Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Überblick

1.1 Einführung

Im Rahmen des APRICOT–Systems (APRICOTS = A PRototype Implementation of a ConTract System) soll die Implementierung und Ausführung von Abläufen aus Wirtschaft, Industrie, Dienstleistung oder anderen Bereichen untersucht werden. Das APRICOT–System wird zur Zeit am IPVR[1] der Universität Stuttgart entwickelt.

Es handelt sich hierbei um ein prototypisches Laufzeitsystem zur Abwicklung von ConTracts.

ConTracts sind langlebige, verteilte Anwendungen, die sich aus kurzen ACID-Transaktionen aufbauen und deren Ablauf durch einen Kontrollmechanismus oberhalb der Transaktionen gesteuert wird. ConTracts können migriert, angehalten und wieder fortgesetzt werden. Im Falle eines Fehlers wird eine Forward–Recovery ausgeführt. Die Synchronisation von ConTracts erfolgt transaktionsübergreifend mit Hilfe von Prädikaten, welche die von einer Anwendung benötigten Isolationsbedingungen explizit beschreiben.

Das Laufzeitsystem APRICOTS gewährleistet den robusten, kontrollierten und fehlertoleranten Ablauf von ConTracts: Ein ConTract wird unter allen Umständen zu Ende geführt. Es stellt Komponenten für die Betriebsmittelverwaltung und die Verwaltung der verteilten Transaktionen und Beziehungen zur Verfügung.

Diese Aspekte bleiben dem Anwendungsprogrammierer verborgen. Er kann auf der Anwendungsebene einen ConTract erstellen, ohne Rücksicht auf dessen Umsetzung in der prozeduralen Ebene nehmen zu müssen.

[1]Institut für Parallele und Verteilte Höchstleistungsrechner

1.2 Inhalt der Arbeit

Bei allen Arbeiten, die zur Erstellung dieses Systems beitrugen, wurde stets vorausgesetzt und / oder darauf hingearbeitet, daß irgendwann einmal ein Ablauf aus einem der oben genannten Bereiche als ConTract modelliert und tatsächlich ausgeführt wird.

Aber wie erstellt man nun einen ConTract, wie sieht ein Skript aus und wofür ist es, was sind Steps und was tun sie nun wirklich? Welche Vorarbeiten müssen definitiv ausgeführt werden, ehe man seinen eigenen ConTract starten kann?

Diese Arbeit befaßt sich mit diesen Fragen, indem zu einem Beispielablauf aus einem Fertigungsunternehmen ein Skript erstellt wurde. An ihm lassen sich die einzelnen Schritte nachvollziehen, die notwendig sind, um einen ConTract zu modellieren .

1.3 Kapitelübersicht

In *Kapitel 2* wird der gewählte Beispielablauf „Auftragsabwicklung" geschildert. Hier wird auf den Ablauf in der Realität eingegangen.

Im folgenden *Kapitel 3* wird das Skript behandelt. Die einzelnen Teile eines Skriptes werden vorgestellt: ihre Besonderheiten, Aufgaben, teilweise auch Einschränkungen.

Wie man Step Computation Server und Resource Manager zusammenstellt, wird in *Kapitel 4* erklärt.

Kapitel 5 erläutert die an der ConTract–Erstellung direkt beteiligten Komponenten.

Danach werden in *Kapitel 6* einige Fehlermöglichkeiten bei der ConTract-Abwicklung aufgezeigt, wie sie bisher behandelt werden und wie sie eventuell besser behandelt werden sollten.

Im vorletzten *Kapitel 7* sind die bei der ConTract–Erstellung aufgefallenen Schwachstellen aufgeführt. Teilweise ist ein Lösungsvorschlag angegeben.

Schließlich wird in *Kapitel 8* angesprochen, welche Punkte für die Zukunft offengelassen wurden, bzw. was weiteren Untersuchungen bedarf.

Kapitel 2

Eine reale Auftragsabwicklung

Im folgenden wird eine Auftragsabwicklung in einem Fertigungsunternehmen betrachtet. Sie umfaßt den gesamten Verlauf beginnend mit der Angebotsabgabe über die Fertigung eines Artikels bis hin zu dessen Auslieferung.

2.1 Der Ablauf einer normalen Auftragsabwicklung

Die Auftragsabwicklung läßt sich in mehrere Phasen einteilen:

In der ersten Phase wird dem Kunden eine Liste von Angeboten vorgelegt, unter denen er eines oder mehrere auswählt. Er gibt der Firma die eindeutige Kennung der von ihm gewünschten Angebote (Bestellnummer, Angebotsnummer, textuelle Angebotsbezeichnung oder ähnliches) an, welche er in Auftrag geben möchte. Zusätzlich gibt der Kunde weitere Informationen zu jedem Angebot an die Firma weiter: Er teilt der Firma die beabsichtigte Bestellmenge mit, erkundigt sich nach Rabatten.

Die Firma prüft nun, ob sie einen Auftrag mit diesen Vorgaben durchführen kann oder nicht. Beispielsweise könnte die Fertigungskapazität unter- oder überschritten, der gewünschte Fertigstellungstermin kann nicht eingehalten werden, usw. Die hier möglichen Fälle, die einen Konflikt beinhalten werden in Abschnitt 2.2 vertieft behandelt.

Schließlich kommt es im Normalfall zu einer Einigung zwischen Kunde und Firma, was durch einen schriftlichen Auftragsabschluß fixiert wird.

Hier beginnt die zweite Phase. Bei ihrer genaueren Erläuterung soll vorausgesetzt sein, daß nur ein Artikel bestellt wurde. Zudem soll sich dieser Artikel aus fremdgefertigten und selbstgefertigten Teilen zusammensetzen.

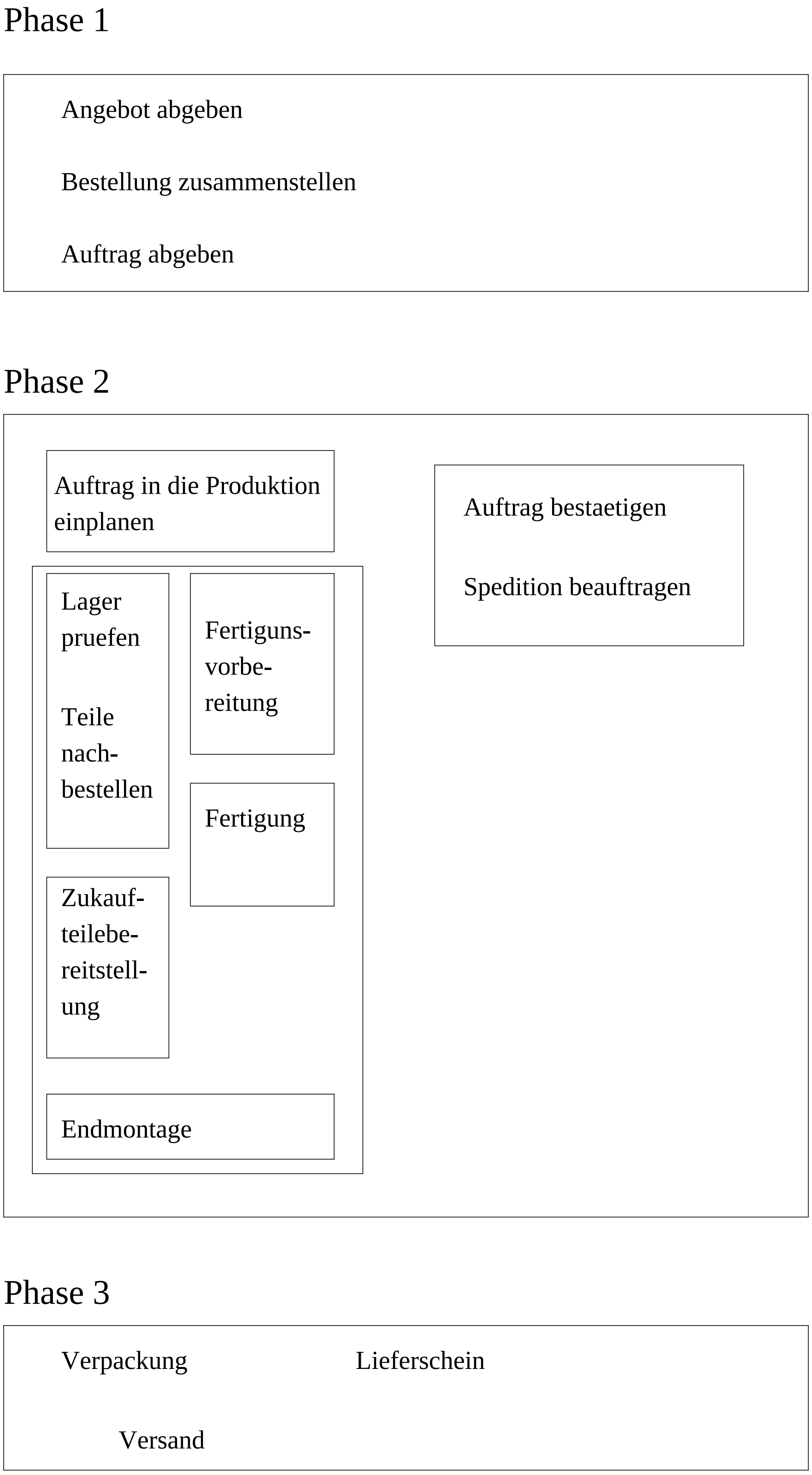

Abbildung 2.1: Die Phasen einer Auftragsabwicklung

In diesem Abschnitt einer Auftragsabwicklung konzentrieren sich die Aktivitäten auf die Seite der Firma. Sie sendet dem Kunden eine Auftragsbestätigung zu und beauftragt eine Lieferfirma für die spätere Auslieferung der bestellten Ware.

Des weiteren muß die Fertigung des Artikels in die laufende Produktion eingeplant werden, d. h. es wird eine eventuelle Nachbestellung von fremdgefertigten Teilen berücksichtigt, ebenso wie die Auslastung der Maschinen, die Für die Produktion des Artikels benötigt werden.

Vor der eigentlichen Fertigung werden die Maschinen überprüft und mit dem notwendigen Werkzeug bestückt. Die Fertigung selbst kann sich aus mehreren kleinen Schritten zusammensetzen, wie z. B. Werkstück drehen, fräsen und anschließend bohren.

Während dieser Zeit werden die noch zusätzlich benötigten fremdgefertigten Teile aus dem Lager geholt und für die Endmontage bereitgestellt, womit diese gleich an das Ende der Bearbeitung des selbstgefertigten Teils angeschlossen werden kann.

In der dritten Phase finden nun die Verpackung des Produktes, die Ausstellung des Lieferscheins und die letztliche Auslieferung des Produkts mittels der beauftragten Lieferfirma statt.

2.2 Ereignisse während einer Auftragsabwicklung

Die zuvor aufgeführte Auftragsabwicklung wird in diesem Abschnitt noch einmal betrachtet, jetzt jedoch werden viele mögliche Situationen durchlaufen. Alle nur irgendwie denkbaren Situationen können aufgrund ihrer Vielzahl nicht aufgeführt werden, es werden deshalb nur diejenigen berücksichtigt, die zu „üblichen Ausnahmefällen“ gerechnet werden können.

Der Kunde bekommt also eine Liste von Angeboten vorgelegt. Ihn interessiert vorläufig keines dieser Angebote, womit die Sache erledigt wäre.

Später fällt ihm ein, daß er doch einen Artikel gut gebrauchen könnte und so beginnt die Auswahl aus dem Sortiment der Angebote noch einmal. Schließlich entscheidet sich der Kunde beispielsweise für zwei Angebote. Eine Bestellung verknüpft er jedoch mit der Bedingung, daß zum einen der Preis für ihn günstiger ist als der im Angebot angegebene, und daß er zum anderen die Artikel spätestens zu einem bestimmten Termin erhält.

Es kann sein, daß die Firma nicht allen Forderungen des Kunden nachkommen kann oder möchte, und somit wäre nun zu diesem Zeitpunkt die „Auftragsabwicklung“ zu Ende. Dies tritt ein, falls die Firma nicht bereit ist, dem Kunden im Preis entgegen zu kommen, weil sie keinen Gewinn erwirtschaften würde oder sogar Unkosten hätte.

