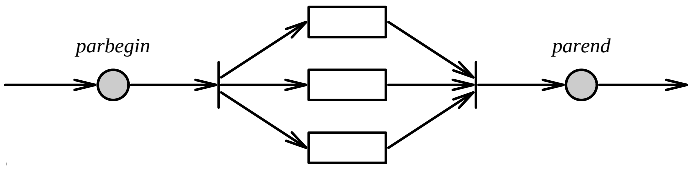
Seq



Die beiden Pseudosteps *seq* und *seqend* klammern eine Reihe von Steps, die sequentiell ablaufen sollen. Der Skript–Compiler verwendet diese Klammerung für Sequenzen, die innerhalb anderer Kontrollkonstrukte geschachtelt werden.

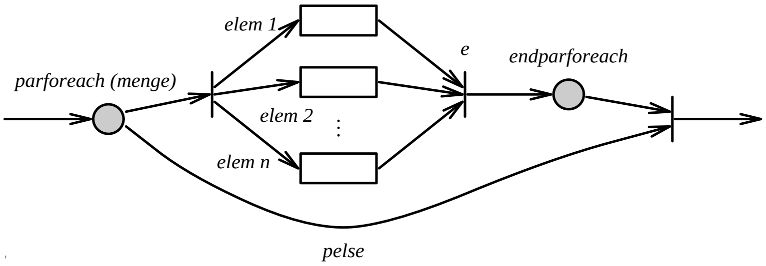
[3]Details der Konfliktbehandlung siehe Abschnitt 3.1.4 auf Seite 29.

Par



Mit den Pseudosteps *parbegin* und *parend* wird eine feste Anzahl parallel auszuführender Zweige definiert. Jeder Zweig kann wiederum ein Prädikat–Transitionsnetz enthalten.

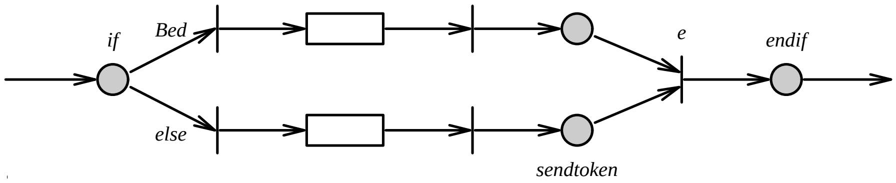
Parforeach



Parforeach erlaubt ebenfalls die Ausführung einer Reihe paralleler Zweige. Allerdings wird deren Anzahl erst zur Laufzeit bestimmt.

Der Pseudostep *parforeach* enthält die Beschreibung einer Menge von Werten[4] aus dem Kontext. Der ConTract–Manager berechnet diese Menge zur Laufzeit und startet für jedes Element einen parallelen Ausführungspfad. Ist die berechnete Menge leer, so wird der *pelse* Zweig getriggert. Der Synchronisations–Event *e* führt die parallelen Pfade wieder zusammen. Der ConTract–Manager muß die Anzahl der Token, die dieser Event zur Triggerung benötigt, ebenfalls zur Laufzeit setzen[5].

If

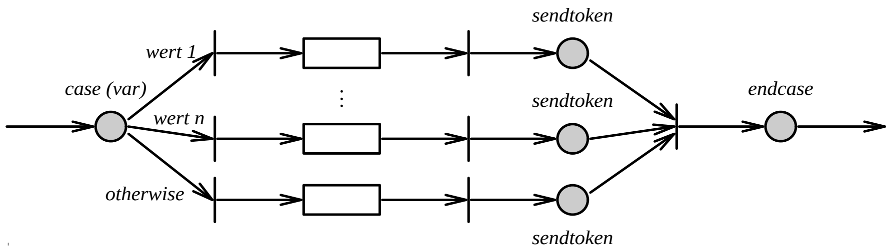


[4]In der aktuellen Version des Prototyps wird dafür eine SQL–Abfrage verwendet.

[5]Dies ist der einzige Fall bei der Interpretation des Prädikat–Transitionsnetzes, in dem die Anzahl der Token zur Triggerung eines Events gesetzt werden muß. Für alle anderen Events im Netz kann dieser Wert bereits vom Skript–Compiler vordefiniert werden. Zur Triggerung von Events, siehe Abschnitt 3.1.1 auf Seite 25.

Die Auswertung der Bedingung[6] im Pseudostep *if*, erlaubt die bedingte Ausführung zweier Zweige im Prädikat–Transitionsnetz. Es wird genau einer der beiden Zweige ausgewählt. Der Abschluß–Event e benötigt deshalb zur Triggerung auch nur ein Token.

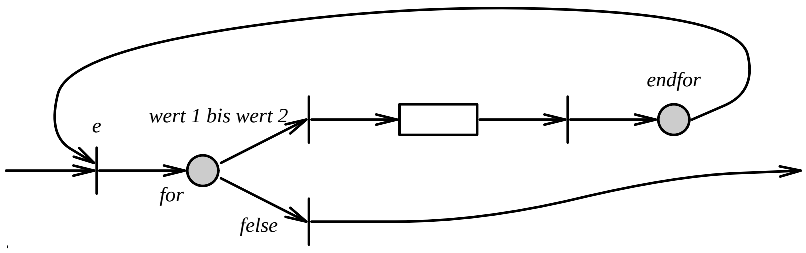
Case



Das *case* Konstrukt erlaubt die bedingte Ausführung eines von mehr als zwei Zweigen. Dazu wird ein Ausgabedatum eines vorangegangenen Steps *(var)* ausgewertet und genau einer der Zweige ausgewählt. Die Werte 1 bis n werden vom Skript–Compiler eingetragen.

Enthält *var* keinen der Werte 1 bis n, so wird der *otherwise* Pfad getriggert. Dieser Pfad kann auch leer sein. In diesem Fall wird der Abschluß–Event vor *endcase* direkt getriggert. Dieser Event benötigt nur ein Token zur Triggerung.

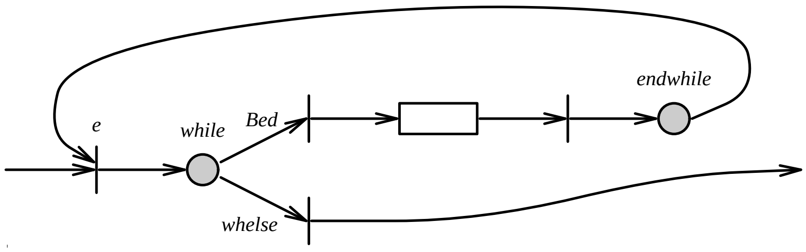
For



Die *for* Schleife wird für jeden der Werte 1 bis n einmal durchlaufen. Nach dem letzten Durchlauf wählt der ConTract–Manager den *felse* Zweig um mit der Interpretation des Prädikat–Transitionsnetzes fortzufahren.

[6]In der aktuellen Version des Prototyps werden Bedingungen durch Ausdrücke beschrieben, die Ausgabedaten aus dem Kontext der vorangegangenen Steps verwenden.

While



Die *while* Schleife hat fast dieselbe interne Struktur, wie die *for* Schleife. An die Stelle des Wertebereichs tritt hier eine Ausführungsbedingung. Die Schleife wird solange durchlaufen, bis die Bedingung den Wahrheitswert *false* liefert.

3.1.4 Konfliktbehandlung

An dieser Stelle möchte ich das Beispiel der Flugbuchung aus Abschnitt 2.2.2 wieder aufgreifen. Die Abteilung eines Unternehmens verfügt hier über ein bestimmtes Reisebudget für seine Mitarbeiter. Dieses Budget darf durch die Flugbuchung nicht überschritten werden.

Die Flugbuchung wird von einem ConTract durchgeführt, dessen Skript folgende Steps enthält:

Step 1	Reisebudget abfragen
Step 2	Flug auswählen und Preis ermitteln
Step 3	Flugbuchung vornehmen
Step 4	Buchungsdaten ausgeben

Die Steps laufen sequentiell ab.

Für unser Beispiel wollen wir folgenden Fall annehmen:

Step 1	liefert als Ergebnis 1000,– DM.
Step 2	wählt daraufhin einen Flug aus, der 900,– DM kostet.
	Zwischen der Ausführung von Step 2 und Step 3 bucht ein. parallel arbeitendes Programm 400,– DM vom Reisebudget ab.
Step 3	scheitert, da das Reisebudget erschöpft ist.

Im Transaktionsmodell wäre dieser Fall nicht eingetreten. Bei der Abfrage des Reisebudgets hätte die Einrichtung einer Schreibsperre auf dieses Datenobjekt die Dauerhaftigkeit seines Werts garantiert. Das Invariantenkonzept des ConTract–Modells sieht in diesem Fall die Einrichtung einer *Eingangsinvarianten* für Step 3 vor. Die Eingangsinvariante beschreibt die notwendigen Voraussetzungen, die ein Datenobjekt vor Ausführung des Steps erfüllen muß.

In der aktuellen Version des APRICOT–Systems werden Invarianten als SQL–Abfragen formuliert. Die Eingangsinvariante für Step 3 würde demnach so aussehen[7]:

```
SELECT count (*)
FROM   abteilung
WHERE        abtnr = :abtnr
       AND budget >= :flugpreis
```

Die Auswertung dieser SQL–Abfrage ergibt den Wert 0 für *false* oder 1 für *true*.

Scheitert die Eingangsinvariante, so wird die Konfliktbehandlung gestartet. Im vorliegenden Fall könnte diese Konfliktbehandlung darin bestehen, das Reisebudget der Abteilung zu erhöhen. Gelingt dies[8], so kann der Step, der die Konfliktbehandlung ausgelöst hat, erneut gestartet werden.

Bei der Wiederholung des Steps kann die Eingangsinvariante natürlich erneut scheitern. Die Konfliktbehandlung würde wiederholt und Step nochmal gestartet werden. Der ConTract–Manager läßt nur eine bestimmte Anzahl an Wiederholungen zu, um hier endlose oder zumindest eine große Zahl von Wiederholungen zu vermeiden.

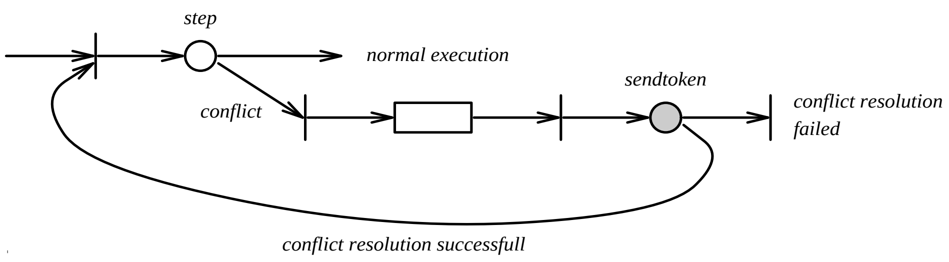


Abbildung 3.3: Konfliktauflösung im Prädikat–Transitionsnetz

Im Prädikat–Transitionsnetz ist die Konfliktbehandlung durch eine Verzweigung in einen speziellen Ausführungspfad realisiert. Meldet der ConTract–Processing–Monitor mit der Terminierungsmeldung eines Steps das Scheitern der Eingangsinvarianten, so verzweigt der ConTract–Manager in den *conflict* Pfad (Abbildung 3.3). Bei der Behandlung des Pseudosteps *sendtoken* prüft der ConTract–Manager, ob der Konflikt aufgelöst werden konnte. Ist dies der Fall, so kann der konfliktauslösende Step wiederholt werden. Scheitert die Konfliktbehandlung jedoch, so triggert der ConTract–Manager den vordefinierten Event *conflict resolution failed* (siehe Abschnitt 3.1.2) — die Kompensation des ConTracts wird ausgelöst.