Ein weiterer Fall wäre, daß die Firma nicht in der Lage ist, das gewünschte Produkt aufgrund ihrer hohen Maschinenauslastung rechtzeitig fertig zu bekommen. Terminschwierigkeiten können auch dann auftreten, wenn die Firma selbst für die Artikelherstellung erst noch fremdgefertigte Teile bestellen müßte, weil sie von diesen nicht ausreichend viele am Lager hat. Hier können die Probleme darin bestehen, daß diese Teile relativ schwer zu erhalten sind, oder eine so lange Lieferdauer besteht, daß es nicht mehr für eine fristgerechte Herstellung der vom Kunden gewünschten Artikel reicht.

Ist auch dieser Konflikt zwischen Kundenwünschen und Firmeninteressen überwunden und es zu einem Kompromiß zwischen Kunde und Firma gekommen, wird ein Vertrag abgeschlossen.

Die Firma wird für den Kunden eine Bestätigung des Vertrags ausstellen und sie ihm zusenden. Außerdem wird mit Hilfe des Produktions–Planungs–Systems (PPS) der Auftrag in die laufende Produktion eingegliedert und es wird auch eine Spedition mit der späteren Auslieferung der bestellten Ware beauftragt.

Hier ist nun denkbar, daß der Kunde inzwischen doch noch einen anderen Anbieter entdeckt hat, der alle seine ursprünglichen Forderungen akzeptiert. Er tritt also an die Firma heran und versucht entweder, die Firma dazu zu bewegen, ihm doch weiter als vereinbart entgegenzukommen, oder aber er storniert seinen Auftrag.

Für den ersten Fall wird der Auftrag korrigiert, die Firma führt eine neue Produktionsplanung durch und disponiert bei der eventuell schon beauftragten Lieferfirma den Auslieferungstermin um.

Im zweiten Fall muß die Firma den Auftrag des Kunden stornieren und auch gegebenenfalls ihren eigenen Auftrag an die Lieferfirma. Weiterhin wird der Auftrag durch das Produktions–Planungs–System wieder ausgeplant.

Ansonsten beginnt jetzt die Fertigung. Für den hier geschilderten Ablauf soll zur Fertigung auch eine Bestellung von fremdgefertigten Teilen gehören, die notwendig wird, weil nicht genügend der erforderlichen Teile am Lager sind. Sollten die Rohteile, die von der Firma selbst erst noch bearbeitet werden nicht ausreichend vorhanden sein, müssen sie natürlich auch beschafft werden.

An sich wird die Zeit bis zum Eintreffen der nachbestellten Teile schon bei der Erstellung des Produktionsplanes durch das PPS berücksichtigt. Interessant wird hier jedoch der Aspekt, wenn sich zeitliche Verzögerungen bei der Auslieferung dieser Teile an die Firma ergeben.

Diese haben auf jeden Fall zur Folge, daß das PPS eine erneute Produktionsplanung durchführen muß. Eine weitere Auswirkung kann sich auf den Liefertermin des Artikels ergeben, wenn keine Pufferzeiten vorhanden sind oder die eingeplanten nicht ausreichen.

Bei der Bearbeitung des von der Firma selbst zu fertigenden Teils ist natürlich auch eine Vielzahl von Komplikationen denkbar.
Begonnen werden kann hier mit dem Ausfall einer Werkzeugmaschine. Sie muß repariert werden und vor allen Dingen muß das PPS für eine Umverteilung der auf dieser

Maschine zur Bearbeitung vorgesehenen Teile sorgen.
Weiter kann passieren, daß während der Bearbeitung des selbst zu fertigenden Teils ein Fehlschlag auftritt. Ein weniger tragisches Ereignis ist wohl, daß die Maschine zu heiß wird, was außer einer gewissen Wartezeit bis die Maschine wieder abgekühlt ist weiter keine Auswirkungen hat.

Ist es jedoch so, daß entweder das Werkstück selbst (also das zu bearbeitende Teil) oder aber ein Werkzeug kaputt geht (z. B. der Bohrer an der Werkzeugmaschine Bohrmaschine), wird die Sache etwas komplizierter, da hier verschiedene Aktionen notwendig sind. Zuerst muß der Fehler analysiert, dann der momentane Zustand von Maschine und Werkstück ermittelt werden, um schließlich die entsprechende Maßnahme durchführen zu können.

Hier vorkommende Möglichkeiten sind unter anderem:

- Strom, NC–Programm oder ähnliches war ausgefallen, weder am Werkstück noch an der Maschine ist etwas kaputt; nachdem der Fehler behoben ist, kann normal weitergemacht werden

- Die Maschine hat ein Fehlverhalten gezeigt (z. B. das Werkstück wird nicht ordentlich geschliffen). Hier kann das Werkstück möglicherweise durch eine nochmalige Ausführung des gleichen Bearbeitungsvorgangs doch noch „gerettet“ werden, ansonsten gehört es wohl zum Ausschuß. Selbstverständlich ist es hier notwendig, die genaue Fehlerquelle zu finden, falls dies nicht ein einmaliges Vorkommnis war.

- Das Werkstück ist bei seiner Bearbeitung kaputt gegangen, es muß weggeworfen werden

- Das Werkzeug ist beschädigt und muß gewechselt werden. Hier ist zu prüfen, inwieweit das defekt gewordene Werkzeug einen Schaden am Werkstück verursacht hat.

Sollte während der Fertigung ein Fehler auftreten, kann dies zu einer Verschiebung des Fertigstellungstermins führen, eventuell eine Nachbestellung von Rohteilen zur Folge haben, etc. Die Gründe dafür sind, daß unter Umständen zusätzliche Aktionen notwendig sind oder bereits durchgeführte Schritte wiederholt werden müssen.

Wurde die Fertigung der bestellten Artikel erfolgreich durchlaufen, muß der letztlich entstehende Artikel aus den nun bearbeiteten und fremdgefertigten Teilen montiert werden. Geht jetzt etwas kaputt, muß eventuell noch einmal die Fertigung durchlaufen werden, bzw. eine Nachbestellung von Zukaufteilen[1] erfolgen.

Schließlich werden die Produkte verpackt und der Lieferschein ausgestellt.

[1] Zukaufteile entsprechen fremdgefertigten Teilen

Auch bei der Verpackung kann etwas beschädigt werden, was dann die gleichen Auswirkungen wie eine fehlerhafte Endmontage hätte. Eventuell ist auch nur nicht genügend Verpackungsmaterial vorhanden.

Natürlich muß berücksichtigt werden, daß die ganze Zeit über seit dem Beginn der Fertigung die Möglichkeit besteht, daß der Kunde seinen Auftrag storniert oder auch nur einen der angeforderten Artikel abbestellt. Die schon gefertigten Teile müssen dann ins Lager gebracht werden. Ist das Produkt gar schon montiert, könnte vor einer Einlagerung noch eine Demontage erfolgen (sofern dies möglich, sinnvoll oder notwendig ist).

Bei der Auslieferung können natürlich auch noch unerfreuliche Situationen auftreten: das beauftragte Speditionsunternehmen beschädigt die Fracht, verzögert die Zustellung der Artikel, etc.

Zum Schluß soll der Vollständigkeit halber noch erwähnt werden, daß mit der Auslieferung noch kein Abschluß der Auftragsabwicklung gegeben sein muß. Oft besteht noch eine Garantiefrist auf das Produkt oder gar ein langfristig wirkender Wartungsvertrag, so daß hier durchaus noch einige Ereignisse vorkommen können. Diese Punkte werden allerdings im Skript nicht berücksichtigt.

Kapitel 3

Der ConTract „Auftragsabwicklung“

Eine solche im vorigen Kapitel geschilderte Auftragsabwicklung soll nun als ConTract ablaufen. Bis dies tatsächlich möglich ist, müssen einige Vorbereitungen getätigt werden.

Man benötigt ein Skript, das den formalen Ablauf beschreibt. Dann muß man die im Skript aufgeführten Steps kodieren (ebenso die von Steps eventuell benötigten Services).

Nachdem dies alles erledigt ist, sind diese Steps und Services in Step Computation Servern und Resource Managern zusammenzustellen, welche wiederum zu Komponenten zusammengefaßt werden. Die Komponenten laufen schließlich als Prozesse im Rechnernetz.

In den folgenden Abschnitten werden die einzelnen Bestandteile der Vorgehensweise zur Scripterstellung genauer erläutert, oft anhand eines Beispiels aus dem Skript „Auftragsabwicklung“. Das Skript selbst ist in kodierter Form im Anhang zu finden.

Auf die Erstellung der Step Computation Server und Resource Manager, sowie der Komponenten wird im Abschnitt 4 eingegangen.

3.1 Das Skript

Das Skript baut sich aus einer Sammlung von Steps auf. Mittels einer speziellen Skriptsprache (siehe hierzu [Masman91]) werden sie durch einen explizit anzugebenden Kontrollfluß geordnet. Weiterhin werden für den Fall des Scheiterns einer Entry_Invariante Konfliktbehandlungsstrategien angegeben, Steps für den Fall einer Kompensation und natürlich die Kontextelemente für die jeweiligen Steps.

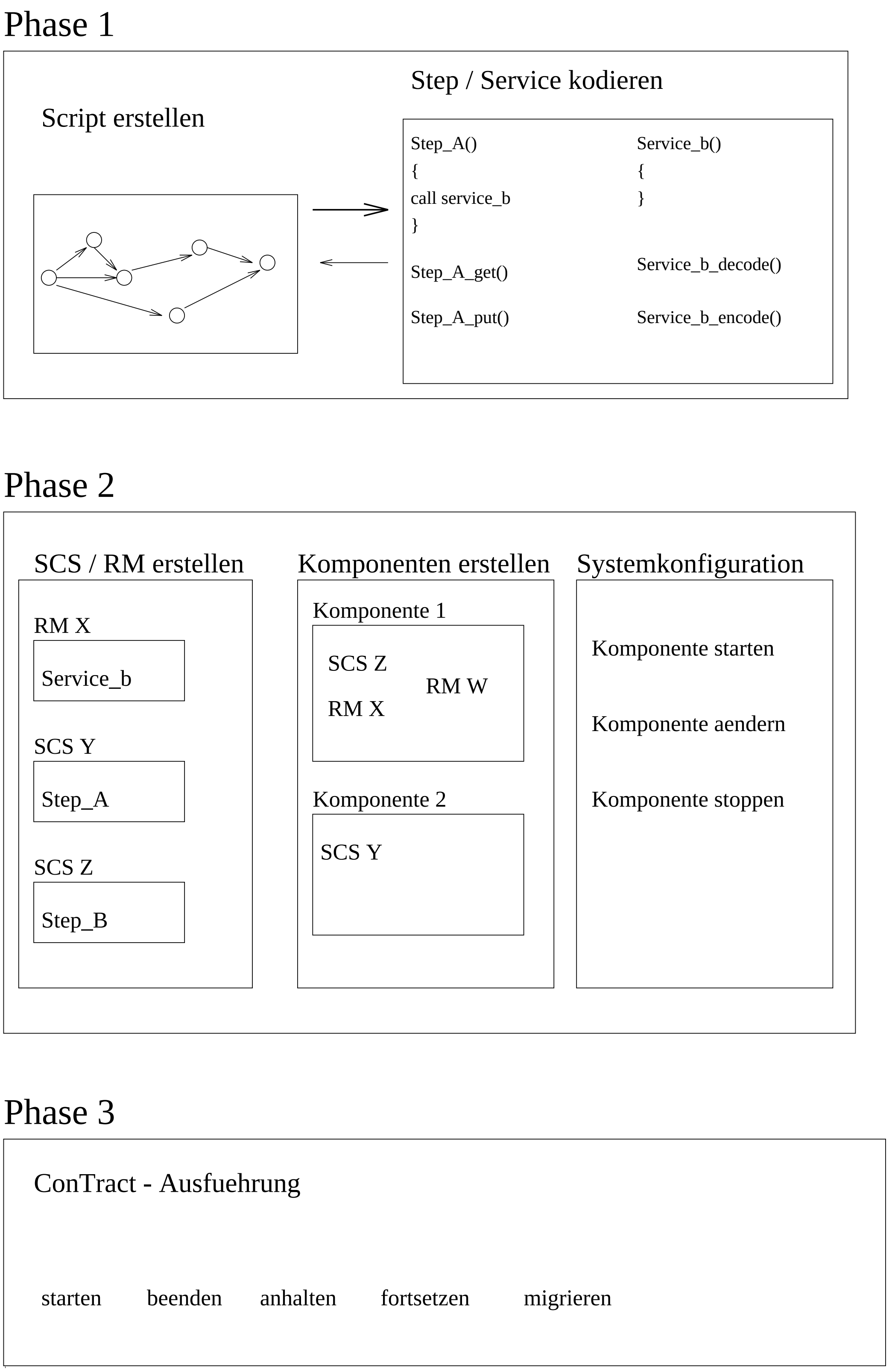

Abbildung 3.1: Phasen einer ConTract-Erstellung

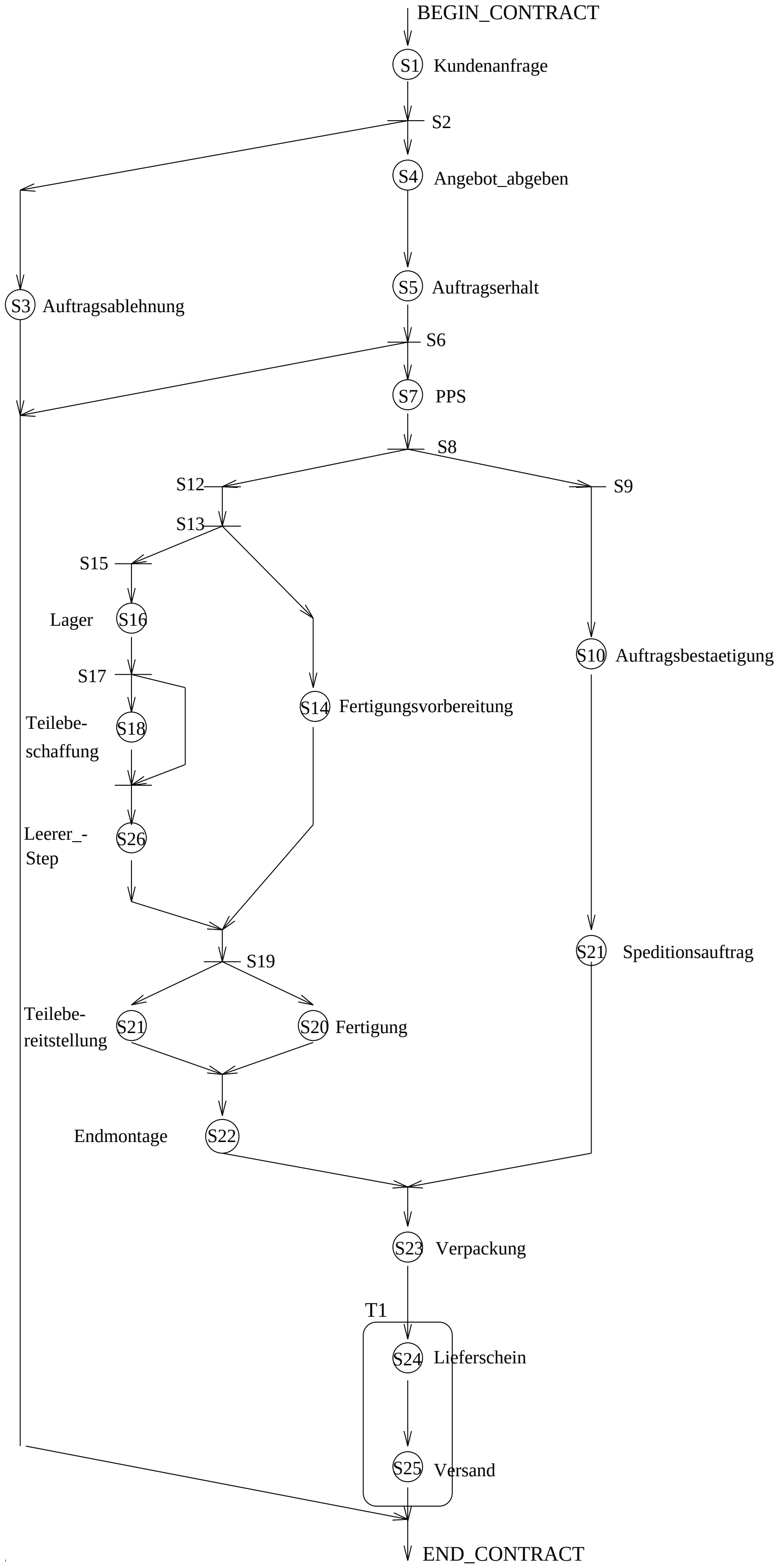

Abbildung 3.2: Das Script „Auftragsabwicklung“ in graphischer Darstellung

CONTRACT <*ConTract_Name*>

BEGIN_CONTEXT_DECL
    Liste aller Kontextelemente
END_CONTEXT_DECL

BEGIN_SCRIPT
    Steps mit Kontrollflußanweisungen
END_SCRIPT

BEGIN_CONFLICT_RESOLUTION
    für alle Steps mit einer Entry_Invariante hier angeben, was im Falle eines Konfliktes getan werden soll: jeweils einen oder mehrere Konfliktlösungs–Steps
END_CONFLICT_RESOLUTION

BEGIN_TRANSACTIONS
    hier Transaktionen angeben, falls im Skript welche vorkommen sollen und eventuell DEPENDENCY's
END_TRANSACTIONS

BEGIN_COMPENSATIONS
    zu jedem Step einen Compensation–Step angeben
END_COMPENSATIONS

Abbildung 3.3: Schematischer Aufbau eines Skripts

3.2 Kontrollflußdeklarationen

Wesentliches Element für die Kontrollflußdeklaration ist die Sequenz, die normalerweise implizit vorgegeben ist. Sie muß dann explizit angegeben werden, wenn „geschachtelt“ wird. Vergleiche dazu im Skript „Auftragsabwicklung“ Step `S9` und `S12` (*SEQ-Anweisung*).

Weiterhin besteht durch das *IF-Element* die Möglichkeit, aufgrund eines während des ConTract–Ablaufs ermittelten Wertes in einen von zwei möglichen Pfaden zu verzweigen. Im vorgegebenen Skript (`S17`): Falls nicht genügend Teile im Lager sind, müssen diese nachbestellt werden, andernfalls muß nichts getan werden. Dieser ermittelte Wert muß ein Kontextelement sein.

Mit der *PARBEGIN* und *PAREND–Anweisung* wird angegeben, welche Steps oder auch ganze Abschnitte parallel ausgeführt werden können. Werden die durch *PARBEGIN* und *PAREND* geklammerten Teile doch sequentiell ausgeführt, ist ihre Ausführungsreihenfolge ohne Belang. Beispiele im Skript „Augftragsabwicklung“sind:

- **S19**: `S20: Fertigung` und `S21: Teilebereitstellung` können parallel zueinander ausgeführt werden; laufen sie dennoch (aus welchen Gründen auch immer) sequentiell ab, ist es egal, welcher Step zuerst ausgeführt wird.

- **S8**: dies ist ein Beispiel dafür, daß es möglich ist, auch komplexere Teile parallel zueinander auszuführen. Hier baut sich jeder Teil aus einer beliebigen Anordnung von Steps auf.

Es sind noch weitere, jedoch im Skript „Auftragsabwicklung“ nicht verwendete Konstrukte möglich. Hierzu siehe [Masman91] .

3.2.1 Invarianten

Zur Synchronisation über Step- (bzw. Transaktions-) Grenzen hinweg werden im Skript Invarianten verwendet. Diese dienen dazu, einen bestimmten Zustand zu sichern, auf den man sich zu einem späteren Zeitpunkt beziehen kann. Eine Entry_-Invariante muß sich also genau auf die vorangegangene Exit_Invariante beziehen. Sollte der so vorgegebene Zustand nicht mehr gegeben, also ein Konflikt entstanden sein, muß dieser mit Hilfe von *CONFLICT_RESOLUTION-Steps* gelöst werden. *CONFLICT_RESOLUTION-Steps* sind zu jedem Step, der eine Entry–Invariante besitzt, anzugeben.

Es gibt verschiedene Strategien, die den Zustand, den die Exit_Invariante vorgegeben hat, für die Entry_Invariante sichern [WäReu91]:

- POLICY := `locking`
  Hier wird der von der Exit_Invariante gelesene oder geschriebene Wert so erhalten wie er ist, also eine Sperre auf den Wert gehalten.

- POLICY := `escrow`
  Der gelesene oder geschriebene Wert darf sich innerhalb gewisser Schranken ändern.

- POLICY := `check_revalidate`
  Der Wert darf sich beliebig ändern, zu einem bestimmten Zeitpunkt (spätestens wenn die zugehörige Entry_Invariante ausgeführt wird) wird geprüft, ob der spezifizierte Wert wieder gegeben ist.

Im Skript „Auftragsabwicklung“ soll mit der Exit_Invariante von Step `S26: Leerer-_Step` sichergestellt werden, daß genügend Teile am Lager sind, sodaß sich die Fertigung (Step `S20: Fertigung`) auf diesen Zustand verlassen kann. Dieser Step bezieht sich mittels seiner Entry_Invariante auf den zuvor hergestellten Zustand.

Sollte dieser nicht mehr gegeben sein, weil inzwischen Teile aus dem Lager genommen wurden, werden die angegebenen *CONFLICT_RESOLUTION-Steps* ausgeführt. Im vorliegenden Skript ist es nur ein Step, nämlich `S120: Teilenachbestellung`, der das Lager wieder mit Rohteilen auffüllt.

In diesen Skript mußte bezüglich der Exit_Invariante ein „`Leerer_Step`“ (Step `S26`) eingeführt werden, da der Zustand des Lagers auf jeden Fall gesichert werden soll, egal welcher Zweig nach der *IF*-Abfrage (`S17`) durchlaufen wird. Diese Abfrage entspricht einem Pseudo-Step und da man für Pseudo-Steps keine Invarianten angeben kann, muß man hier auf diese Weise ein bischen „tricksen“ .

Hier noch eine Bemerkung:
Der *CONFLICT_RESOLUTION-Step* `S120: Teilenachbestellung` wird nur dann ausgeführt, wenn die Entry_Invariante von Step `S20: Fertigung` fehlschlägt, also nicht genügend Teile für die Fertigung vorhanden sind. Dieser Step sorgt dann dafür, daß wieder genügend Rohteile im Lager sind, also speziell die Teile besorgt werden, die für die Fertigung benötigt werden.

Hier wäre es ebensogut möglich gewesen, statt dieses speziellen Steps `S120: Teilenachbestellung` den schon im normalen Skriptablauf vorkommenden Step `S18: Teilebeschaffung` zu verwenden.

Der Vorteil einer Wiederverwendung dieses Steps im Skript wäre gewesen, daß insgesamt ein Step weniger hätte programmiert werden müssen.

Der Nachteil bei der Wiederverwendung von Step `S18: Teilebeschaffung` ergibt sich jedoch darin, daß nicht nur eine Nachbestellung der speziell in der Fertigung benötigten Rohteile durchführt wird, sondern auch für die später in der Endmontage benötigten Zukaufteile. Dieser zusätzliche Teil ist aber für die Konfliktlösung zum einen nicht notwendig sondern geradezu überflüssig, und ergibt somit zum anderen einen hohen Aufwand.

Hier noch eine weitere Bemerkung zur Formulierung der Entry_Invariante:
In der Realisation von Step `S20: Fertigung` werden die Services *drehen, fräsen* und *bohren* aufgerufen, wobei jeweils die Zahl der noch zu bearbeitenden Teile berücksichtigt wird. Vor dem Beginn der Fertigung interessiert also nicht nur die Zahl der wirklichen Rohteile, sondern auch, wieviele Teile schon „halbfertig“ im Lager sind. Aus diesem Grund ergibt sich für die Invariante eine Berechnung der Summe aller fertigen, halbfertigen und Rohteile, deren Ergebnis größer sein muß als die Anzahl der tatsächlich zu fertigenden Artikel, damit es nicht zu einer Konfliktsituation kommt.