[7] Die Variable *flugpreis* wird von Step 1 erzeugt und im Kontext ([BuRai90]) gespeichert.

[8] …der Chef ist damit einverstanden …

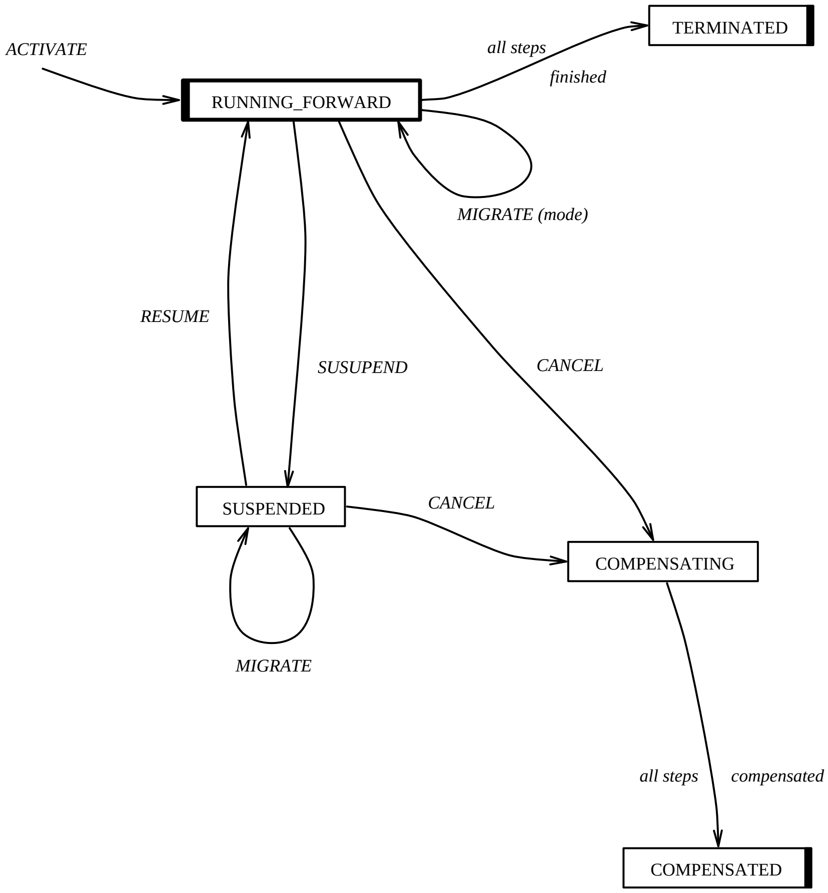


Abbildung 3.4: Die Modellwelt des ConTract–Clients — Zustände eines ConTracts

3.2 Die ConTract–Client–Schnittstelle zur Steuerung der ConTract–Abwicklung

In Abschnitt 2.4 wurden die einzelnen Kommandos der Schnittstelle kurz beschrieben. Hier möchte ich nun zeigen, welche Modellvorstellung der Benutzer via ConTract–Client von der Abwicklung eines ConTracts hat. Abbildung 3.4 zeigt die verschiedenen Zustände und Zustandsübergänge eines ConTracts aus Sicht des ConTract–Client.

Zur Identifikation eines ConTracts im APRICOT–System, vergibt der ConTract–Manager einen global eindeutigen *cid*[9]. Der ConTract–Client erhält diesen ID als Rückgabewert beim Start eines ConTracts mit dem Kommando ACTIVATE. Er benötigt den cid um weitere Kommandos bezüglich dieses ConTracts absetzen zu können. In der aktuellen Version des Prototypsystems APRICOTS ist der ConTract–Manager jeweils nur für einen ConTract zuständig.

[9] ConTract–ID.

Die Prozeßverwaltung des Systems sorgt dafür, daß für jeden aktiven ConTract ein ConTract–Manager existiert. Aufgrund dieser eins zu eins Zuordnung ist es nicht notwendig den cid bei jedem Kommando des ConTract–Clients an den ConTract–Manager weiterzuleiten. Das Kommunikationssystem nimmt das Kommando vom ConTract–Client entgegen. Der ConTract–Client muß dabei jeweils den cid angeben. Das Kommunikationssystem übernimmt nun die Zuordnung von cid zu ConTract–Manager. Der cid wird an der ConTract–Manager–Schnittstelle herausgefiltert. Alle anderen Parameter werden an den ConTract–Manager weitergeleitet.

3.2.1 Start und Abbruch

ACTIVATE

Parameter:

- Symbolischer Name des ConTracts

Der ConTract–Manager initialisiert seine Datenstrukturen zur Verwaltung des ConTracts und erzeugt den global eindeutigen *cid*. Dann lädt er den Zwischen–Code und startet die Ausführung des ConTracts. Der ConTract befindet sich nun im Zustand RUNNING_FORWARD. Schließlich quittiert der ConTract–Manager den erfolgreichen Start, indem er dem ConTract–Client den cid mitteilt.

CANCEL

Mit diesem Kommando wird der ConTract abgebrochen und seine Kompensation eingeleitet. Der Abbruch kann während des normalen Ablaufs (RUNNING_FORWARD), oder auch während einer Unterbrechung (SUSPENDED) erfolgen. Der ConTract–Manager quittiert den Abbruch, sobald die Kompensation läuft.

Das Kommando hat keine Parameter.

3.2.2 Unterbrechungen

SUSPEND

Parameter:

- mode (SOFT oder HARD)
- duration (TEMPORARY oder PERMANENT)

Der ConTract wird unterbrochen, das heißt, seine Ausführung wird angehalten. Bei der Unterbrechung erzeugt der ConTract–Manager die *Suspend Liste*[10]. In dieser Liste werden alle momentan aktiven Aufträge gespeichert. Diese Daten benötigt das RESUME–Kommando zum Wiederanfahren des ConTracts.

Die Unterbrechung kann beliebig lange andauern. Der ConTract–Client kann die Ausführung des ConTracts jederzeit mit dem RESUME–Kommando fortsetzen. Sobald der gewünschte Zustand erreicht ist, quittiert der ConTract–Manager die Unterbrechung.

Die Art der Unterbrechung wird durch die Parameter *mode* und *duration* näher bestimmt. *Mode* kann mit den Werten SOFT oder HARD belegt werden. SOFT bedeutet: Unterbrechung, so *preiswert* wie möglich. Der Preis, der dafür zu zahlen ist, daß bereits berechnete Zwischenergebnisse verworfen werden, kann unter Umständen sehr hoch sein. Bei der weichen Unterbrechung werden deshalb alle noch laufenden Berechnungen zu Ende geführt, bevor die Verarbeitung angehalten wird. Somit ist die optimale Ausnutzung der vorhandenen Betriebsmittel gewährleistet.

Bei HARD erfolgt die Unterbrechung hingegen so *schnell* wie möglich, auch wenn dabei bereits berechnete Zwischenergebnisse zunächst verlorengehen. *Verloren* heißt hier nicht, daß sich der ConTract nun in einem undefinierten Zustand befindet — gerade das soll im ConTract–Modell verhindert werden (siehe Zusicherung 1 auf Seite 8). Das APRICOT–System versucht lediglich die Verarbeitung so schnell wie möglich in einem definierten Zustand einzufrieren, wobei noch laufende Berechnungen verworfen werden.

Der Parameter *duration* beschreibt die Zeitdauer einer Unterbrechung. Der Wert TEMPORARY (vorläufig) veranlaßt den ConTract–Manager, die ConTract–Daten nach der Unterbrechung im Hauptspeicher des Rechners zu belassen. Diese Methode der Datenhaltung kann vorteilhaft für kurze Unterbrechungen oder auf besonders ausfallsicheren Rechnern angewendet werden. Sie ist schnell bei der Unterbrechung und beim Wiederanfahren der Verarbeitung. Andererseits sind aufwendige Recovery–Maßnahmen erforderlich, falls der Rechner ausfällt.

PERMANENT (dauerhaft) speichert die ConTract–Daten nach der Unterbrechung auf einem externen (dauerhaften) Speicher. Bei längeren Unterbrechungen oder bei großem Ausfallrisiko des verfügbaren Rechners sollte diese Betriebsart verwendet werden. Die Unterbrechung und die Fortsetzung der Verarbeitung dauern hier etwas länger als in der TEMPORARY–Betriebsart, da die Daten ein- und ausgelagert werden müssen.

RESUME

Die Ausführung eines ConTracts, der zuvor mit SUSPEND unterbrochen wurde, wird wieder gestartet. Der ConTract–Manager startet den ConTract unabhängig von der Art der Unterbrechung. Er lädt alle ConTract–Daten und beginnt mit der Ausführung an der *Aktivitätslinie*[11] des ConTract–Skripts, die durch die Suspend Liste[12] beschrieben wird.

RESUME hat keine Parameter.

[10]Siehe Abschnitt 2.8.

[11]Alle vor der Unterbrechung aktiven Punkte des Prädikat–Transitionsnetzes.

[12]Siehe Abschnitt 2.8.

3.2.3 Wanderung im Rechnernetz

MIGRATE

Parameter:

- mode (SOFT oder HARD)
- Empfänger der Migration (ConTract–Manager)

Der ConTract kann sowohl während der normalen Ausführung als auch während einer Unterbrechung migriert werden. Allerdings unterbricht der ConTract–Manager die Ausführung eines ConTracts, falls er nicht schon unterbrochen ist, um ihn zu übertragen und auf dem Zielrechner erneut zu starten. Dieses Verfahren, den ConTract zuerst zu unterbrechen und dann zu migrieren, habe ich gewählt, um einen einheitlichen Mechanismus für unterbrochene und laufende ConTracts verwenden zu können.

Der ConTract–Client wählt mit dem Parameter *mode* die Art der Unterbrechung, falls sich der ConTract im Zustand RUNNING_FORWARD befindet. Dieser Parameter wird dabei analog zum SUSPEND–Kommando verwendet.

Wurde die Migration erfolgreich abgewickelt, so teilt der Empfänger deren Erfolg dem aufrufenden ConTract–Client mit. Die Prozeßverwaltungseinheit des APRICOT–Systems muß daraufhin dem Empfänger der Migration einen neuen ConTract–Client zuteilen[13], damit der Benutzer nun seinen Arbeitsplatz wechseln kann.

3.2.4 Zustandsabfragen

STATUS

Parameter:

- infolevel

Die Abfrage des ConTract–Zustands hat keine Auswirkungen auf den Ablauf des ConTracts. Art und Umfang der abfragbaren Daten hängen stark von der Implementierung des ConTract–Managers ab. Die Zustandsabfrage wurde deshalb noch nicht im Detail spezifiert. Für den Benutzer wäre zum Beispiel interessant, welche Steps gerade ablaufen und auf welchen Rechnern des Netzes sie ausgeführt werden.

In seiner vorläufigen Version stellt das STATUS–Kommando den Parameter *infolevel* zur Verfügung. Mit ihm können Art und Umfang der gewünschten Information gewählt werden.

[13] Siehe dazu Abschnitt 2.9 auf Seite 19.

3.3 Verwaltung der Aufträge für die Ausführungskomponenten

In Abschnitt 2.5 haben wir die Auftragsverwaltung bereits im Überblick betrachtet. Hier möchte ich nun darstellen, wie Aufträge erzeugt werden und welche Schnittstellen zur Bearbeitung der Aufträge benötigt werden.