3.2.2 Kontextelemente

Jeder Step hat einen speziellen, anwendungsspezifischen inhaltlichen Ablauf. Damit bei der Abarbeitung eines ConTracts ein Step auf der gleichen Datenbasis arbeiten kann, den ein anderer vorgegeben hat, müssen die entsprechenden Werte, die die Steps benötigen, sicher zwischengespeichert werden. Dies geschieht über die Kontextelemente.

Unmittelbar vor dem eigentlichen Aufruf eines Steps, also dem Funktionsaufruf innerhalb eines Step Computation Servers, werden die Kontextelemente, die der jeweilige

Step für seine Abarbeitung benötigt, aus dem Kontext geholt und dem Step als Parameter übergeben.

Dann wird der Step-Code abgearbeitet und eventuell Werte (neue und / oder veränderte) zurückgegeben, die jetzt im Kontext abgelegt werden.

Diese Kontextelemente müssen im Skript angegeben und später bei der Programmierung der Steps entsprechend berücksichtigt und korrekt verwendet werden.

Umgekehrt gilt dies auch bei der Erstellung eines Skripts, falls für dieses Skript auf bereits programmierte Steps zurückgegriffen wird.

Kontextelemente, die ein Step aus dem schon vorliegenden Kontext benötigt, werden als *IN-Parameter* im Skript aufgeführt, die vom Step produzierten als *OUT-Parameter*. Verändert der Step den Wert eines Kontextelements, so ist dieses sowohl als *IN*–, als auch als *OUT–Parameter* anzugeben oder eleganter mittels *IN_OUT*.

Die möglichen Typen für die Kontextelemente sind LONG, DOUBLE, FLOAT, STRING und MCURSOR (vgl. auch [Masman91] und [BuRa90]).

Soll ein Step mit einem von einem speziellen Vorgängerstep erzeugten Kontextwert arbeiten und dieser Wert in der Zwischenzeit von anderen Steps verändert worden sein, gibt es die Möglichkeit, durch die Angabe des entsprechenden Erzeuger-Steps genau den benötigten Kontextelementwert zu erhalten. Im Skript müßte dazu beim *IN–Parameter* „*BY Sx*“ angegeben werden, wobei x für die Nummer des gewünschten Erzeuger-Steps steht.

Es gibt noch gewisse Einschränkungen bei der Angabe von Kontextelementen im Skript:

1. Der erste Step hat nur *OUT–Parameter*.

2. Jeder *IN–Parameter* muß in irgendeinem im Ablauf vorher auftretenden Step als *OUT–Parameter* vorhanden sein.
   Es ist also eine in Abbildung 3.4 dargestellte Kontextelementdeklaration nicht erlaubt.
   Wird das Kontextelement `artikel` nur von Sl erzeugt, hier aber der Zweig mit Sm durchlaufen, existiert das Kontextelement *artikel* nicht, auf welches Sn aufbaut.

3.2.3 Transaktionen

Die Abhängigkeiten der Steps untereinander können durch die Angabe von Transaktionen festgelegt werden: wenn der Lieferschein nicht ausgestellt wurde, soll auch kein Versand erfolgen, ebenso soll der Kunde keinen Lieferschein erhalten, falls der Versand nicht klappt und somit die bestellten Artikel nicht bei ihm ankommen.

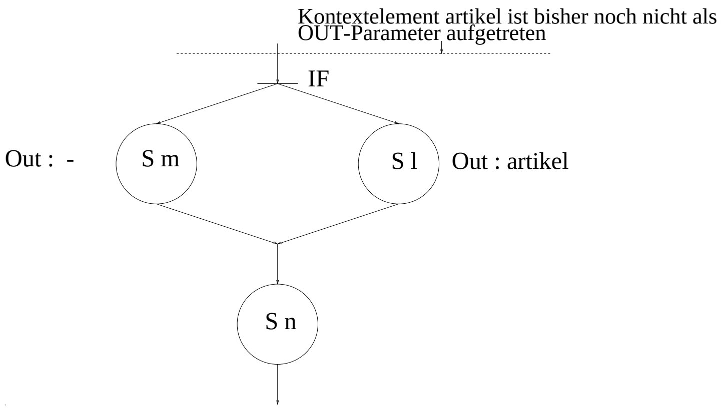

Abbildung 3.4: Nicht erlaubte Deklaration des Kontextelements `artikel`

In einer Transaktion können Steps zusammengefaßt werden. Dies geschieht durch die Angabe `T`x`(S`a`, S`b`, ...)`. Bricht einer der in der Transaktion angegebenen Steps ab, werden alle weiteren in der Transaktion angegebenen und schon ausgeführten Steps ebenfalls abgebrochen, d. h. auf diese wird ein *ABORT* ausgeführt. Dann wird die Transaktion nochmals gestartet. Die Zahl der Wiederholungen entscheidet der ConTract–Manager ([Wahl91]; dies gilt im übrigen auch für die Wiederholungen von abgebrochenen Steps).

Diese Strategie, bei der die abgebrochene Transaktion nochmals durchgeführt wird, ist implizit vorgegeben. Es sind aber auch andere Strategien möglich. Sie müssen explizit angegeben werden. Um dies zu veranschaulichen, soll für das Skript „Auftragsabwicklung“ eine zusätzliche Transaktion eingeführt werden:

```
T2(S19, S20, S21, S22)
```

Als Strategie ist jetzt möglich:

```
DEPENDENCY (T2: a|->b :S13)
```

Bedeutung: Geht bei der Fertigung oder der Endmontage etwas schief, wird nochmals mit Step `S13` begonnen, also erst der Lagerbestand nochmals überprüft und die Fertigungsvorbereitung durchgeführt, ehe nochmals gefertigt und das Endprodukt montiert wird.

Eine weitere Strategie wäre:

`DEPENDENCY (T`x`: a|->b :T`y`)`

`T`x und `T`y stehen für zuvor deklarierte Transaktionen. Hier wird beim Abbruch von `T`x Transaktion `T`y begonnen.

3.2.4 Kompensations–Steps

Diese Steps werden nur benötigt, wenn der Benutzer des ConTracts dem ConTract–Manager die Anweisung gibt, diesen ConTract zu kompensieren. Sie werden keines-falls dann ausgeführt, wenn der Step aus irgendeinem Grund abbricht. Die vom Step hinterlassenen Effekte werden durch ein *ABORT* zurückgesetzt, sofern es sich um Datenbank-Änderungen handelt, für reale Aktionen muß eine Recovery durchgeführt werden (durch z. B. einen Recovery Manager; vgl. hierzu Abschnitt 7.2).

Mit Hilfe der Kompensations–Steps werden die von den Steps vorgenommenen Änderungen rückgängig gemacht. Hier kann keine Kompensation im Sinne eines *ABORT* durchgeführt werden, sondern es muß eine logische Kompensation erfolgen.

Dies möchte ich am Beispiel der Fertigung genauer erläutern:
Bearbeitete Teile können nicht in den ursprünglichen Zustand zurückversetzt werden. Hier besteht die Kompensation darin, die gefertigten Teile ins Lager zu bringen.

3.2.5 Der Skript-Compiler

Hat man ein Skript fertig erstellt, ist es mit dem Skript-Compiler ([Masman91]) zu übersetzen. Hier werden die Kontrollflußangaben, die Invarianten und anderes mehr so umgesetzt, daß sie der ConTract–Manager später bei der Ausführung eines ConTracts verwenden kann.

Der Aufruf des Skript-Compilers erfolgt mit

`actcp` <*filename*>

wobei im File *filename* der Scriptcode steht.

3.3 Step

Die Steps bilden jeweils für sich Einheiten, die einen Programmschritt der speziellen Anwendung realisieren. Als Einheit kann zum Beispiel das Auflisten aller Angebote, die Auswahl eines dieser Angebote durch den Benutzer, dessen weitere Wünsche bei der potentiellen Bestellung und die Überprüfung durch die Firma, ob dieses Angebot letztlich so abgegeben werden kann, angesehen werden. Diese Einheit wird dann durch einen Step realisiert (Im Script „Auftragsabwicklung“ durch Step `S1: Kundenanfrage`). Als besonderes Beispiel ist der Step `S20: Fertigung` zu nennen, der auf anwendungsunabhängige Dienste wie *Teile drehen – fräsen – bohren* zurückgreift, um die Fertigung zu realisieren.

Sollen Abhäniggkeiten unter den Steps entstehen, die selbst ja unabhängige Einheiten bilden, sind diese Abhängigkeiten durch die Kontrollflußkonstrukte im Skript festzulegen. Diese steuern dann den Ablauf: *SEQ, PAR, IF, WHILE, PARFOREACH*, etc. (vgl. Abschnitt 3.2). Eine weitere Möglichkeit zur Bildung von Beziehungen sind die in Abschnitt 3.2.3 schon angesprochenen Transaktionen.

Im Skript werden zu jedem Step Kontextelemente angegeben. Da ein Step jeweils nur einem Teilschritt der gesamten Anwendung entspricht, kann sich sein Inhalt auf

Werten aufbauen, die von anderen Steps erzeugt wurden. Diese Werte müssen wie schon angesprochen als Kontextelemente im Kontext vorhanden sein und von dort geholt werden, damit ein Step mit ihnen arbeiten kann. Bei der Programmierung eines Steps ist es deshalb wichtig, diese „Kontextelemente“ richtig zu benutzen. Man muß also den jeweiligen Zweck der Kontextelemente und auch ihren Typ kennen. Der Wert, den das Kontextelement jeweils wirklich hat, darf zwar als eingeschränkt vorausgesetzt werden, sollte aber im Step selbst trotzdem nochmals überprüft werden.

Jeder Step entspricht einer (C-)Funktion. Über seine Schnittstelle bekommt er die benötigten Kontextelementwerte und gibt auch die von ihm erzeugten und / oder veränderten Kontextelementwerte zurück. Aus dem Kontext geholt werden die vom Step benötigten Werte unmittelbar vor dem Aufruf des Step-Codes, dort abgelegt werden die zurückgegebenen Werte entsprechend sofort nach der Beendigung der eigentlichen Step-Codes.
Hierzu werden für jeden Step zwei Funktionen zur Verfügung gestellt:
`<stepname>_get()` für das Holen der Kontextelementwerte
`<stepname>_put()` für das Speichern der Kontextelementwerte im Kontext

Die inhaltlichen Aufgaben eines Steps werden im Wesentlichen durch Datenbankzugriffe, Service-Aufrufe und Interaktionen mit dem Benutzer, der den ConTract gestartet hat, realisiert.

Auf die Service-Aufrufe wird in Abschnitt 5.1 näher eingegangen.

Die Interaktionen mit dem Benutzer beschränken sich vorläufig noch auf die Bildschirmausgabe und die Eingabe von Werten über die Tastatur.

3.4 Service

Ein Service ist ebenfalls eine (C-)Funktion. Ein Service kann synchron oder asynchron aufgerufen werden. Sinn eines Service ist, einen allgemeinen, d. h. einen unabhängig verwendbaren Dienst zur Verfügung zu stellen. Ein Service dient zur Verwaltung einer gemeinsam genutzten Resource oder der Ausführung einer bestimmten Aktion. Jeder Service kann selbst wieder einen Service (auch Subservice genannt) aufrufen. Siehe hierzu Abbildung 3.5.

Für die Durchführung eines Service kann es notwendig sein, dem Service Vorgaben zu liefern. Beispielsweise benötigt der Dienst *werkstueck_bohren* die eindeutige Kennung des zu bohrenden Werkstücks und eine Angabe, wieviele Werkstücke zu bearbeiten sind. Diese Information muß dem Service beim Aufruf über das Kommunikationssystem mitgegeben werden. Da in der vorliegenden Prototypversion nur ein „verschicken“ von ASCII-Strings möglich ist, müssen die mitgegebenen Werte vor dem Funktionsaufruf im Resource Manager entpackt, also in die einzelnen Bestandteile zerlegt werden. Umgekehrt wird nach Beendigung des Service das korrespondierende Verfahren angewendet: Eventuelle Rückgabewerte werden „verpackt“ . Für jeden

Ein Script enthält die Steps A, B und C.