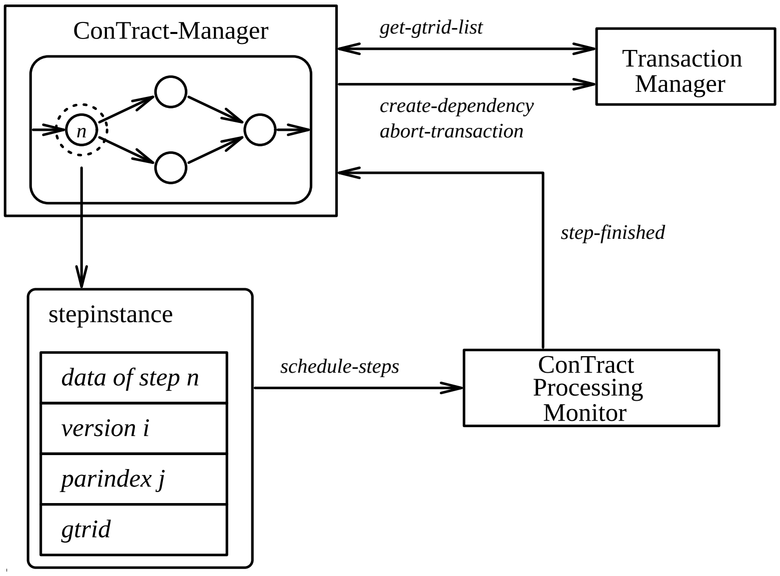


Abbildung 3.5: Verwaltung der Aufträge durch den ConTract–Manager

Jede Ausführungskomponente verfügt über den ausführbaren Programm–Code eines oder mehrerer Steps. Jedes dieser Programme repräsentiert einen Step und wird durch einen *logischen Namen* gekennzeichnet. Der logische Name wird bei der Step–Programmierung vergeben und bei der Programmierung des Skripts für den Step–Aufruf verwendet. Der ConTract–Manager muß nun nicht mit jedem Ausführungsauftrag den gesamten ausführbaren Programm–Code mitschicken. Es genügt, dem ConTract–Processing–Monitor den logischen Namen des Step zu übergeben. Der ConTract–Processing–Monitor teilt dann den Auftrag einer Ausführungskomponente zu, die über den gewünschten Programm–Code verfügt.

3.3.1 Step–Instanzen

Die Step–Instanz repräsentiert den Ausführungsauftrag für die Ausführungskomponenten. Sie enthält Step–Daten, wie Eingangs- und Ausgangsinvariante sowie den logischen Namen des Steps.

Ein Step kann nacheinander mehrmals ausgeführt werden, das heißt, er erzeugt mehrere Instanzen. Befindet sich ein Step auf Skriptebene zum Beispiel innerhalb einer Schleife, so wird bei jedem Schleifendurchlauf eine Step–Instanz erzeugt. Bei *parforeach* läuft dieselbe Version eines

Steps sogar in mehreren Instanzen parallel. Zur Unterscheidung der Step–Instanzen verwaltet der ConTract–Manager deshalb eine Versionsnummer und eine Kennung für parallele Instanzen.

3.3.2 Die Schnittstelle zum ConTract–Processing–Monitor

Der ConTract–Processing–Monitor übernimmt im APRICOT–System die Rolle des *Schedulers*. Er nimmt die Aufträge des ConTract–Managers entgegen. Nach der Ausführung der Steps schickt er die zugehörigen Ausführungsergebnisse zurück. Der ConTract–Processing–Monitor definiert für den ConTract–Manager also eine Schnittstelle zu den Ausführungskomponenten. Diese Schnittstelle enthält im wesentlichen zwei Funktionen:

1. *schedule_steps*

   Mit dieser Funktion teilt der ConTract–Manager dem ConTract–Processing–Monitor mit, daß er Step–Instanzen starten möchte. Die Step–Instanzen werden in einer Liste übergeben, sodaß nicht für jede Instanz ein Aufruf erfolgen muß. Der Kommunikationsaufwand wird dadurch so gering wie möglich gehalten.

2. *step_finished*

   Der ConTract–Processing–Monitor erhält von den Ausführungskomponenten Nachricht, wenn die Ausführung einer Step–Instanz beendet ist. Er leitet diese Meldung an den ConTract–Manager weiter. Jede *step_finished* Meldung enthält genau diejenige Step–Instanz, deren Ende mitgeteilt wird. Außerdem wird ein *returncode* geliefert, der über den Erfolg oder Mißerfolg der Ausführung Auskunft gibt.

3.3.3 Die Schnittstelle zum Transaction–Manager

Wie in 2.6 besprochen wurde, ist der ConTract–Manager auch für die Abbildung der Steps auf Transaktionen zuständig. Außerdem muß er bei einem HARD–SUSPEND oder bei einem HARD–MIGRATE dafür sorgen, daß alle aktiven Steps so schnell wie möglich abgebrochen werden. Er verfügt dazu über drei Funktionen in seiner Schnittstelle zum Transaction–Manager:

1. *get_gtrid_list*

   Jede Step–Instanz erhält eine eigene Transaktions–ID (*gtrid*)[14]. Mit der Funktion get_gtrid_list kann der ConTract–Manager beim Transaction–Manager eine Reihe von Transaktions–IDs anfordern.

   Die Transaktions–ID ist global eindeutig. Sie wird von den Ausführungskomponenten zur Anmeldung ihrer Transaktion beim Transaction–Manager benötigt.

2. *create_dependency*

[14] Global TRansaktion ID.

Mit dieser Funktion vereinbart der ConTract–Manager Beziehungen zwischen Transaktionen. Er teilt dem Transaction–Manager dazu die entsprechenden Transaktions–IDs und die Art der Beziehung[15] mit.

3. *abort_steps*

   Im Falle einer harten Unterbrechung der ConTract–Ausführung ruft der ConTract–Manager diese Funktion auf. Sie veranlaßt den sofortigen Abbruch einer Reihe von aktiven Steps, die noch aktiv sind. Als Parameter wird eine Liste von Transaktions–IDs übergeben. Der Transaction–Manager bricht die entsprechenden Transaktionen ab. Dadurch wird die Ausführung der Steps beendet und über den ConTract–Processing–Monitor an den ConTract–Manager gemeldet.

[15]Transaktionsbeziehungen werden in [Schü91] und [Lutz91] besprochen.

Kapitel 4

Entwurf eines prototypischen ConTract–Managers

Ursprüngliches Ziel des Entwurfs war es, einem ConTract–Manager mehrere ConTracts zur Bearbeitung zuzuweisen. Gegen Ende der Studienarbeit fiel diese Anforderung jedoch, aufgrund von Änderungen im Betriebsmodell, weg. Obwohl der ConTract–Manager (derzeit) nur einen einzigen ConTract verarbeiten muß, möchte ich die bisher entworfene Struktur beibehalten. Sie kann ohne Einschränkungen auch zur Verarbeitung eines einzelnen ConTracts verwendet werden. In der weiteren Entwicklung des APRICOT–Systems kann der ConTract–Manager auch leicht wieder zur Verarbeitung mehrerer ConTracts erweitert werden.

Bei der Darstellung des Entwurfs habe ich versucht die wesentlichen Grundgedanken und Entwurfsentscheidungen zu verdeutlichen. Dabei habe ich auf die Wiedergabe der Realisierung in Quelltext oder Pseudo–Code und eine detaillierte Darstellung der implementierten Datenstrukturen bewußt verzichtet. Die entsprechenden Unterlagen sind als Quelltext und Programmdokumentation des APRICOT–Systems verfügbar.

4.1 Entwurf der obersten Abstraktionsebene

Der Entwurf beschreibt die wesentlichen Mechanismen und Entwurfsentscheidungen. Details des Entwurfs werden nur insofern angesprochen, als sie für das Verständnis der grundlegenden Funktionen von Bedeutung sind.

Die Kompensation eines ConTracts wurde in diesen Entwurf bereits ansatzweise integriert. Notwendige Aktionen zur Einleitung der Kompensation werden hier beschrieben. Dabei wurde nicht mehr zwischen ConTract–Manager und Compensation–Manager unterschieden. Beide Aufgabenbereiche werden hier also vom ConTract–Manager abgedeckt. Da die Kompensation selbst nicht mehr Teil dieser Arbeit ist, beschränkt sich die Darstellung jedoch auf den Übergang von der normalen Ausführung zur Kompensation.

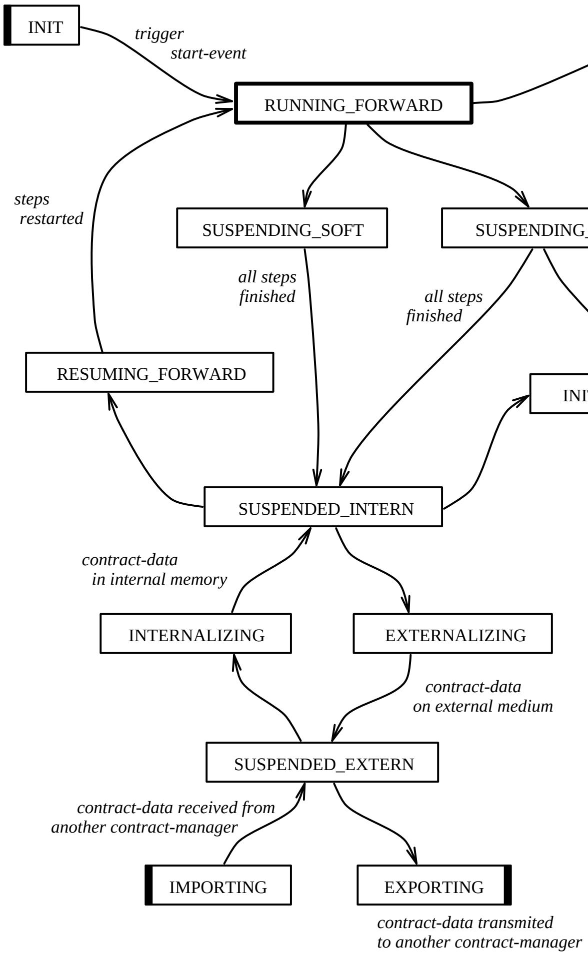


Abbildung 4.1: Die Modellwelt des ConTract–Managers — Zustände eines ConTracts

4.1.1 Die Modellwelt des ConTract–Managers

Die Zustände eines ConTracts aus Sicht des ConTract–Managers sind in Abbildung 4.1 in Form eines Zustandsdiagramms dargestellt. Alle Zustände aus der Modellwelt des ConTract–Clients (Abbildung 3.4) sind hier ebenfalls enthalten. Zusätzlich kennt der ConTract–Manager einige Zustände der internen ConTract–Verarbeitung, die der ConTract–Client nicht sieht.

Die Beschriftung der Pfeile in Abbildung 4.1 weißt auf interne Ereignisse hin, die einen Zustandsübergang auslösen. Im folgenden sollen die Zustände näher erklärt werden. Zu jedem Zustand ist eine kurze Beschreibung der Aktionen aufgeführt, die innerhalb dieses Zustands abgewickelt werden.