Step / Service	ruft auf Service(s)
A	a
B	a, b, c
C	b
a	—
b	d
c	a
d	—

Bezogen auf das Script „Auftragsabwicklung“ :
Step Fertigung ($\hat{=}$ Funktion *Fertigung()*) ruft die Services *werkstueck_drehen*, *werkstueck_fraesen* und *werkstueck_bohren* auf.

Abbildung 3.5: Beispiel für Service–Aufrufe

Service sind deshalb zwei zusätzliche Funktionen zur Verfügung zu stellen:

<service_name>_decode() auspacken der Parameter
<service_name>_encode() einpacken der Rückgabewerte

Kapitel 4

Das APRICOTS User Interface

Wird ein ConTract, d. h. die Abarbeitung eines Skriptes durch den ConTract–Manager gestartet, müssen Server vorhanden sein, welche die benötigten Steps und Services an ihrer Schnittstelle anbieten.
Hierfür werden wahlweise eine oder mehrere APRICOTS-Komponenten definiert, die diesen Code letztlich enthalten.

Das APRICOTS User Interface hilft dem Benutzer dabei, auf einfache Weise aus den programmierten Steps und Services Step Computation Server und Resource Manager zusammenzustellen und aus diesen ablauffähige APRICOTS-Komponenten[1] zu bilden.

Im APRICOT–System gibt es zwei Arten von Servern: Step Computation Server (SCS) und Resource Manager (RM).
Ein Step Computation Server beinhaltet den Code von genau einem Step, ein Resource Manager den Code eines Services.

Dabei muß der Code aller im Skript aufgeführter Steps in jeweils mindestens einem Step Computation Server und die von den Steps benötigten Services in jeweils mindestens einem Resource Manager vorhanden sein.

Aus diesen Step Computation Servern und Resource Managern sind dann APRICOTS-Komponenten zusammenzustellen.

Eine APRICOTS-Komponente kann aus mehreren Servern bestehen:

- einem oder mehreren Step Computation Servern
- einem oder mehreren Resource Managern
- jeweils einem oder mehreren Step Computation Servern und Resource Managern

[1] APRICOTS-Komponenten beinhalten stets einen Teil des Kommunikationssystems. Hierzu gehören die Step Computation Server und Resource Manager. Im Gegensatz dazu werden z. B. der ConTract–Manager und der Transaction–Manager unter dem Begriff „Komponente“ behandelt.

Die APRICOTS-Komponenten laufen später auf Rechnerknoten als Prozesse, wobei auf einem Knoten mehrere APRICOTS-Komponenten gestartet werden können.

4.1 Vorgehensweise zur Erstellung der APRICOTS-Komponenten

Zur Erstellung der APRICOTS-Komponenten ist es notwendig, nach der „Bottom-up-Methode“ vorzugehen. Zuerst legt man die statische Konfiguration fest, dann die dynamische Konfiguration.

Für die statische Konfiguration beginnt man mit der Definition der Steps und Services. Man gibt den Pfad an, wo das File mit dem (C-)Code zu finden ist, dann gibt man an, wie der Service-Namen (Dienstname) heißen soll. Zusätzlich muß man angeben, wie die Funktion selbst aufzurufen ist. Diese Angabe wird vom APRICOTS User Interface in den Step Computation Server, bzw. den Resource Manager eingebaut. Der Dienstname der Services entspricht dem in den Steps und Services beim Aufruf von „`RM_call_sync()`“ (genaueres siehe Abschnitt 5.1) angegebenen, der Dienstname der Steps dem Stepnamen selbst. Anmerkung: das APRICOTS User Interface kennt keinen Unterschied zwischen Step und Service. Ihre Definition ist stets eine „Servicedefinition“.
Man verfährt für alle Steps und Services auf die beschriebene Weise.

Hat man alle Steps und Services definiert, kann man die Definition der Step Computation Server und Resource Manager angehen. Hier wird für jeden zu definierenden Step Computation Server, bzw. Resource Manager die Liste aller im System verfügbaren Services angezeigt (ein Teil davon wurde ursprünglich als Step programmiert), aus denen man sich den gewünschten auswählt. (Deshalb auch „Bottom-up-Methode“; solange ein Service (Step) nicht definiert ist, kann man auch keinen Step Computation Server bzw. Resource Manager definieren, der diesen Service (Step) enthält.) Nur hier wird intern eine Unterscheidung zwischen Step und Service vorgenommen: für die Erstellung eines Step Computation Servers werden die Files `<stepname>_get.c` und `<stepname>_put.c`, für einen Resource Manager die Files `<servicename>_decode.c` und `<servicename>_encode.c` verwendet.

Ist man mit der Definition der Step Computation Server und Resource Manager fertig, kann man schließlich die APRICOTS-Komponenten definieren. Hier bekommt man entsprechend alle vorhandenen Step Computation Server und Resource Manager angezeigt, aus denen man beliebig welche auswählt. Einzige Einschränkung: Pro APRICOTS-Komponente darf ein Step Computation Server oder Resource Manager nicht mehrfach ausgewählt werden.

Ist man mit der Definition aller APRICOTS-Komponenten (also der Erstellung der statischen Konfiguration) fertig, kann man die dynamische Konfiguration angehen. Über diesen Teil des APRICOTS User Interface ist es möglich, die zuvor definierten

APRICOTS-Komponenten zu starten. Dies bedeuted, daß sie jetzt ins Kommunikationssystem aufgenommen werden und ab sofort für Service-Aufrufe verfügbar sind.

Laufen bereits APRICOTS-Komponenten (es sind welche gestartet worden), kann man sie hier wieder stoppen. Diese Möglichkeit wird zum Beispiel dann benötigt, wenn man die APRICOTS-Komponenten umgestalten möchte, man sie also erweitern möchte oder einen Step Computation Server / Resource Manager wieder aus ihr entfernen möchte.

Diese Schritte müssen natürlich nicht mehr alle ausgeführt werden, wenn man den x-ten ConTract erstellt. Dann kann man möglicherweise auf schon definierte Services, Step Computation Server / Resource Manager oder gar fertige APRICOTS-Komponenten zurückgreifen.

Kapitel 5

Die Beteiligung anderer Komponenten am ConTract

An einem ConTract sind alle Komponenten[1] in irgendeiner Art beteiligt. Für die Erstellung und Benutzung eines ConTracts sind jedoch die drei Komponenten *Kommunikationssystem, APRICOTS User Interface* und der *ConTract–Manager* von besonderem Interesse.

Das Kommunikationssystem dient der Verteilung der Step- und Service-Aufrufe. Man muß deshalb seine Schnittstelle bei der Programmierung der Steps und Services kennen.

Das APRICOTS User Interface wurde schon ausführlich im Vorigen Kapitel behandelt.

Der ConTract–Manager schließlich führt den ConTract dann aus, indem er die im Skript deklarierten Elemente richtig umsetzt und ausführt.

5.1 Das Kommunikationssystem

Jeder Step muß in einem Step Computation Server und jeder Service in einem Resource Manager verfügbar sein, damit er aufgerufen und ausgeführt werden kann. Step Computation Server und Resource Manager sind wiederum zu APRICOTS-Komponenten zusammengefaßt, die vom Kommunikationssystem verwaltet werden. Nur über das Kommunikationssystem ist ein Step oder Service letztlich „ansprechbar“. Dieses besitzt die Informationen, ob ein Step / Service überhaupt in einer gestarteten APRICOTS-Komponente vorhanden ist, und wenn ja, in welcher APRICOTS--Komponente genau der Step / Service angeboten ist. Außerdem verfügt es über die

[1] außer den ausführlicher behandelten sind dies der Transaction–Manager, der Resource–Manager und andere

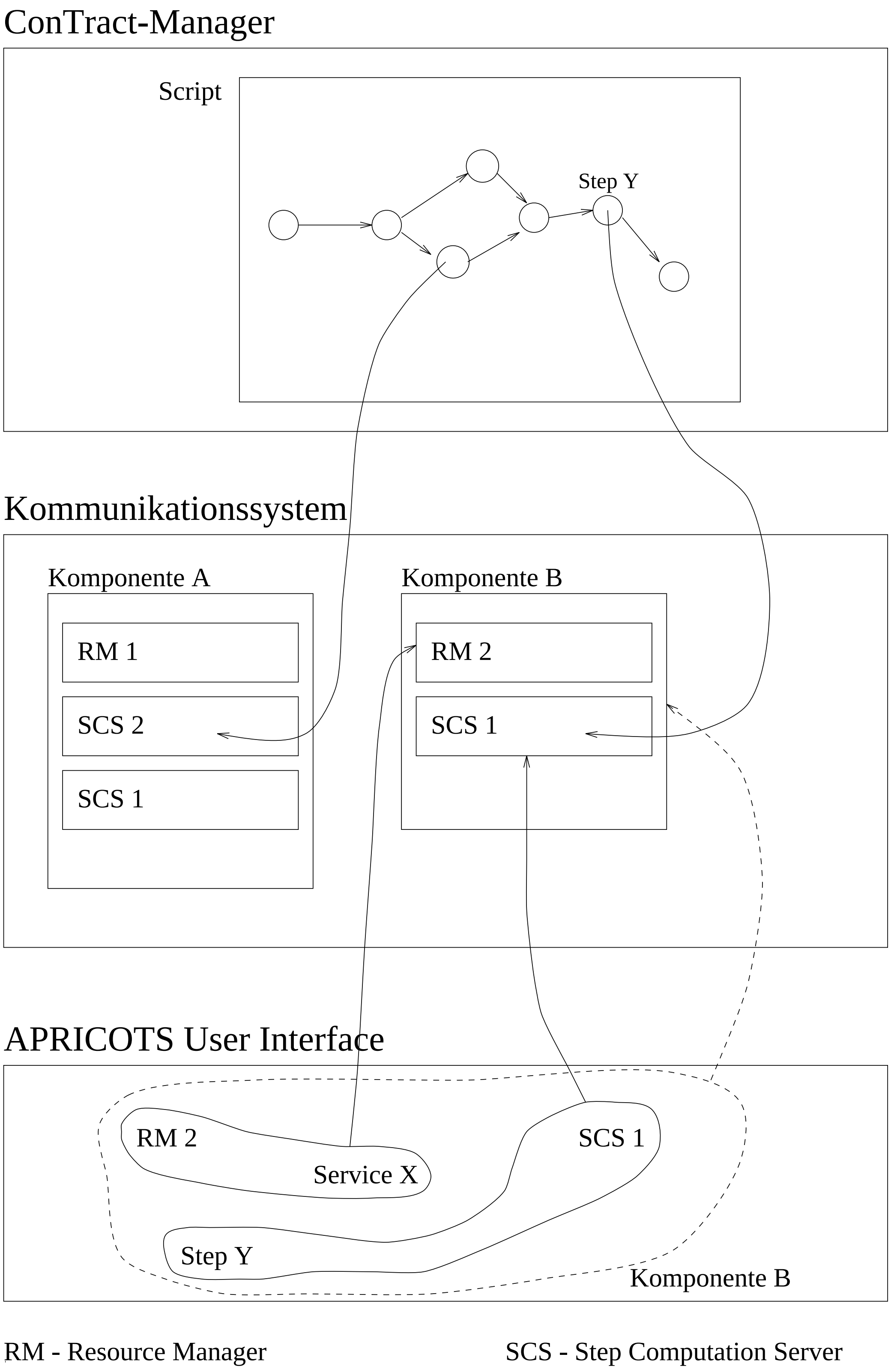

Abbildung 5.1: Die drei Komponenten und ihr Zusammenhang

Information, auf welchem Rechnerknoten diese APRICOTS-Komponente als Prozeß läuft.

Das Kommunikationssystem ordnet also den Aufruf eines Steps oder Services einer APRICOTS-Komponente zu. Für einen solchen Aufruf, den (außer dem ConTract–Manager beim Aufruf eines Steps[2]) nur die Steps und Services benutzen, muß die vom Kommunikationssystem vorgegebene Schnittstelle verwendet werden.