INIT *Startzustand* aus Sicht des ConTract–Managers — alle Da-

	tenstrukturen zum Start eines neuen ConTracts werden initialisiert.
RUNNING_FORWARD	Normaler Ablauf der ConTract–Abwicklung.
TERMINATED	*Endzustand* der ConTract–Abwicklung — der ConTract wurde vollständig und korrket abgewickelt.
SUSPENDING_SOFT	Die Stepverarbeitung wird im SOFT–Modus angehalten. Der ConTract–Manager wartet in diesem Zustand auf die Terminierung aller noch aktiven Step–Instanzen und erzeugt die Suspend Liste.
SUSPENDING_HARD	Die Stepverarbeitung wird im HARD–Modus angehalten. Der ConTract–Manager bricht alle aktiven Step–Instanzen ab und erzeugt die Suspend Liste.
SUSPENDED_INTERN	Die Verarbeitung ist unterbrochen. Alle ConTract–Daten befinden sich im Hauptspeicher.
SUSPENDED_EXTERN	Die Verarbeitung ist unterbrochen. Die ConTract–Daten wurden auf ein externes Speichermedium (Festplatte) ausgelagert.
RESUMING_FORWARD	Die Stepverarbeitung wird, anhand der Eintragungen in der Suspend Liste, wieder gestartet.
EXTERNALIZING	Die ConTract–Daten werden vom Hauptspeicher auf ein externes Speichermedium (Festplatte) ausgelagert.
INTERNALIZING	Die ConTract–Daten werden vom externen Speichermedium (Festplatte) in den Hauptspeicher eingelesen.
EXPORTING	*Endzustand* bei der Migration aus Sicht des Absenders der Übertragung. Alle ConTract–Daten werden an einen anderen ConTract–Manager verschickt. Dies geschieht dadurch, daß der Sender dem Empfänger mitteilt, wo sich die ausgelagerten ConTract–Daten befinden[1]. Der Empfänger kann dann auf diese Daten direkt zugreifen. Diese Art der Datenübergabe habe ich der direkten Versendung über das Kommunikationssystem vorgezogen, da die Prototypumgebung für APRICOTS den Zugriff auf die zentrale Datenbank von jedem Rechnerknoten aus zuläßt. Dadurch reduziert sich der Übertragungsaufwand und die Dauer der Übertragung erheblich.
IMPORTING	*Startzustand* bei der Migration aus Sicht des Empfängers der Übertragung. Alle ConTract–Daten werden auf die lokale Festplatte kopiert, falls ein direkter Zugriff auf den Datenbereich des Absenders nicht möglich ist.
INIT_COMPENSATION	Mit Hilfe der Eintragungen in der Suspend Liste werden die ersten Kompensationsaktionen des ConTracts gestartet.

[1] Datei oder Datenbank.

COMPENSATING	Normale Durchführung der Kompensation.
COMPENSATION_ERROR	Bei der Kompensation trat ein Fehler auf, der nicht automatisch zu beheben war. Der ConTract–Client wird von dem aufgetretenen Fehler unterrichtet. Wurde der Fehler durch den ConTract–Client oder den Systemadministrator behoben, so kann die Kompensation fortgesetzt werden.
COMPENSATED	*Endzustand* der ConTract–Abwicklung — Die Kompensation ist vollständig abgeschlossen.

4.1.2 Optimierung des Durchsatzes

Der ConTract–Manager empfängt Nachrichten vom ConTract–Client, vom ConTract–Processing–Monitor und von anderen ConTract–Managern[2]. Er verarbeitet diese Nachrichten und verschickt seinerseits Nachrichten an die aufgeführten Komponenten. Jede empfangene Nachricht wird bearbeitet bevor der ConTract–Manager die nächste entgegegennimmt. Das heißt die Verarbeitung von Nachrichten ist von außen nicht unterbrechbar. Die Grundstruktur des ConTract–Managers sieht also wie folgt aus[3]:

```
main ()
{
   ...
   do {
      lookup_message ();
      work_on_message ();
   } while (!END);
   ...
}
```

Der Durchsatz an Nachrichten hängt davon ab, wie lange ihre Verarbeitung dauert. Insbesondere kann die Verarbeitung beliebig lange blockiert werden, wenn der ConTract–Manager eine andere Komponente *synchron* aufruft[4]. Um einen optimalen Durchsatz zu erreichen, verzichtet der ConTract–Manager auf synchrone Aufrufe anderer Komponenten[5]. Alle Verbindungen mit anderen Komponenten werden *asynchron*, durch Austausch von Nachrichten realisiert. Innerhalb der Verarbeitung einer Nachricht werden also nur Nachrichten abgesendet und nicht empfangen. Der Empfang von Nachrichten findet im ConTract–Manager nur an einem Punkt, der Funktion *lookup_message*, statt. Diese Funktion wird vom Kommunikationssystem ([Burk91]) zur Verfügung gestellt.

[2]Bei der Migration.

[3]In C–Syntax.

[4]Der synchrone Aufruf entspricht dem Aufruf einer Prozedur oder einer Funktion in einer prozeduralen Programmiersprache. Die Verarbeitung des aufrufenden Programms wird solange blockiert, bis die Prozedur zurückkehrt.

[5]Mit Ausnahme des Aufrufs get_gtrid_list an den Transaction–Manager, der eine sehr kurze Antwortzeit hat.

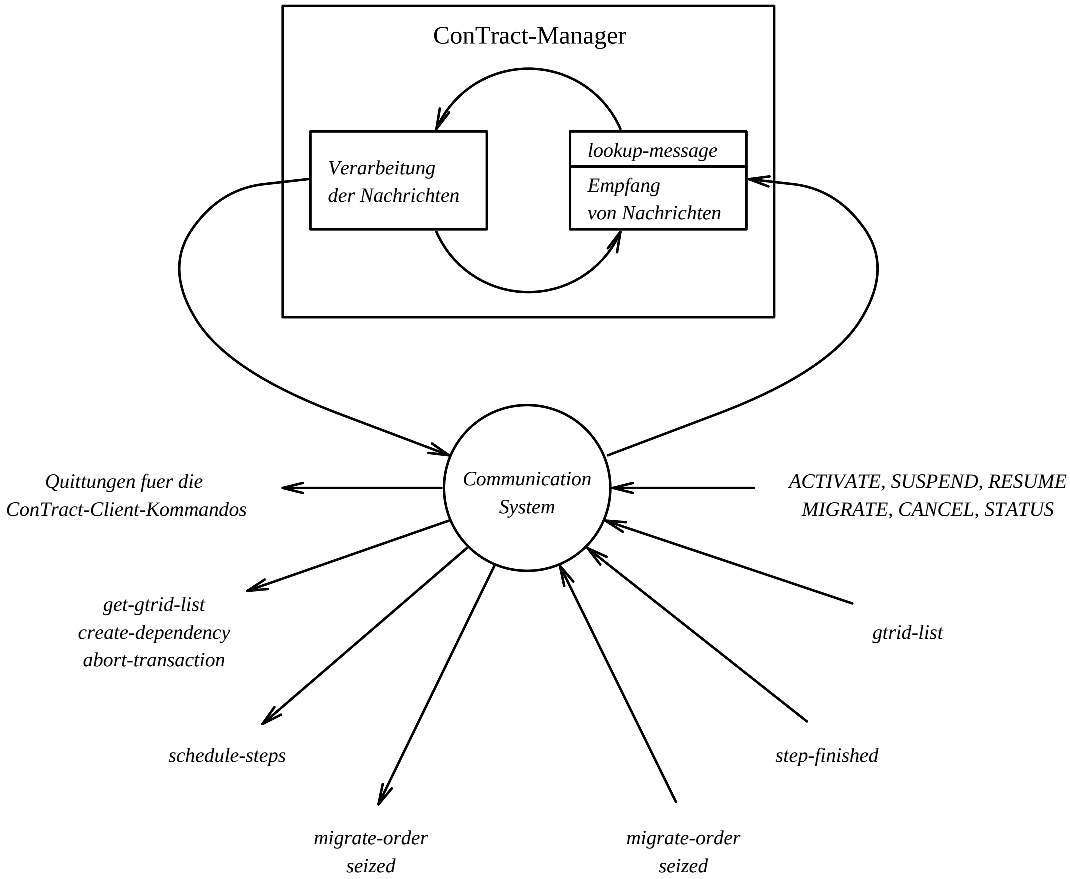


Abbildung 4.2: Die Verarbeitung von Nachrichten im ConTract–Manager

4.1.3 Funktionen zur Bearbeitung eingehender Nachrichten

Jeder Nachricht, die der ConTract–Manager empfangen und bearbeiten kann, ist eine Funktion zugeordnet. Diese Funktionen werden im folgenden *Nachrichtenfunktionen* genannt. Jede dieser Funktionen ruft Funktionen der tieferliegenden Schichten auf, die dann die Verarbeitung durchführen. Im folgenden werden sie als *Verarbeitungsfunktionen* bezeichnet. Die Art der Weiterverarbeitung hängt dabei jeweils vom aktuellen Verarbeitungszustand des ConTracts ab. Die Nachricht RESUME kann zum Beispiel in zwei Zuständen — in SUSPENDED_INTERN und in SUSPENDED_EXTERN — sinnvoll weiterverarbeitet werden. In allen anderen Zuständen ist das RESUME–Kommando nicht erlaubt.

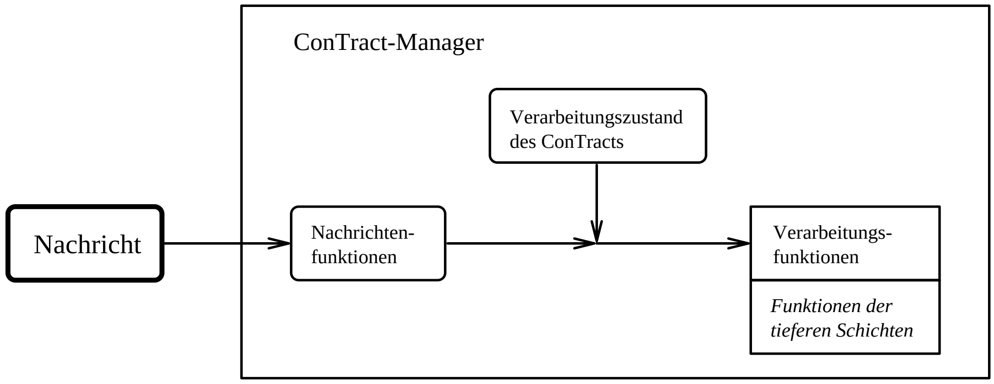


Abbildung 4.3: Auswahl der Verarbeitungsfunktion für eine Nachricht abhängig vom aktuellen Verarbeitungszustand eines ConTracts

Die Nachrichtenfunktionen

Insgesamt existieren auf dieser obersten Ebene des ConTract–Managers neun Funktionen zur Verarbeitung von Nachrichten.

(a) Sechs Funktionen zur Realisierung der ConTract–Client–Kommandos:

CM_activate	*CM_suspend*	*CM_resume*
CM_cancel	*CM_migrate*	*CM_status*

Die Kommandos des ConTract–Clients wurden bereits in Abschnitt 3.2 ausführlich besprochen.

(b) Für die Terminierungsmeldung einer Step–Instanz durch den ConTract–Processing–Monitor steht eine Funktion zur Verfügung:

CM_step_finished

Eine Darstellung der *step_finished* Nachricht findet sich in Abschnitt 3.3.2.

(c) Zur Behandlung der Migrationsnachrichten werden zwei Funktionen benötigt:

CM_migrate_order *CM_seized*

Die Nachrichten zur Durchführung der Migration werden in Abschnitt 4.1.3 besprochen.

Folgen von Zustandsübergängen

Jede Nachricht, die vom ConTract–Client an den ConTract–Manager gesendet wird löst eine Folge von Zustandsänderungen aus. Das Kommando SUSPEND (HARD,TEMPORARY) führt zum Beispiel über den Zwischenzustand SUSPEND_HARD in den Zustand SUSPENDED_INTERN. Das Kommando MIGRATE (HARD) benutzt dieselben Zwischenzustände. Die Folge von Zustandsänderungen endet jedoch nicht in SUSPENDED_INTERN, sondern sie läuft weiter über EXTERNALIZING. Der ConTract–Manager speichert deshalb das gewünschte Ziel eine Zustandsänderung in der Variablen *desired_action*. Am Wert dieser Variablen kann er während der Verarbeitung erkennen, ob sich der ConTract noch in einem Zwischenzustand oder bereits im gewünschten Zielzustand einer Folge von Zustandänderungen befindet.

Realisierung der ConTract–Client–Kommandos

Die beiden folgenden Tabellen beschreiben die Abbildung der ConTract–Client–Kommandos auf die Verarbeitungsfunktionen.

ConTract–Zustand	**activate**	**suspend**	**resume**
(ConTract nicht vorhanden)	CM_init_and_start_cont ()	error–4	error–4
INIT	error–19	error–1	error–1
RUNNING_FORWARD	error–19	CM_start_suspend_cont ()	error–2
TERMINATED	error–19	error–3	error–3
SUSPENDING_SOFT	error–19	error–5	error–5
SUSPENDING_HARD	error–19	error–5	error–5
SUSPENDED_INTERN	error–19	error–6	CM_resume_cont_and_reply ()
SUSPENDED_EXTERN	error–19	error–6	CM_resume_cont_and_reply ()
RESUMING_FORWARD	error–19	error–7	error–7
EXTERNALIZING	error–19	error–8	error–8
INTERNALIZING	error–19	error–9	error–9
EXPORTING	error–19	error–10	error–10
IMPORTING	error–19	error–10	error–10
INIT_COMPENSATION	error–19	error–11	error–11
COMPENSATING	error–19	error–12	error–12
COMPENSATION_ERROR	error–19	error–13	error–13
COMPENSATED	error–19	error–14	error–14

ConTract–Zustand	**cancel**	**migrate**	**status**
(ConTract nicht vorhanden)	error–4	error–4	error–4
INIT	error–1	error–1	error–1
RUNNING_FORWARD	CM_start_cancel_cont ()	CM_start_migration_of_running_cont ()	CM_reply_cont_status ()
TERMINATED	error–3	error–3	error–3
SUSPENDING_SOFT	error–5	error–5	error–5
SUSPENDING_HARD	error–5	error–5	error–5
SUSPENDED_INTERN	CM_start_cancel_cont ()	CM_migrate_suspended_cont ()	CM_reply_cont_status ()
SUSPENDED_EXTERN	CM_start_cancel_cont ()	CM_migrate_suspended_cont ()	CM_reply_cont_status ()
RESUMING_FORWARD	error–7	error–7	error–7
EXTERNALIZING	error–8	error–8	error–8
INTERNALIZING	error–9	error–9	error–9
EXPORTING	error–10	error–10	error–10
IMPORTING	error–10	error–10	error–10
INIT_COMPENSATION	error–11	error–11	error–11
COMPENSATING	error–12	error–12	CM_reply_cont_status ()
COMPENSATION_ERROR	error–13	error–13	CM_reply_cont_status ()
COMPENSATED	error–14	error–14	CM_reply_cont_status ()

Die Terminierungsmeldung einer Step–Instanz durch den ConTract–Processing–Monitor

Die step_finished Meldung des ConTract–Processing–Monitors liefert einen Returncode, der über Erfolg oder Mißerfolg der Ausführung einer Step–Instanz Auskunft gibt. Folgende Returncodes sind derzeit vorgesehen:

- *done*
  Die Step–Instanz wurde erfolgreich ausgeführt.
- *aborted*
  Die Transaktion der Step–Instanz wurde abgebrochen.
- *entry_invariant_failed*
  Die Eingangsinvariante ist gescheitert.
- *failed*
  Die Ausführung der Step–Instanz ist aus anderen Gründen gescheitert[6]

Die folgende Tabelle stellt die Verzweigung der Verarbeitung, in Abhängigkeit vom Returncode, dar.

[6]Zum Beispiel Ausfall einer Ausführungskomponente.

ConTract–Zustand	done	aborted	entry_ invariant_failed	failed
(ConTract nicht vorhanden)	error–4	error–4	error–4	error–4
INIT	error–15	error–15	error–15	error–15
RUNNING_FORWARD	CM_trigger_step_ and_schedule ()	error–16	CM_trigger_step_ and_schedule ()	CM_start_comp()
TERMINATED	error–15	error–15	error–15	error–15
SUSPENDING_SOFT	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()
SUSPENDING_HARD	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()	CM_finished_ step_in_susp ()
SUSPENDED_INTERN	error–15	error–15	error–15	error–15
SUSPENDED_EXTERN	error–15	error–15	error–15	error–15
RESUMING_FORWARD	error–15	error–15	error–15	error–15
EXTERNALIZING	error–15	error–15	error–15	error–15
INTERNALIZING	error–15	error–15	error–15	error–15
EXPORTING	error–15	error–15	error–15	error–15
IMPORTING	error–15	error–15	error–15	error–15
INIT_COMPENSATION	error–15	error–15	error–15	error–15
COMPENSATING	CM_schedule_ next_ comp_steps ()	error–16	CM_error_in_ comp ()	CM_error_in_ comp ()
COMPENSATION_ERROR	CM_finished_ step_in_ comperr ()	error–16	CM_finished_ step_in_ comperr ()	CM_finished_ step_in_ comperr ()
COMPENSATED	error–15	error–15	error–15	error–15

Durchführung der Migration

An einer Migration sind immer zwei ConTract–Manager beteiligt. Möchte ein ConTract–Manager einen ConTract migrieren, so schickt er die Nachricht *migrate_order* an den Empfänger der Migration. Der Empfänger bearbeitet den Migrationswunsch mit der Nachrichtenfunktion *CM_migrate_order*. Nachdem der Empfänger den ConTract übernommen hat, schickt er an den Sender die Quittungsnachricht *seized*. Die Nachrichtenfunktion *CM_seized* bearbeitet diese Nachricht beim Sender. Der Sender kann nun seine internen Verwaltungsdaten freigeben und den migrierten ConTract aus seinem Speicher löschen. Die folgende Tabelle beschreibt die Verarbeitung dieser Migrations–Nachrichten.

ConTract–Zustand	**migrate–order**	**seized**
(ConTract nicht vorhanden)	CM_migrate_order_from_cm ()	error–4
INIT	error–18	error–17
RUNNING_FORWARD	error–18	error–17
TERMINATED	error–18	error–17
SUSPENDING_SOFT	error–18	error–17
SUSPENDING_HARD	error–18	error–17
SUSPENDED_INTERN	error–18	error–17
SUSPENDED_EXTERN	error–18	error–17
RESUMING_FORWARD	error–18	error–17
EXTERNALIZING	error–18	error–17
INTERNALIZING	error–18	error–17
EXPORTING	error–18	CM_cont_exported_to_cm ()
IMPORTING	error–18	error–17
INIT_COMPENSATION	error–18	error–17
COMPENSATING	error–18	error–17
COMPENSATION_ERROR	error–18	error–17
COMPENSATED	error–18	error–17

Die Verarbeitungsfunktionen des ConTract–Managers

Die Verarbeitungsfunktionen werden hier kurz beschrieben. Ihre Aufrufe sind in den Tabellen zur Abbildung der Nachrichtenfunktionen verzeichnet.

CM_cont_exported_to_cm
Abschluß der Migration auf Senderseite. Die Daten des ConTracts werden, nach der erfolgreichen Migration, gelöscht.

CM_error_in_comp
Bearbeitung eines Fehlers während der Kompensation.

CM_finished_step_in_comperr
Eingehende step_finished Meldungen im Zustand COMPENSATION_ERROR werden abgefangen und gespeichert. Nach der Behebung des Fehlers kann die Migration an diesen Stellen fortgesetzt werden.

CM_finished_step_in_susp
Step_finished Meldungen während der Suspendierung werden gesammelt und in der Suspend Liste gespeichert.

CM_init_and_start_cont
Erzeugung und Start eines neuen ConTracts.

CM_migrate_order_from_cm
Bearbeitung einer Aufforderung zur Übernahme eines ConTracts.

CM_migrate_suspended_cont
Migration eines ConTracts, der bereits suspendiert ist.

CM_reply_cont_status
Zustand eines ConTracts an den ConTract–Client melden.

CM_resume_cont_and_reply
ConTract aus dem Zustand SUSPENDED wieder starten und eine entsprechende Nachricht an den ConTract–Client schicken.

CM_schedule_next_comp_steps
Kompensations–Steps starten.

CM_start_cancel_cont
Abbruch eines ConTracts durch Kompensation einleiten.

CM_start_comp
Kompensation eines ConTracts im Zustand RUNNING_FORWARD anstoßen.

CM_start_migration_of_running_cont
Migration eines ConTracts im Zustand RUNNING_FORWARD anstoßen.

CM_start_suspend_cont
Suspendierung eines ConTracts im Zustand RUNNING_FORWARD anstoßen.

CM_trigger_step_and_schedule
Triggerung der Nachfolge–Events einer Step–Instanz, die korrekt beendet wurde. Start der daraus hervorgehenden Step–Instanzen.
Diese Funktion behandelt die normale Ausführung eines ConTracts.

4.1.4 Fehlerbehandlung im ConTract–Manager

Die Fehlerkodes, die bei der Verarbeitung der eingehenden Nachrichten auftreten können, haben folgende Bedeutungen:

error–1	Kommando vom ConTract–Client während die Aktivierung noch läuft[7].
error–2	Empfang des RESUME–Kommandos vom ConTract–Client obwohl der ConTract schon läuft.
error–3	ConTract–Client–Kommando im Zustand TERMINATED.
error–4	Nachricht für unbekannten ConTract (Der ConTract–Manager hat keinen ConTract in Bearbeitung).
error–5	ConTract–Client–Kommando während die Suspendierung läuft[7].
error–6	SUSPEND–Kommando obwohl der ConTract bereits suspendiert ist.

[7]ConTract–Client–Kommandos sind durch ConTract–Client–Kommandos nicht unterbrechbar — auch nicht durch STATUS.

error–7	ConTract–Client–Kommando während RESUME läuft[7].
error–8	ConTract–Client–Kommando während Externalisierung läuft[7].
error–9	ConTract–Client–Kommando während Internalisierung läuft[7].
error–10	ConTract–Client–Kommando während Migration läuft[7].
error–11	ConTract–Client–Kommando während Kompensation gestartet wird[7].
error–12	Unerlaubtes ConTract–Client–Kommando im Zustand COMPENSATING.
error–13	Unerlaubtes Kommando vom ConTract–Client im Zustand COMPENSATION_ERROR.
error–14	Unerlaubtes ConTract–Client–Kommando im Zustand COMPENSATED.
error–15	Der ConTract hat zur Zeit keinen aktiven Step (Der ConTract–Processing–Monitor schickt step_finished Meldungen obwohl keine Steps aktiv sind!).
error–16	Eine Step–Instanz endet mit dem Returncode ABORT, ohne daß ein ABORT–Kommando abgeschickt wurde.
error–17	Seized–Nachricht, obwohl der ConTract gerade nicht migriert wird.
error–18	Der Sender einer Migration will einen ConTract migrieren, der beim Empfänger schon in Bearbeitung ist (erkennbar anhand der *cid*).
error–19	Der ConTract soll aktiviert werden, obwohl er bereits läuft.

Die Fehler werden von den Nachrichtenfunktionen an den ConTract–Client gemeldet. Nachrichten, die einen Fehler ausgelöst haben, werden verworfen. Die Verarbeitung des ConTracts wird dann normal fortgesetzt.