Die Schnittstelle ist so gestaltet, daß sie für alle Variationen eines Service-Aufrufs vom Aufbau her die gleiche ist. Dies möchte ich etwas genauer erläutern. Zuvor soll aber folgende Vereinbarung getroffen werden: Steps sind für das Kommunikationssystem nichts weiter als ein Service. Das Kommunikationssystem kennt auch keinen Unterschied zwischen einem Step Computation Server und einem Resource Manager, da die Schnittstelle dieser beiden zum Kommunikationssystem hin gleich ist. In diesem Abschnitt benutze ich deshalb den Begriff „Resource Manager“, auch wenn hier „Step Computation Server“, und „Service-Aufruf “, auch wenn hier „Step-Aufruf“ stehen könnte. Entsprechend wird der Begriff „Service“ verwendet, auch wenn „Step“ gemeint sein könnte.

Grundsätzlich kann ein Service synchron oder asynchron aufgerufen werden. Steps (und auch die von ihnen benötigten Services) sind jedoch sequentielle Einheiten. Es macht somit keinen Sinn, wenn sie asynchrone Aufrufe tätigen. Deshalb die nächste Einschränkung für diesen Abschnitt: Es sollen nur synchrone Aufrufe betrachtet werden.

Die Schnittstelle hat folgende Parameter (die genaue Schnittstelle entnehme man [Laube92]):

```
fehler = RM_call_sync (
                meine_adresse,
                ...,
                zeiger_auf_dienstadresse,
                ...,
                dienstname,
                rm_class,
                lokalitaet,
                nachrichtenlaenge,
                nachricht )
```

Erklärungen zu den einzelnen Parametern:

- *meine_adresse*
  In diesen Parameter wird vom Kommunikationssystem die Adresse des Aufrufers eingetragen, damit eine Antwort des aufgerufenen Dienstes richtig vom Kommunikationssystem zugeordnet werden kann.

[2] Irrtum vorbehalten: es können auch andere Komponenten Service-Aufrufe im hier verwendeten Sinne tätigen, mir sind momentan jedoch keine solchen Komponenten bekannt.

- *zeiger_auf_dienstadresse*
  Wenn die Adresse des Resource Managers unbekannt ist, wird hier 0 (Null) eingetragen. Dies ist immer beim ersten Aufruf eines Services der Fall.
  Diesen Parameter kann man dann nutzen, wenn ein Step / Service mehrmals hintereinander einen Dienst aufruft und dabei jedesmal den gleichen Resource Manager mit der Service-Ausführung beauftragen möchte.
  Beispiel: Der Resouce Manager, der den Dienst *drehen* ausgeführt hat, soll auch den Dienst *bohren* ausführen. Hier benutzt man dann die Zuweisung:
  `zeiger_auf_bohren_adresse = zeiger_auf_drehen_adresse`
  und übergibt beim Aufruf `RM_call_sync(...)` den Parameter
  `zeiger_auf_bohren_adresse`.

- *dienstname*
  Hier gibt man den Service-Namen des gewünschten Dienstes an. Dieser Name entspricht nicht unbedingt dem Funktionsnamen des aufgerufenen Services.
  Beispiel: der Service-Name kann *bohren* sein, die Funktion, die diesen Dienst letztlich ausführt heißt *werkstueck_bohren()*.

- *rm_class*
  Hier besteht die Möglichkeit, den Namen des Resource Managers anzugeben, der den Service ausführen soll, oder aber 0 (Null), falls der Resource Manager egal ist.
  Beispiel:
  der Resource Manager *Bohrer* bietet den Dienst *bohren*
  der Resource Manager *Bearbeitungszentrum* bietet den Dienst *bohren*

- *lokalitaet*
  Hier gibt der Aufrufer an, ob der gewünschte Service lokal, d. h. auf dem gleichen Rechnerknoten oder aber remote (entfernt) ausgeführt werden soll. Man kann die Verteilung aber auch dem Kommunikationssystem überlassen.
  Mögliche Belegungen sind:
  0 : es ist egal, wo der Service ausgeführt wird
  1 : der Service soll lokal ausgeführt werden
  2 : der Service soll remote ausgeführt werden

  Über die genauen Verteilungsstrategien informiere man sich in [Laube92]. Dort wird auch erläutert, wie die Wahl für den Fall *lokalitaet = 0* getroffen wird. Außerdem wird beschrieben, wie das Kommunikationssystem auf einen „lokal angeforderten“ Service reagiert, der nicht lokal, dafür aber remote verfügbar ist und umgekehrt (*lokalitaet = 2*).

- *nachrichtenlaenge, nachricht*
  Diese Parameter dienen dazu, dem aufzurufenden Service eine Nachricht zu verschicken.
  Beispiel: der Service *bohren* benötigt die Artikelnummer und die Menge der zu

bohrenden Teile.
Diese Angaben werden vom Aufrufer in Form eines ASCII-Strings verschickt, wobei die Teile, aus denen sich der String zusammensetzt, durch „@“[3] getrennt sein müssen und deren Reihenfolge durch den Service, bzw. dessen zugehörige Funktion `<service_name>_decode()` festgelegt ist.
Für die Gegenrichtung gilt entsprechendes. Schickt ein Service an den Aufrufer eine Nachricht zurück, muß der Service die Reihenfolge der Einzelstücke des ASCII-(Antwort-)Strings nach den Angaben des Aufrufers ordnen.

Zusammenfassung: Wie das Kommunikationssystem die Vorgaben letztlich verarbeitet, bleibt dem Aufrufer verborgen. Für ihn stellt sich die Schnittstelle immer einheitlich dar.

5.2 Der ConTract–Manager

Der ConTract–Manager zeichnet sich dafür verantwortlich, den ConTract korrekt auszuführen. Er muß die Steps in der im Skript angegebenen Reihenfolge aufrufen, die angegebenen Transaktionen beachten, auf Fehler richtig reagieren.

Ansonsten soll er die Wünsche des Benutzer zur ConTract-Ausführung erfüllen. Dies sind

- Starten des ConTracts
- Anhalten des ConTracts
- Fortsetzen des ConTracts
- Stoppen des ConTracts
- Migrieren des ConTracts

Wünscht der Benutzer eine Kompensation des ConTracts, muß der ConTract–Manager die entsprechenden Kompensations-Aktionen einleiten.
Die Aufgaben des ConTract–Managers im allgemeinen und für die angegebenen Punkte im speziellen sind in [Wahl91] genauer erläutert.

[3] dieses Trennzeichen wurde von den APRICOTS-Programmierern festgelegt

Kapitel 6

Die Abwicklung des ConTracts

In diesem Kapitel werden die Möglichkeiten für Fehler bei der ConTract-Abwicklung vorgestellt, ihre bisherige Behandlung geschildert und aufgezeigt, welche Fehlerbehandlungsstrategie gegebenenfalls für den jeweiligen Fall angebracht wäre.

6.1 Fehler während des Step oder Service-Ablaufs

Hier gibt es zwei Möglichkeiten. Die erste ist, daß ein Programmierfehler vorliegt. Hier hängt es von der Schwere des Programmierfehlers ab, welches Verhalten das APRICOT-System zeigt. Hier hilft nur eines: Step bzw. Service-Code überarbeiten.

Die zweite Möglichkeit besteht darin, daß das Zwischenergebnis innerhalb eines Steps oder Services eine weitere Fortführung des Steps / Services nicht zuläßt. Dieses Zwischenergebnis kann das Ergebnis einer Datenbankanfrage sein, ein berechneter Wert, etc. Der Step / Service bricht hier mit einer Fehlermeldung ab. Es erfolgt eine Wiederholung des Steps (Bricht ein Service mit einer Fehlermeldung ab, wird sofort auch der Step abgebrochen, der diesen Service aufgerufen hatte.) Beispiel: Es sind nicht genügend Teile im Lager, um *anzahl* viele zu bohren.

6.2 Ein Service ist nicht verfügbar

Eine Fehlermeldung „Service ist nicht verfügbar“ tritt auf, wenn das Kommunikationssystem feststellt, daß der von einem Step angeforderte Service in keiner APRICOTS-Komponente enthalten ist. Hier wird der Step abgebrochen und wiederholt. Besser wäre es, daß der ConTract–Manager den Contract anhält und eine (geeignete) Meldung an den Benutzer schickt, damit dieser den Service in eine APRICOTS-Komponente einbetten kann. Eleganter wäre es hier, wenn der benötigte Service automatisch vom APRICOT–System in einen Resource Manager und dieser in eine

APRICOTS-Komponente eingebettet werden würde, und danach mit der ConTract-Ausführung fortgefahren wird.

6.3 Invariantenfehler

Eine Entry_Invariante schlägt fehl, wenn der von der Exit_Invariante vorgegebene Datenbankzustand verändert wurde. Dies kann dann der Fall sein, wenn die Exit_Invariante keine Sperre angefordert hatte (z. B. bei den POLICIES escrow und check_revalidate). Hier wird der Konfliktlösungsstep ausgeführt.

6.4 Fehlerhafte Benutzereingaben

Benutzereingaben müssen vom Step auf ihre Korrektheit geprüft werden. Sind sie mit einem Fehler behaftet (z. B. wurde keiner von fünf möglichen Werten eingegeben), gibt es zwei Möglichkeiten, den Fehler zu behandeln, wobei in der bisherigen Realisation die erste angewendet wurde:

1. Der Step bricht ab. Er wird wiederholt.
2. Der Step erfragt so lange eine Eingabe, bis diese befriedigend ausfällt. Hier ergeben sich weiter keine Auswirkungen.

Kapitel 7

Probleme, Schwachpunkte, Verbesserungsvorschläge

In diesem Kapitel werden die bei der Erstellung des ConTracts „Auftragsabwicklung“ aufgetretenen Konflikte, Probleme und bemerkten Schwachpunkte aufgelistet und zum Teil mögliche Verbesserungen angegeben. Inwieweit diese Lösungsvorschläge tatsächlich realisiert werden können wird hier nicht untersucht. Sie sollen vielmehr als Anregung zu einer Diskussion dienen. Die Auflistung der einzelnen Punkte erfolgt ohne Ordnung, da sie die unterschiedlichsten Bereiche tangieren.

7.1 Invarianten

7.1.1 Invarianten zu Pseudosteps

Eine Invariante bzgl. eines Pseudosteps ist nicht möglich, erscheint aber im vorliegenden Skript im ersten Moment als sinnvoll. Hier soll im speziellen der Zustand des Lagers, genauer die im Lager vorhandene Menge von bestimmten Teilen für die später folgende Fertigung erhalten werden. Der gegebene Zustand des Lagers soll als spätere Bezugsgrundlage dienen, egal welcher Zweig durchlaufen wird, also unabhängig davon, ob eine Teilebeschaffung notwendig ist oder nicht.

- Eine Möglichkeit wäre, eine Exit_Invariante für den Step `S16: Lager` und zusätzlich beim Step `S18: Teilebeschaffung` anzugeben (die gleiche). Falls keine Teile nachbestellt werden müssen sichert die Invariante nach dem „Lager“ den dortigen Zustand. Wird eine Nachbestellung von Teilen jedoch notwendig, werden beide angegebene Exit_Invarianten ausgeführt. Diese sichern dann folglicherweise jeweils einen anderen Lagerzustand. Hier müßte für den Step `S20: Fertigung` sichergestellt sein, daß sich dessen Entry_Invariante auf die richtige, in diesem Fall aktuellere Exit_Invariante bezieht. (Es werden in diesem Fall beide Exit_Invarianten ausgeführt; vgl. dazu auch die Überlegungen in Abschnitt 7.1.2)

- Da bisher Invarianten nur bei „normalen" Steps möglich sind, ergeben sich für die Ausführung der Invarianten keine Probleme. Hier ist jeweils derjenige Step Computation Server zuständig, der den Step als Dienst bereitstellt, bei dem zuvor / danach die Invariante auftritt. Durch die Verwendung des Steps `S26: Leerer_Step` ist die Zuordnung, wer für die Invariantenausführung verantwortlich ist, eindeutig gewährleistet.

  Der sich dabei ergebende Nachteil:

  - Es muß der Step `Leerer_Step` und zusätzlich die Funktionen `Leerer_Step_get()` und `Leerer_Step_put()` codiert werden, auch wenn die letzteren beiden in diesem Fall „dummy"–Funktionen entsprechen, da der Step weder Kontextelemente benötigt, noch welche produziert.
  - Es muß ein Step Computation Server erstellt werden, der diesen Step als Dienst anbietet. Gibt es diesen Step Computation Server nicht in irgendeiner APRICOTS-Komponente, kann der ConTract nicht ausgeführt werden: nicht nur weil die Invariante nicht ausführbar ist, sondern schon weil der Step selbst dem Kommunikationssystem als Dienst bekannt ist.
  - Es wird ein Step mehr ausgeführt, der nicht nur überflüssig ist, sondern eine Fehlerquelle mehr bedeuted (Fehler beim Aufruf, Step nicht als Dienst bekannt, etc.).