4.2 Datenstrukturen

Die Daten des ConTract–Managers lassen sich in verschiedene Abstraktionsebenen gliedern. Dieses Schichtenmodell der Datenstrukturen ist in Abbildung 4.4 dargestellt.

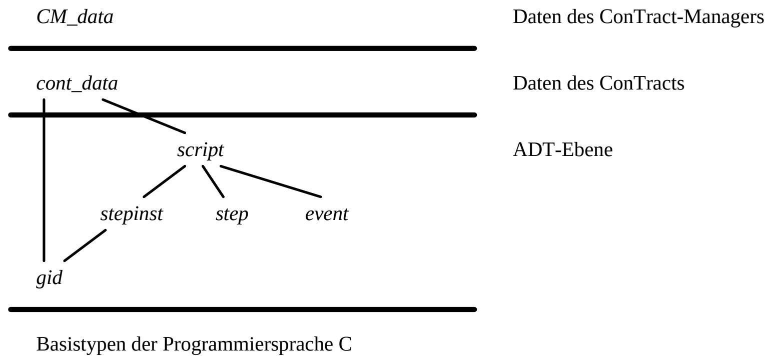


Abbildung 4.4: Schichtenmodell der Daten

Im folgenden möchte ich die einzelnen Ebenen und ihre innere Struktur besprechen. Die Betrachtung soll dabei von den logisch tieferliegenden zu den höherliegenden Schichten erfolgen.

Damit wird jeweils die Voraussetzung zum Verständnis der nächsthöheren Abstraktionsebene geschaffen. Eine grundlegende Kenntnis der Basistypen der Programmiersprache C setze ich dabei voraus. Verwendet wurden einfache Typen, wie *long*, *char* und *Zeigertypen*, sowie zusammengesetzte Typen (*struct*).

4.2.1 Abstrakte Datentypen

Zur Darstellung des ConTract–Skripts wurden im wesentlichen abstrakte Datentypen[8] und Zeigerstrukturen verwendet. Die abstrakten Datentypen wurden

(a) für Datenstrukturen gewählt, die von mehreren Komponenten — und das heißt auch von mehreren Programmierern — verwendet werden und

(b) für Datenstrukturen, die später voraussichtlich Änderungen und Erweiterungen unterworfen sein werden.

Durch die definierte Schnittstelle des ADTs werden Details der Implementierung vor den Anwendungsprogrammen und vor den anderen Programmierern im Projekt verborgen. Damit wird die Änderungsfreundlichkeit des Anwendungsprogramms erreicht. Eine einführende Besprechung des Konzepts der abstrakten Datentypen befindet sich in Kapitel 5.

Folgende abstrakte Datentypen sind im Entwurf des ConTract–Managers enthalten:

1. *gid*

   Der gid[9] definiert eine Übermenge aller global eindeutigen IDs im APRICOT–System. Hierunter fallen unter anderem ConTract–ID (cid), Step–Instanz–ID (stepinst_id) , Transaktions–ID (gtrid) und IDs für die einzelnen Prozesse im System.

   Der gid enthält Daten, die seine globale Eindeutigkeit gewährleisten. Außerdem verfügt er über eine Variable, die angibt, um welche Art der oben genannten IDs es sich bei einer bestimmten Instanz des gid handelt (cid, gtrid, . . .).

2. *stepinst*

   Dieser Datentyp speichert die Step–Instanzen des ConTract–Managers. Der gid geht als Step–Instanz–ID in seine Struktur mit ein.

3. *step*

   Der abstrakte Datentyp step dient der Darstellung der Steps im Prädikat–Transitionsnetz des ConTract–Skripts.

4. *event*

   Mit dem Datentyp event werden die Events des Prädikat–Transitionsnetzes realisiert.

[8] Abstrakter DatenTyp = ADT.
[9] global id

5. *script*

   Im abstrakten Datentyp script sind stepinst, step und event als Unter–ADTs enthalten. Neben der internen Repräsentation des Prädikat–Transitionsnetzes enthält script auch die definierten Transaktionsbeziehungen und die Verwaltung der Step–Instanzen.

   Dieser Datentyp hat eine sehr komplexe Struktur. Er enthält mit dem Prädikat–Transitionsnetz und den Ausführungsaufträgen den wichtigsten und größten Teil der ConTract–Daten.

4.2.2 Zustandsdaten eines ConTracts

Auf dieser Ebene sind alle Daten, die einen ConTract beschreiben, zusammengefaßt[10]. Im einzelnen sind dies:

1. *Cid*

   Der cid ist eine Instanz des abstrakten Datentyps gid.

2. *Suspend_list*

   Die Suspend Liste enthält alle Step–Instanzen, deren Ausführungsende bei der Suspendierung des ConTracts durch die step_finished Nachrichten gemeldet wurde. Außerdem wird für jede Step–Instanz der Returncode der step_finished Meldung gespeichert. Der ConTract–Manager benötigt diese Daten beim Wiederanfahren des ConTracts. Er kann dann anhand des Returncodes entscheiden, ob die Step–Instanz erneut ausgeführt werden muß oder ob die Nachfolge–Events des zugehörigen Steps getriggert werden können.

3. *Cont_script*

   Diese Variable ist eine Instanz des abstrakten Datentyps script. Die statischen und dynamischen Daten des Skripts sind in den ADTs step, event und stepinst enthalten. Alle Operationen auf dem Skript, wie Triggerung der Steps und Events oder die Erzeugung und Verwaltung von Aufträgen, werden durch die Schnittstelle des abstrakten Datentyps script zur Verfügung gestellt.

4. *Running_steps*

   Die momentan aktiven Step–Instanzen werden in dieser Liste verwaltet.

5. *Contract_state*

   Diese Zustandsvariable beschreibt den internen Verarbeitungszustand des ConTracts aus Sicht des ConTract–Managers. Die möglichen Werte und Zustandsübergänge sind in Abbildung 4.1 auf Seite 39 dargestellt.

6. *Desired_action*

   Die Variable desired_action definiert eine Folge von Zustandsänderungen bei der Verarbeitung eines ConTracts[11].

[10]Vgl. Abschnitt 2.8

[11]Siehe dazu Abschnitt 4.1.3 auf Seite 43.

7. *Statistische Daten*

   Hierunter fallen Daten wie die Startzeit des ConTracts, die Anzahl der gerade aktiven Step–Instanzen usw.

4.2.3 Daten des ConTract–Managers

Diese Abstraktionsebene faßt alle Daten zusammen, die vom ConTract–Manager verarbeitet werden. Im wesentlichen sind dies die im vorigen Abschnitt beschriebenen Daten des ConTracts. Außerdem erhält der ConTract–Manager eine global eindeutige Komponenteninstanz–ID. Diese ID wird von der Prozeßverwaltung des APRICOT–Systems vergeben und dient zur Identifikation der einzelnen Prozesse im System. Schließlich speichert der ConTract–Manager noch statistische Daten, zum Beispiel über Art und Anzahl der verarbeiteten Nachrichten.

4.3 Ein- und Auslagerung der ConTract–Daten

Im Zustand EXTERNALIZING werden alle Daten des ConTracts auf ein externes Speichermedium (Festplatte) ausgelagert. Das Einlesen in den Hauptspeicher erfolgt im Zustand INTERNALIZING.

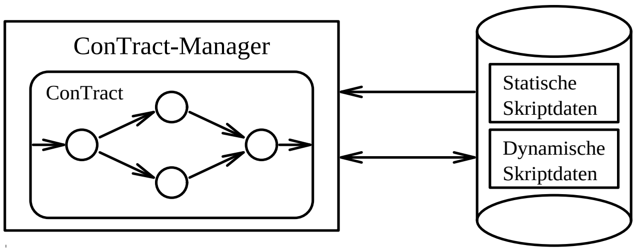


Abbildung 4.5: Ein- und Auslagerung der ConTract–Daten

Für die Schnittstelle zur Speicherung der Daten auf der Festplatte standen beim Entwurf des Prototyps zwei Möglichkeiten zur Auswahl.

1. *Speicherung in Dateien des zugrundeliegenden Betriebssystems.*

   Diese Art der Speicherung ist zwar sehr laufzeit- und speichereffizient. Alle Maßnahmen zur Sicherung der Daten — insbesondere Recovery und Transaktionsschutz — müssen hier jedoch vom ConTract–Manager selbst getroffen werden.

2. *Speicherung in einem relationalen Datenbanksystem unter SQL.*

   Die Speicherung ist hier zwar weniger effizient als bei 1. Der ConTract–Manager braucht jedoch keine aufwendigen Maßnahmen zum Schutz der Daten zu treffen. Außerdem ist mit der SQL–Schnittstelle die leichte Übertragbarkeit auf andere Betriebsysteme und Programmiersprachen gewährleistet.

Aufgrund der beschriebenen Vorteile wurde der zweite Ansatz gewählt.

Für den Umfang der Daten, die extern gespeichert werden, stehen wiederum zwei Alternativen zur Auswahl:

1. *Trennung der statischen und dynamischen Daten. Nur die dynamischen Daten werden extern abgespeichert. Die statischen Daten müssen nicht bei jeder Externalisierung gespeichert werden, da sie unverändert (statisch) im Zwischen–Code vorliegen.*

   Die Speicherung der ConTract–Daten ist hier schnell und speichereffizient. Allerdings muß sichergestellt sein, daß sich der Zwischen–Code während der Laufzeit des ConTracts nicht verändert — zum Beispiel durch Änderung und erneute Übersetzung des Skripts. Dies kann auch durch die Einführung einer Versionsverwaltung für den Zwischen–Code erreicht werden.

2. *Gemeinsame Speicherung der statischen und dynamischen Daten.*

   Der Aufwand zur Speicherung ist erheblich größer als bei 1. Allerdings ist hier keine Versionsverwaltung für den Zwischen–Code notwendig. Für jeden ConTract werden alle Daten, die er später bei der Internalisierung gebraucht werden, in einem eigenen Datenpaket gespeichert.

Aufgrund der Effizienzvorteile und des geringeren Implementierungsaufwands wurde für den Prototyp die erste Alternative gewählt.

Kapitel 5

Das Implementierungskonzept abstrakter Datentypen

Dieses Kapitel liefert eine einführende Besprechung zum Konzept des abstrakten Datentyps[1]. Außerdem wird ein Implementierungskonzept für die Programmiersprache C vorgestellt und diskutiert. Dieses Konzept wird innerhalb des Prototypsystems APRICOTS zur Implementierung abstrakter Datentypen verwendet.

Ziel bei der Anwendung abstrakter Datentypen, ist das *Information Hiding*[2]. Die Implementierung eines Datentyps und der auf ihm möglichen Operationen bleibt, gegenüber dem Programm das den Datentyp verwendet, verborgen. Das Programm greift nur über eine *definierte Schnittstelle* — die Operationen des ADTs — auf die Daten zu. Die Spezifikation eines abstrakten Datentyps besteht aus:

1. Dem Namen des Datentyps
2. Den erlaubten Operationen auf dem Datentyp

5.1 Vorteile für die Software–Entwicklung

Für die Programmierung eines ADTs ist ein einzelner Programmierer[3] aus dem Projektteam zuständig. Er implemeniert die Schnittstelle, das heißt die Operationen auf dem Datentyp, und die interne Struktur der Daten. Wird diese Schnittstelle bei Änderungen im ADT nur erweitert, so müssen die Anwendungsprogramme nicht angepaßt werden — sie sind nach wie vor lauffähig. Teuer, im Sinne umfangreicher Anpassungen von Anwendungsprogrammen, werden Änderungen im ADT nur dann, wenn bisher verfügbare Funktionen wegfallen oder sich ändern. Falls die

[1]...oder kurz ADT.

[2]Es wird so viel Information wie möglich in einem Modul verborgen gehalten.

[3]Oder eine kleine Gruppe von Programmierern.

Schnittstelle erhalten bleibt, wirken sich Änderungen in der Struktur der Daten überhaupt nicht auf das Anwendungsprogramm aus. Insbesondere kann eine hardwareabhängige, physische Repräsentation der Daten hinter den Operationen des ADTs verborgen werden. Wichtige Vorteile bei der Verwendung von abstrakten Datentypen sind damit:

- Änderungsfreundlichkeit der Anwendungsprogramme.
- Übertragbarkeit durch weitgehende Unabhängigkeit von den Gegebenheiten des Betriebsystems und der Hardware.
- Die Vereinbarung einer Schnittstelle für Datenzugriffe verhindert, daß unzulässige Operationen auf den Daten ausgeführt werden. Fehlerhafte Zugriffe auf Daten, insbesondere bei komplexen Strukturen, werden weitgehend unterbunden. Dies ist besonders wichtig, wenn im Projektteam mehrere Programmierer mit denselben Datenstrukturen umgehen.

5.2 Ein konkreter Implementierungsvorschlag in C

Für jeden ADT werden zwei Dateien angelegt. Die Grundstruktur dieser ADT–Dateien wird im Anhang A beschrieben. Die Header–Datei `adt_adtname.h`[4] enthält eine Beschreibung der externen Schnittstelle des ADTs. Der eigentliche Quelltext des ADTs befindet sich in der Datei `adt_adtname.c`. Hier sind die Datentypen und Funktionen zur Implementierung des ADTs und seiner Operationendefiniert.

Ziel des Konzepts des abstrakten Datentyps ist es, die Datenstruktur vor den anderen Modulen des Systems zu verbergen und Zugriffe nur über definierte Operationen zuzulassen. Um dieses Ziel mit der Implementierungssprache C zu erreichen wird folgendes Konzept verwendet:

1. Die Strutur des ADTs wird nur in der Datei `adt_adtname.c`, als Typ PRIVATE_ADT_adtname, definiert. Das ADT–Modul selbst greift über einen Zeiger auf diesen Typ zu. In der Schnittstelle wird ein Synonym für diesen Zeigertyp bereitgestellt, das extern zugänglich ist. Die Datei `adt_adtname.h` wird extern als Schnittstellenbeschreibung und innerhalb des ADT–Moduls zur Prüfung der Parametertypen verwendet. Deshalb muß im ADT–Modul eine #define Konstante (INSIDE_ADT_adtname) vereinbart werden, anhand deren der C–Compiler erkennt, ob der Header vom ADT–Modul selbst oder von einem Anwendungsprogramm eingebunden wird.

   Der folgende Abschnitt der Header–Datei beschreibt diesen Mechanismus der Typdefinition:

   ```
   #ifdef INSIDE_ADT_<adtname>
      /* privater Typ im ADT-Modul */
      typedef  PRIVATE_ADT_<adtname> *  ADT_<adtname>;
   ```

[4] Der Teiltext `adtname` ist, auch im folgenden, jeweils durch den Namen des implementierten ADTs zu ersetzen.

```
#else
   /* public pointer-type */
   typedef  char *  ADT_<adtname>;
#endif
```

2. Interne Unterfunktionen der ADT–Operationen werden als Funktionen der Speicherklasse `static` deklariert und sind damit nur lokal verfügbar.

3. Die Operationen des ADT werden als Funktionen der Speicherklasse `extern` deklariert und mit ihrer Parameterliste in der Schnittstelle aufgeführt. Sie sind damit auch außerhalb des Moduls verwendbar.

4. Jeder ADT verfügt zumindest über zwei Funktionen zur Erzeugung und Freigabe seiner Instanzen:

   (a) Der *Konstruktor* fordert Speicher für eine Instanz des ADTs an. Er liefert als Ausgabedatum einen Zeiger auf die bereitgestellte Instanz.

   (b) Eine Instanz des ADTs wird mit dem *Destruktor* wieder freigegeben. Der Zeiger auf die Instanz enthält nach dem Aufruf des Destruktors den Wert NULL.

   Bevor irgendeine andere Operation auf einer Instanz des ADTs durchgeführt werden kann, muß diese Instanz mit dem Konstruktor erzeugt werden. Wird die Instanz nicht mehr benötigt, so kann das Anwendungsprogramm sie mit dem Destruktor wieder freigeben.

5. Für die Funktionsnamen der ADT–Operationen gelten bestimmte Namenskonventionen. Der Text `ADT_adtname_` wird dabei immer als Präfix verwendet.

ADT_adtname_construct	Der Konstruktor des ADTs.
ADT_adtname_destruct	Der Destruktor des ADTs.
ADT_adtname_get_var	Diese Funktion liefert den Wert der Variablen *var* aus der Datenstruktur des ADTs.
ADT_adtname_set_var	Diese Funktion belegt die Variable *var* in der Datenstruktur des ADTs mit einem Wert, der als Parameter übergeben wird.
ADT_adtname_equal	Mit dieser Funktion können zwei Instanzen des ADTs auf Gleichheit geprüft werden.
ADT_adtname_copy	Alle Werte einer Instanz können mit dieser Funktion in eine andere Instanz kopiert werden.
ADT_adtname_init	Die Initialisierung einer Instanz wird mit dieser Funktion durchgeführt.

5.3 Kritik an der Implementierung in C

Die Programmiersprache C ist nicht für die Implementierung von abstrakten Datentypen entworfen worden. In C ist es zum Beispiel nicht möglich, die Struktur eines Datentyps in einem

Modul zu verbergen, dessen Namen jedoch trotzdem global bekannt zu machen. Um die Datenstruktur dennoch zu verbergen, dürfen Anwendungsprogramme nur über Zeiger auf ADTs zugreifen.

Trotz der unbestreitbaren Vorteile beim Einsatz abstrakter Datentypen, möchte ich auf folgende Kritikpunkte des vorgestellten Konzepts hinweisen:

1. Die Implementierung komplexer Zeigeroperationen von Hand ist sehr fehleranfällig.
2. Wird der Destruktor eines abstrakten Datentyps irrtümlicherweise nicht aufgerufen, so treten Fehler in der Speicherverwaltung auf — der Umfang des belegten Speichers wird immer größer.
3. Bei der Unterschlagung des Konstruktors erfolgt sogar ein Zugriff über einen undefinierten Zeiger. Ein solcher Fehler ist im Anwendungsprogramm nur sehr schwer zu lokalisieren.
4. Die *Vererbung* von Operationen geschachtelter ADTs muß von Hand programmiert werden. C stellt dafür keinen Mechanismus bereit. Hier wäre der Einsatz einer objektorientierten Programmiersprache sicherlich sinnvoller.

5.4 Abstrakte Datentypen am Beispiel der Step–Instanz

Die Erzeugung und Verwaltung von Step–Instanzen wurde in Abschnitt 3.3 ausführlich besprochen. Hier möchte ich nun die Implementierung eines ADTs am Beispiel der Step–Instanz vorstellen.

5.4.1 Die Datenstruktur der Step–Instanz

Der ConTract–Manager und die anderen Komponenten des APRICOT–Systems verwenden einen Zeiger vom Typ `ADT_stepinst` um Instanzen dieses ADTs zu handhaben. Die Datenstruktur der Step–Instanz ist im ADT–Modul `adt_stepinst.c` definiert. Sie enthält Unterstrukturen, die selbst wieder ADTs sind. Es handelt sich dabei global eindeutige IDs vom Typ `ADT_gid` zur Identifikation von Objekten im APRICOT–System.

Die Typdefinition für die Step–Instanz sieht wie folgt aus:

```
struct {
   ADT_gid        stepinst_id;  /* global eindeutige Kennung */
   ADT_gid        gtrid;        /* Transaktions-ID */
   ADT_gid        cid;          /* ConTract ID */

   /* Daten des zugeordneten Steps: */
   t_step_id      step_id;
```

```
    t_step_name    step_name;    /* Logischer Name des Steps */
    t_inv          entry_inv, exit_inv;
    t_policy       policy;
    t_sched_hints  sched_hints;

    /* Versionskennung der Stepinstanz: */
    long           version;    /* Versionsnummer der Stepinstanz */
    long           par_index;  /* parallele Instanzen einer Version */
}
```

Die Datentypen `t_step_id`, `t_step_name`, `t_inv`, `t_policy` und `t_sched_hints` sind in einer Header–Datei des APRICOT–Prototyps definiert. Es handelt sich dabei um Synonyme für einfache Typen, wie Ganzzahlen und Strings.

5.4.2 Operationen auf Step–Instanzen

Für die Step–Instanz stehen eine ganze Reihe von Operationen zur Verfügung, die sich in verschiedene Gruppen aufteilen lassen.

Erzeugung und Freigabe

`ADT_stepinst_construct`

Der Konstruktor zur Erzeugung einer Step–Instanz.

`ADT_stepinst_destruct`

Der Destruktor zur Freigabe einer Step–Instanz.

Ausgabe zu Testzwecken

`ADT_stepinst_fprintf`

Diese Funktion druckt den Inhalt einer Step–Instanz in ASCII–Notation auf eine Datei aus. Dadurch kann beim Test des ADT–Moduls oder des Anwendungsprogramms der Inhalt einer Step–Instanz überprüft werden.

Ausgabe der einzelnen Variablen

`ADT_stepinst_get_stepinst_id`
`ADT_stepinst_get_gtrid`
`ADT_stepinst_get_cid`

Die *get–id* Funktionen liefern jeweils eine Kopie der Unter–ADTs `stepinst_id`, `gtrid` und `cid` vom Typ `ADT_gid`.

```
ADT_stepinst_get_step_id
ADT_stepinst_get_step_name
ADT_stepinst_get_entry_inv
ADT_stepinst_get_exit_inv
ADT_stepinst_get_policy
ADT_stepinst_get_sched_hints
ADT_stepinst_get_version
ADT_stepinst_get_par_index
```

Diese Gruppe von Funktionen kopiert den Inhalt der jeweiligen Variablen und übergibt die Werte an das Anwendungsprogramm.

Speicherung der IDs in einer Datenbank

```
ADT_stepinst_get_dbstr_stepinst_id
ADT_stepinst_get_dbstr_gtrid
ADT_stepinst_get_dbstr_cid
```

Zur Speicherung der IDs vom Typ `ADT_gid` in einer Datenbank werden die Inhalte der jeweiligen ADTs in ASCII–Notation in einem String abgelegt. Dieser String kann dann zum Beispiel in einem Attribut einer relationalen Datenbank gespeichert werden. Die Verwendung eines Strings zur Speicherung hat den Vorteil, daß Strukturänderungen im ADT gid keine Strukturänderungen der Datenbank nach sich ziehen.

Belegung der Variablenwerte

```
ADT_stepinst_set_stepinst_id
ADT_stepinst_set_gtrid
ADT_stepinst_set_cid
```

Die IDs werden mit den Werten eines übergebenen IDs überschrieben.

```
ADT_stepinst_set_stepdata
ADT_stepinst_set_version
ADT_stepinst_set_par_index
```

Mit diesen Funktionen werden die jeweiligen Variablen der Step–Instanz auf Werte gesetzt, die als Parameter übergeben werden.

Kopie einer Step–Instanz

```
ADT_stepinst_copy
```

Eine Kopie einer Step–Instanz kann mit dieser Funktion erstellt werden. Dazu werden zwei Step–Instanzen übergeben, die zuvor beide mit dem Konstruktor erzeugt werden müssen.