- Läßt man andererseits Invarianten für Pseudosteps zu, ergeben sich aber auch hier Probleme:

  - Der Skript-Compiler müßte diese Konstruktion verarbeiten können
  - Wer ist jetzt für die Ausführung der Invarianten zuständig? Pseudosteps werden nur vom ConTract–Manager bearbeitet (sie beinhalten die Kontrollkonstrukte). Dadurch wäre also der ConTract–Manager für die Invariantenausführung verantwortlich. Direkt, indem er sie selbst überprüft oder indirekt, indem er die Bearbeitung abgibt (an wen?).

Zusammenfassung: Betrachtet man den Aufwand der bei den genannten Aspekten notwendig ist, läßt sich feststellen, daß die Angabe einer Exit_Invariante sowohl bei Step `S16: Lager`, als auch bei Step `S18:Teilebeschaffung` die einfachste und am wenigsten aufwendige Lösung ist.

Da bisher jedoch eine eindeutige und korrekte Zuordnung (also in diesem Fall der Bezug der Entry_Invariante auf die richtige Exit_Invariante) nicht gegeben ist, wurde die Version gewählt, die sich mit einem zusätzlichen Step, dem `Leerer_Step` behilft.

Für zukünftige Erstellungen von ConTracts ist diese Variante auch nicht mehr mit einem allzu hohen Aufwand verbunden, da dann der `Leerer_Step` schon in programmierter Form vorliegt und wiederverwendet werden kann. Möglicherweise kann man sogar zusätzlich auf einen schon fertigen Step Computation Server zurückgreifen, der diesen Step als Dienst anbietet.

7.1.2 „Geschachtelte“ Invarianten

Die hier verwendete Bedeutung des Begriffs „geschachtelte Invarianten“ soll an einem Beispiel verdeutlicht werden. Siehe dazu Abbildung 7.1. Weitere Unterstellung: die Steps sollen sequentiell aufeinander folgen.

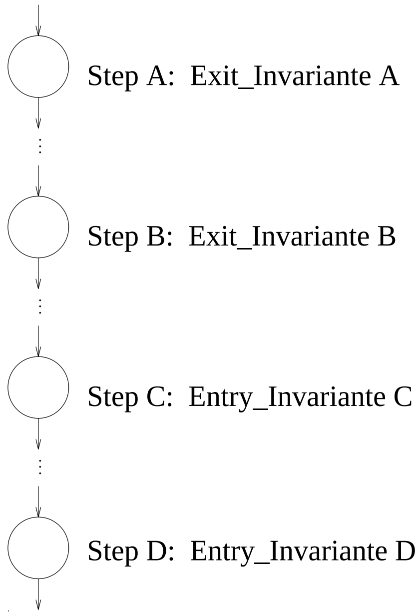

Abbildung 7.1: Beispiel für eine Folge von Invarianten

Hier gibt es mehrere Möglichkeiten:

1. Man setzt voraus, daß diese Konstruktion nicht möglich sein soll. Eine Entry_Invariante soll sich immer genau auf die vorangegangene Exit_Invariante beziehen und um dies sicherzustellen darf keine „Schachtelung“ wie diese auftreten.
Folgerung: Exit_Invariante und Entry_Invariante müssen abwechseln.

2. Voraussetzung: Eine Entry_Invariante soll sich immer auf die genau vorangegangene Exit_Invariante beziehen.
Invariante C bezieht sich also auf Invariante B.
Invariante D bezieht sich ebenfalls auf Invariante B, da diese die vorangehende Exit_Invariante ist.
Folgerung: Invariante A ist überflüssig.

3. Zwei mögliche Varianten:
   - C bezieht sich auf A, D bezieht sich auf B
   - C bezieht sich auf B, D bezieht sich auf A

   Bei beiden Alternativen muß im Skript die Möglichkeit bestehen, eine Angabe zu machen, auf welche Exit_Invariante sich die Entry_Invariante bezieht. Denkbar wäre hier die Angabe des entsprechenden Steps, auf dessen Exit_Invariante ein Bezug bestehen soll.

7.1.3 Die Formulierung von Invarianten

Invarianten werden vom Step Computation Server überprüft. D. h. er führt eine Datenbankanfrage durch. Selbst wenn die Anfrage an sich korrekt ist, also keine Fehlermeldung von der Datenbank zurückgegeben wird, kann der Step Computation Server im Normalfall nicht entscheiden, ob das Ergebnis der Anfrage korrekt ist.

Beispiel: Es wird geprüft, ob mehr als 5 Teile eines bestimmten Artikels (charakterisiert durch seine Artikelnummer) im Lager sind.
Die Anfrage lautet:

```
select artikelnummer
from lager
where artikelnummer = :artikelnr
and menge > 5;
```

Sollten nur 3 Teile am Lager sein, verläuft die Anfrage korrekt, es wird aber eine leere Menge zurückgegeben.

Es sind noch viele andere Anfragen denkbar. Aber immer müßte dem Step Computation Server „gesagt“ werden, bei welchen Ergebniswerten die Anfrage als „richtig“ und bei welchen sie als „falsch“ zu interpretieren ist.

Eine Lösung, dieses Problem zu umgehen, wäre folgende:
Die Formulierung ist so zu gestalten, daß ein „richtige“Anfrage einen Ergebniswert > 0 zur Folge hat, „falsche“ Ergebnisse einen Wert $= 0$ bedingen.

Die Anfrage für das obige Beispiel müßte dann lauten:

```
select count(artikelnummer)
from lager
where artikelnummer = :artikelnr
and menge > 5;
```

Gibt es den Artikel nicht oder maximal 5 mal, ist der Ergebniswert $= 0$, sonst > 0.

Man möge sich anhand weiterer Beispiele überlegen, warum ich gerade die angegebene Einteilung für die Interpretation der Ergebniswerte für sinnvoll halte und deshalb für die Formulierung der Invarianten vorschlage.

7.2 Fehlerbehandlung

Je mehr Skripte aus verschiedenen Anwendungsbereichen entwickelt werden (zum Beispiel Bank, Fertigung, Dienstleistung, etc.), umso umfangreicher werden die auftretenden möglichen Fehler. Auf viele davon müßte mit einer speziellen Aktion reagiert werden. Da hierfür der ConTract–Manager verantwortlich ist, müßte dieser dann theoretisch eine Unmenge von Fehlern behandeln können, d. h. somit auch, er müßte ständig erweitert werden.

7.2.1 Fehlerklassen

Vorschlag:
Der ConTract–Manager bearbeitet eine fest vorgegebene geeignete Menge von Fehlern. Alle anderen Fehler werden von einer weiteren Komponente im APRICOT–System bearbeitet. Vergrößert sich im Laufe der Zeit die Menge der von den Steps und Services zurückgegebenen Fehler, müßte dann diese Komponente erweitert werden. Dies ist ist zwar auch aufwendig, dürfte jedoch leichter zu realisieren sein.

Eine Aufteilung der zu behandelnden Fehler könnte so aussehen:

- *Fehler, die der ConTract–Manager direkt verarbeitet:*
  Dazu gehören ablaufspezifische Fehler wie z. B. Fehler beim Kommunikationssystem: Timeout bei Service-Aufruf, Dienst ist nicht bekannt / verfügbar. Diese Fehler werden dem Step oder Service vom Kommunikationssystem gemeldet, welche dann sofort ihre Aktion beenden und diese Fehlermeldung an den ConTract–Manager weitergeben (zumindest in der momentanen Realisierung).

  Als ein weiterer ablaufspezifischer Fehler kann ein Fehler bei den Interaktionen mit dem Benutzer (im Moment durch die Dienste terminal_input / -output realisiert) angesehen werden.

  Zusammenfassend läßt sich sagen, daß bei dieser Art von Fehlern die Reaktion des ConTract–Managers darin besteht, den Step zu wiederholen.

- *Fehler, die die weitere Komponente bearbeiten müßte:*
  Anwendungsspezifische Fehler, genauer: physikalische Fehler. Dies möchte ich am Beispiel „Endmontage“ zeigen:
  Ein Teil geht während der Montage kaputt. Da es sich hier um reale Abläufe handelt, kann nicht einfach ein *ABORT* durchgeführt und der Step nochmals gestartet werden, sondern es sind komplexe Reaktionen notwendig. Es muß geprüft werden, wodurch der Fehler entstanden ist, ob noch Teile heil geblieben sind, die dann gegebenenfalls ins Lager gestellt werden, ob die Endmontage ohne weitere Aktion wiederholt werden kann, da noch genügend Teile vorhanden sind oder ob gar Teile nachbestellt und / oder nochmals gefertigt werden müssen, etc.

Für solche komplexen Fehlerbehandlungen könnte der in [Schmidt92] vorgestellte „Physical Recovery Manager" eingesetzt werden.

7.2.2 Schnittstelle zu einer weiteren Komponente

Die Schnittstelle für diese zusätzliche Komponente sollte enthalten:

- Einen Parameter, über den der genaue Fehlercode weitergegeben wird, den zuvor der ConTract–Manager erhalten hat.
- Einen Parameter, der den Namen des Steps (oder eine sonstige eindeutige Charakteristik des Steps), bei dem der Fehler auftrat, beinhaltet.
- Rückgabewert der aufgerufenen Komponente.
  Die hier von der Komponente zurückgegebenen Werte sollten wiederum so gestaltet sein, daß der ConTract–Manager weiß, ob eine erneute Ausführung des Steps veranlaßt werden soll, oder ob mit einem anderen Step (beinhaltet auch den nächsten) fortgefahren werden soll.

  Hier kann eine detailliertere Diskussion ansetzen:

  - inwieweit kann es sinnvoll sein, nochmals mehrere Steps auszuführen ?
  - wenn es sinnvoll ist, mehrere Steps auszuführen, dann welche ?
  - wie sind diese dem ConTract–Manager gegebenenfalls mitzuteilen (Namen aller auszuführenden Steps oder des einen Steps, mit dem im Ablauf nochmal begonnen werden soll [Beispiel Endmontage: nochmal alle Steps ab Fertigungsvorbereitung oder gar PPS ausführen]) ?
  - ConTract–Manager: er müßte in der Lage sein, hierzu an „beliebiger" Stelle im Skript nochmals aufzusetzen.
- weitere Parameter (wenn ja, welche sind noch notwendig / erscheinen als sinnvoll ?)

7.2.3 Abschließende Bemerkung

Inwieweit die vorgeschlagene Einteilung sinnvoll und / oder durchführbar ist, bedarf wohl noch einiger Diskussion und auch Untersuchungen, zumal sich die angesprochenen Recovery Manager (Physical Recovery Manager) noch im Entwicklungsstadium befinden.

7.3 Wiederverwendbarkeit von Steps und Services

Prinzipiell lassen sich Steps und Services auf jeden Fall wiederverwenden.

Man benötigt für ihre Wiederverwendung eine detaillierte Angabe über:

- die Parameter, die an die Funktion zu übergeben sind (Schnittstellenbeschreibung)
- eine genaue Beschreibung des Codes:
  - was wird gemacht
  - welche Voraussetzungen müssen gegeben sein: benötigte Datenbank, etc.
  - welche Services werden aufgerufen

Hierzu wird eine geeignete Verwaltung benötigt (vgl. auch [BuRa90] und [Masman91]).