Vergleich des Step–Instanz–IDs

`ADT_stepinst_compare_stepinst_id`

Der `stepinst_id` einer Step–Instanz wird mit einem übergebenen ID verglichen. Die Funktion prüft, ob die beiden IDs gleiche Werte beinhalten.

Kapitel 6

Ausblick

Manche Probleme bei der Spezifikation und dem Entwurf des ConTract–Managers konnten in dieser Arbeit nur kurz angesprochen werden. Zur Weiterentwicklung des Prototyps sind folgende Nachfolgearbeiten denkbar:

1. *Kompensation von ConTracts*

   In Abschnitt 2.3 wurden bereits die grundlegenden Mechanismen und Problemstellungen der Kompensation besprochen. Der Entwurf in Kapitel 4 enthält außerdem einen Ansatz zur Integration der Kompensationsaufgabe in den ConTract–Manager. Hier wäre insbesondere die Frage zu erörtern welchen Synchronisationsbedingungen die Ausführung der Kompensations–Steps genügen muß. Welche Steps können parallel ausgeführt werden, welche müssen sequentiell ablaufen?

   Nach der Auswahl und Ausarbeitung eines Kompensationsverfahrens kann die Kompensation in den bestehenden Prototyp des ConTract–Managers integriert werden.

2. *Recovery im ConTract–Manager*

   Um die Wiederaufsetzbarkeit zu realisieren, wird die Verarbeitung eines ConTracts in atomare Einheiten gegliedert. Der Beginn dieser Einheiten definiert jeweils die Aufsetzpunkte für die Recovery. Der Entwurf in Kapitel 4 beschreibt bereits Grundeinheiten der ConTract–Verarbeitung, die zur Implementierung der Recovery verwendet werden können. Diese Grundeinheiten sind die Nachrichtenfunktionen, die jeweils eine empfangene Nachricht verarbeiten. Die Nachrichtenfunktionen müssen alle anfallenden Ausgabedaten sammeln und jeweils am Ende ihrer Ausführung ausgeben und protokollieren.

   Die entsprechenden Verfahren, zur Datensicherung und zum Wiederaufsetzen nach einem Systemausfall, müssen entworfen und in den Prototyp eingebunden werden.

3. *Entwurf und Implementierung eines Logging–Systems*

   Art und Umfang der Log–Daten und der Zugriffsfunktionen müssen spezifiziert werden. Außderdem ist eine geeignete Repräsentation der Daten auf einem externen Speichermedium zu entwerfen.

Anhang A

Quelltexte für abstrakte Datentypen

A.1 Die Datei adt_adtname.h

```
/*
** Datei. . . . . . : adt_<adtname>.h
** Headername . . . : <longname> header
** Kurzbeschreibung : Externe Schnittstelle des ADT <adtname>
**
** Autor. . . . . . : peter wahl
**
** Aenderungen
** Datum        Ver  getestet   Zweck
** <datum>      1.0  nein       erste offizielle Version
*/

#ifndef IN      /* 'syntactic sugar' fuer die Parameter-Uebergabe */
#define IN
#define OUT
#define INOUT
#endif


/* ------------------------------------------------------------------------------
** Typ(en)
*/

#ifdef INSIDE_ADT_<adtname>
   /* privater Typ im ADT-Modul */
   typedef  PRIVATE_ADT_<adtname> *  ADT_<adtname>;
#else
```

```
   /* public pointer-type */
   typedef  char *  ADT_<adtname>;
#endif


/* ---------------------------------------------------------------------------
** Funktion(en) - Operationen auf dem ADT
*/

extern  int     ADT_<adtname>_construct (OUT ADT_<adtname> *ptr_ADT);
extern  void    ADT_<adtname>_destruct (INOUT ADT_<adtname> *ptr_ADT);
extern  void    ADT_<adtname>_fprintf (IN FILE *stream,IN ADT_<adtname> ADT,
                IN char text[],IN char indent[]);
```

A.2 Die Datei adt_adtname.c

Die Funktion `ADT_adtname_fprintf` druckt den gesamten ADT in ASCII–Notation auf eine Datei aus. Damit kann beim Test die Korrektheit der Operationen überprüft werden.

```
/* ---------------------------------------------------------------------------
** Datei. . . . . . : adt_<adtname>.c
** Modulname. . . . : <longname>
** Kurzbeschreibung : Abstrakter Datentyp fuer ...
**
** Autor. . . . . . : peter wahl
** Testumgebung . . : adt_<adtname>_test.c
**
** Aenderungen
** Datum        Ver  getestet   Zweck
** <datum>      1.0  nein       erste offizielle Version
** ------------------------------------------------------------------------ */
#include <stdio.h>
#include <string.h>
#include <malloc.h>


/* ---------------------------------------------------------------------------
** PRIVATE type(s)
*/
typedef <type> <typename>;

/* ---------------------------------------------------------------------------
```

```
** PRIVATE ADT type
** Dieser Typ definiert die eigentliche (Basis-) Datenstruktur des ADT.
** Alle Zugriffe auf den ADT erfolgen ueber Zeiger auf diesen Typ.
** (siehe header-file)
*/
typedef <type> PRIVATE_ADT_<adtname>;

/* ------------------------------------------------------------------------------
** PRIVATE data
*/
static  <type>  <var>;

#define  INSIDE_ADT_<adtname>
#include "adt_<adtname>.h"



/* ------------------------------------------------------------------------------
** PRIVATE function(s)
*/

/* ------------------------------------------------------------------------------
** <action>
** RETURN: <value>
*/
static  <type>  <fncname> (<arg>)
IN      <type>  <arg1>;
OUT     <type>  <arg2>;
INOUT   <type>  <arg3>;
{

} /* <fncname> */


/* ------------------------------------------------------------------------------
** PUBLIC function(s)
*/

/* ------------------------------------------------------------------------------
** Konstruktor
** RETURN: OK oder NOMEM
*/
extern  int     ADT_<adtname>_construct (ptr_ADT)
OUT     ADT_<adtname>   *ptr_ADT;
```

```
{
        /* Erzeugung des ADT: */
        *ptr_ADT = calloc (1,sizeof (PRIVATE_ADT_<adtname>));
        if (*ptr_ADT == NULL) { return (NOMEM); }

        /* Alle Werte sind mit calloc auf NULL gesetzt */
        <construct or initialize ADT-elements>

        return (OK);
} /* ADT_<adtname>_construct */

/* ------------------------------------------------------------------------------
** Destruktor
** RETURN: -
*/
extern  void    ADT_<adtname>_destruct (ptr_ADT)
INOUT   ADT_<adtname>   *ptr_ADT;
{
        if (*ptr_ADT == NULL) { return; }

        <destruct ADT-elements>

        /* ADT loeschen: */
        free (*ptr_ADT);
        *ptr_ADT = NULL;
} /* ADT_<adtname>_destruct */

/* ------------------------------------------------------------------------------
** Drucke den ADT auf eine Datei aus
**
** RETURN: OK oder NOMEM
*/
extern  int     ADT_<adtname>_fprintf (stream,ADT,text,indent)
IN      FILE            *stream;
IN      ADT_<adtname>   ADT;
IN      char            text[];
IN      char            indent[];  /* Einruecktext */
{
        char    *my_indent;        /* Einruecktext fuer Unterstrukturen */

        my_indent = calloc (1,strlen(indent)+strlen(INDENT_SPACE)+1);
        if (my_indent == NULL) {
           fprintf (stream,
              "%s<not enough memory to print ADT_<adtname>>\n"
```

```
            ,indent
         );
         return (NOMEM);
      }

      strcpy (my_indent,indent);
      strcat (my_indent,INDENT_SPACE);
         /* vergroessere den Einruecktext */

      fprintf (stream,
         "%sADT_<adtname>>%s\n"
         ,indent,text
      );
      fprintf (stream,
         "%s<adtvalue> = %?\n"
         ,my_indent,ADT-><adtvalue>
            /* Einruecktext vor jeder Zeile */
      );

      free (my_indent);
      return (OK);
} /* ADT_<adtname>_fprintf */
```

A.3 Ein Hauptprogramm, das abstrakte Datentypen benutzt

```
/* ----------------------------------------------------------------------------
** Datei. . . . . . : <main>.c
** Programmname . . : <longname>
** Kurzbeschreibung : ... ein Programm, das abstrakte Datentypen verwendet ...
**
** Autor. . . . . . : peter wahl
**
** Aenderungen
** Datum       Ver  getestet   Zweck
** <datum>     1.0  nein       erste offizielle Version
** ------------------------------------------------------------------------- */
#include <stdio.h>
#include "adt_<adtname>.h"


/* ------------------------------------------------------------------------- */
main (argc,argv,envp)
```

```
        int     argc;
        char    *argv [],*envp;
{
        ADT_<adtname>   my_ADT;

        ADT_<adtname>_construct (&my_ADT);
        ... use ADT ...
        ADT_<adtname>_destruct (&my_ADT);
} /* main */
```

Literaturverzeichnis

[Burk91] Burkert, Holger: *Systemarchitektur und Betriebsmodell für das verteilte Transaktionssystem APRICOTS*, IPVR* 1991, Diplomarbeit (in Vorbereitung)

[BuRai90] Burkert, Holger und Raichle, Bernd: *Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext*, IPVR* 1991, Studienarbeit Nr. 859

[Freis87] Freisleben, Bernhard: *Mechanismen zur Synchronisation paralleler Prozesse*, Informatik–Fachberichte 133, Springer Verlag 1987

[Gray81] Gray, J.: *The transaction concept: virtues and limitations*, Proc. VLDB, Cannes, September 1981

[KerRi83] Kernighan, Brian W. und Ritchie, Dennis M.: *Programmieren in C*, Hanser Verlag 1983

[LaDun90] Laube, Dieter und Dung, Tran Van: *Entwurf und Spezifikation eines prototypischen ConTract–Managers*, IPVR* 1990, Studienarbeit Nr. 875

[LoSch87] Lockemann, P.C. und Schmidt, J.W. (Herausgeber): *Datenbank–Handbuch*, Reihe: Informatik–Handbücher, Springer Verlag 1987

[Lutz91] Lutz, Rainer: *Verteilte Transaktionsverarbeitung mit Transaktionsbeziehungsnetzen*, IPVR* 1991, Diplomarbeit (in Vorbereitung)

[Mas90] Masmanidis, Ioannis: *Syntax und Compiler für eine Sprache zur Spezifikation verteilte und paralleler Abläufe*, IPVR* 1990, Studienarbeit Nr. 897

[Petri62] Petri, C.A.: *Kommunikation mit Automaten*, Dissertation, TH Darmstadt 1962

[Reut89] Reuter, Andreas: *ConTracts: A Means for Extending Control Beyond Transaction Boundaries*, Proc. 2nd. Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1989

[Schü91] Schütz, Holger: *Darstellung und Verwaltung von Beziehungen zwischen Transaktionen*, IPVR* 1991, Studienarbeit (in Vorbereitung)

*Institut für parallele und verteilte Höchstleistungsrechner, Universität Stuttgart.

[Som89] Sommerville, Ian: *Software–Engineering*, third edition, Addison Wesley 1989

[Wäch90] Wächter, Helmut: *ConTracts: A Means for Improving Reliability in Distributed Computing*, IPVR* 1990

[WäRe90] Wächter, Helmut und Reuter, Andreas: *Grundkonzepte und Realisierungsstrategien des ConTract–Modells*, In: Informatik – Forschung und Entwicklung, Springer Verlag 1990

[WäRe91] Wächter, Helmut und Reuter, Andreas: *The ConTract–Model*, to be published in: A.K. Elmagarmid (ed.): *Transaction Models for Advanced Database Applications*, Morgan Kaufmann Publishers, 1991