Einige Einschränkungen für ihre Wiederverwendbarkeit gibt es meines Erachtens dennoch. Diese Einschränkungen sind im einzelnen:

- Bei einer Step-Wiederverwendung (und sei es, daß man einen Service jetzt als Step verwendet) müssen die von diesem benötigten Parameter als Kontextelemente im zugehörigen Skript als *IN-Parameter* beim entsprechenden Step und zudem noch als *OUT-Parameter* in im (direkten) Ablauf vorher ausgeführten Steps auftreten.
- Ruft ein Step / Service, der wiederverwendet werden soll, selbst einen Service auf, muß man dies später bei der Erstellung der Resource Manager berücksichtigen: nur wenn wirklich jeder irgendwie benötigte Dienst in einer laufenden APRICOTS-Komponente enthalten ist, kann der ConTract durchgeführt werden. Ist dies nicht der Fall, bricht der ConTract ab, weil das Kommunikationssystem den entsprechenden Dienst nicht kennt.
- Wird vom Step / Service, der wiederverwendet werden soll, ein Service aufgerufen, muß der Letztere eventuell nicht nur in irgendeiner APRICOTS-Komponente vorhanden sein, sondern sogar in einer speziellen. Hiermit meine ich, daß die Art des Service-Aufrufes, wenn dieser erst einmal programmiert ist, festliegt. Da es verschiedene Möglichkeiten für die Parameterbelegung beim „*RM_call*“ gibt, bedeuted dies, daß für einen *remote*-Aufruf der aufzurufende Service im System dann auch remote verfügbar sein muß. Hier nützt es wenig, wenn der Dienst lokal vorhanden ist. (Hier hängt es allerdings auch davon ab, wie das Kommunikationssystem eine *remote*-Anforderung umsetzt, wenn der Dienst nur lokal vorhanden ist.)

7.4 Interaktionen mit dem Benutzer

Bisher steht für die Interaktion mit dem Benutzer ein Service zur Verfügung, der nur an demjenigen Terminal die Ein- und Ausgabe unterstützt, an dem der diesen Dienst anbietende io_server (vom Benutzer) gestartet wurde.

Für spätere Anwendungen ist dieses Verfahren jedoch nicht mehr geeignet. Hier benötigt das APRICOT–System die Information, an welchem Terminal sich der Benutzer befindet. Stoppt der Benutzer beispielsweise den ConTract, um ihn an einem anderen Terminal fortzusetzen, wäre es wünschenswert, daß das APRICOT–System diese Information dann dazu benutzt, die Ein- und Ausgaben des betreffenden ConTracts entsprechend umzuleiten.

Zu diesem Punkt kommt hinzu, daß zur Zeit nur ein einziger io_server gestartet werden kann, d. h. es kann zu jedem Zeitpunkt nur ein einziger ConTract ablaufen. Dies entspricht jedoch nicht der Konzeption des APRICOT–Systems, das eine parallele Ausführung mehrerer ConTracts vorsieht. Hier müssen geeignete Mechanismen gefunden werden, die den Anforderungen an eine sinnvolle Interaktion gerecht werden.

Es ist vorstellbar, daß der ConTract–Manager jedem Step / Service und diese den Subservices die Kennung des Terminals zukommen lassen, an dem eine eventuelle Interaktion stattzufinden hat. Der Step / Service gibt diese Kennung dem Ein- / Ausgabe-Service weiter.

Weiterhin ist es überlegenswert, ob ein Ein- / Ausgabe–Server nicht vom ConTract–Manager (oder einer sonstigen geeigneten Komponente) gestartet wird, also der Benutzer hierfür in Zukunft nicht mehr verantwortlich ist.

Bisher wurden keine konkreten Überlegungen durchgeführt, auf welche Art und Weise man das geschilderte Problem bewältigen könnte. Folglicherweise wurden die Steps für den ConTract „Auftragsabwicklung“ so programmiert, daß sie die bestehende Möglichkeiten nutzen. Sollten sich in Zukunft große Änderungen bezüglich der Ein- und Ausgabe ergeben, sind diese bei der Step-Programmierung zu berücksichtigen und die bisher bestehenden Steps folglicherweise nicht wiederverwendbar.

7.5 Kompensation

Soll ein begonnener ConTract und somit dessen bereits durchgeführten Aktionen zurückgesetzt werden, müssen Kompensationsaktionen ausgeführt werden. Eine Kompensation wird stets durch den Benutzer des ConTracts veranlaßt. Dies kann zu einem beliebigen Zeitpunkt nach dem Start des ConTracts geschehen.

Da die Aktionen, die vor dem Kompensationsbeginn liegen, dauerhaft durchgeführt worden sind, bedeuted dies, daß diese Aktionen durch logische „Gegenaktionen“ rückgängig gemacht werden müssen.

Beispiel: Auftrag abgeben — Auftrag stornieren.
Es läßt sich zu allen Aktionen wenigstens eine Kompensationsaktivität angeben, und sei es auch nur die, daß nichts getan werden muß.

Für den ConTract und speziell das Skript heißt dies, daß zu jedem Step ein oder mehrere Kompensations–Steps angegeben werden müssen. Der ConTract–Manager hat dann bei einer Kompensation zu beachten, daß er nur für bereits vollständig ausgeführte Steps die Kompensations–Steps aktiviert.

Dies ist der Grund, weshalb die Kompensations–Steps nur *IN–Parameter* haben dürfen, die „bis dahin" von den „normalen" Steps als *OUT–Parameter* vorgegeben wurden.

7.5.1 Ausführung der Kompensations–Steps

Da ein Kompensations–Steps zum einen genau diejenigen Änderungen (logisch) zurücksetzt, die sein zugehöriger „normaler" Step vollzogen hat, und zum anderen nur auf Kontextelemente zurückgreift, die bereits vorliegen, könnten Kompensations–Steps theoretisch parallel durchgeführt werden (vgl. hierzu auch [WäReu91]).

Ich möchte hier anhand des Skriptes erläutern, was gegen diese Überlegung spricht:
Im Skript „Auftragsabwicklung" ist für den Step `S20: Fertigung` als Kompensation das Einlagern der fertigen Teile vorgesehen: `C20: Teile_einlagern`.
Für Step `S22:Endmontage` ist der Kompensations–Step `C22: Demontage` gedacht, in dem das montierte Produkt wieder in seine Einzelteile zerlegt wird.

Es leuchtet ein, daß beide Kompensations–Steps genau diese Aktionen umfassen. (Man nehme zur Veranschaulichung nur einmal an, daß die Kompensation direkt nach der durchgeführten Fertigung begonnen wird.)

Es ist ebenfalls logisch, daß bei einer Kompensation die erst nach der bereits durchgeführten Endmontage beginnt, die Reihenfolge erst *demontieren*, danach die einzelnen *Teile ins Lager bringen* fest vorgegeben ist.

Wie man sieht, können jetzt die beiden oben beschriebenen Kompensations–Steps nicht parallel ausgeführt werden. Wie soll man schließlich Teile ins Lager bringen, wenn noch keine vorliegen? (Abgesehen vom Endprodukt, von dem jedoch vorausgesetzt wird, daß es in seine Einzelteilen zerlegt und erst dann eingelagert wird.)

Hier müßte die Möglichkeit bestehen, im Script bei der Auflistung der Compensation–Steps geeignete Kontrollflußdeklarationen anzugeben. Sie müßten „Zeitpunktabhängigkeiten" enthalten, da unterschiedliche Strategien für die Durchführung der Compensation–Steps bestehen (je nach dem Zeitpunkt, an dem der Benutzer eine Kompensation anfordert). Zu einem „frühen" Zeitpunkt könnten die Kompensations–Steps tatsächlich parallel ausgeführt werden, zu einem „späten" Zeitpunkt nicht mehr unbedingt.

Am Beispiel des Skriptes:

früher Zeitpunkt (nach Beendigung von S20, vor Beginn von S22)	später Zeitpunkt (nach Beendigung von S22)
C20 ‖ C1 ‖ C4 ‖ C5 ‖ ...	(C22 → C20) ‖ C1 ‖ C4 ‖ C5 ‖ ...

C22 → C20 soll bedeuten: erst C22, dann C20 ausführen.

Die einzigste Möglichkeit ohne solche Kontrollkonstrukte (also mit den bisherigen Deklarationen) die Kompensation sinnvoll, d. h. logisch korrekt durchzuführen sehe ich darin, Kompensations–Steps in „umgekehrter Reihenfolge“ auszuführen. Dies bedeuted, der zuerst ausgeführte Step wird zuletzt kompensiert, der zuletzt durchgeführte Step zuerst.

Andererseits ist festzustellen, daß eine explizite Reihenfolge für die Kompensation nicht für alle Steps zwingend notwendig ist, sondern nur für einen kleinen Teil. Dabei läßt sich dieser Teil nicht durch bestimmte Kriterien und Merkmale eingrenzen, wie z. B. die Aussage „Alle Steps, die eine reale und zudem physikalische Änderung durchführen müssen in einer bestimmten Reihenfolge kompensiert werden“. Wenn eine Reihenfolge für die Kompensations–Steps notwendig wird, hängt diese vielmehr von der speziellen Art und Logik der durchgeführten realen Änderungen und dadurch bedingten Kompensationsaktionen ab.

Es läßt sich abschließend feststellen, daß die Einführung geeigneter Kontrollkonstrukte sinnvoll ist. Da die geordnete Ausführung von Kompensations–Steps — wie schon erwähnt — nur für einen kleinen Teil aller im Script vorkommenden Steps notwendig ist (wenn überhaupt), ist eine Ausführung der Kompensations–Steps in „umgekehrter Reihenfolge“ (aufgrund meiner oben genannten Begründung: es gibt bisher keine Kontrollkonstrukte) in der Mehrzahl der Fälle unnötig und übervorsichtig.

Mögliche Konstrukte könnten wie folgt aussehen (am Beispiel für die Compensation–Steps für die Fertigung und Endmontage):

- ```
  C20: BEGIN
       CASE < 22 : C20: Teile_einlagern ( ... );
       CASE >= 22 : -- ;
       END
  C22: BEGIN
         SEQ
         C22: Demontage ( ... );
         C20: Teile_einlagern ( ... );
         ENDSEQ
       END
  ```

  Es werden grundsätzlich alle Kompensations–Steps parallel aktiviert. Es ist bei jedem Kompensations–Step eine genauere Angabe zum weiteren Ablauf angegeben.
  ```

Bei einer Kompensation die nach Beendigung von Step `S20: Fertigung` und vor Beginn von `S22:Endmontage` durchgeführt wird, wird für `S20 Fertigung` der erste Fall, also `C20: Teile_einlagern` ausgeführt; man beachte die Angabe „< 22“!
Ist `S22 Endmontage` schon beendet, wird für `S20 Fertigung` keine Kompensation durchgeführt, da diese indirekt durch die Kompensation für `S22 Endmontage` abgedeckt ist.
Bereiche werden dann mittels `CASE ( > 100 ) AND ( <= 150 )` angegeben.

- jeweils mit Angabe eines Bereiches:

```
C20: BEGIN
     (1 - 21) : C20 : Teile_einlagern ( ... );
     (22 - 26 ) : -- ;
     END
C22: BEGIN
     (1 - 26) : SEQ
                C22: Demontage ( ... );
                C20: Teile_einlagern ( ... );
                ENDSEQ
     END
```

Die Angaben in (...– ...) bedeuten „je einschließlich“.

7.5.2 Angabe von Kompensations–Steps

Für viele Steps muß keine Kompensation durchgeführt werden. Da für diese Steps trotzdem Kompensations–Steps angegeben werden müssen, ist im Skript hier jeweils der „`Leerer_Step`“ aufgeführt.

Sollte der Benutzer eine Kompensation initiieren, wird dieser Step x-mal ausgeführt (im vorliegenden Skript bis zu 7 mal). Dieser Aufwand erscheint mir nicht nur überflüssig, er birgt auch die Gefahr in sich, daß bei der Ausführung dieses Kompensations–Steps etwas schiefgeht (die Komponente die diesen Steps anbietet wurde gestoppt, etc.).

Hier ist es angebrachter, daß für Steps die keiner Kompensation bedürfen auch wirklich gar keine Folgeaktion durchgeführt wird.

Diese Forderung könnte im Script so vermerkt werden, daß zwischen *BEGIN_COMPENSATIONS* und *END_COMPENSATIONS* nur diejenigen Steps aufgeführt sind, zu denen Compensation–Steps notwendig sind.
Alternative: eine Angabe wie z. B. `C1: ;` oder `C1: -- ;`, also eine symbolische Angabe für „hier ist nichts tun“.

Kapitel 8

Zukünftige Aspekte

8.1 Austausch Step — Service

In der vorliegenden Arbeit wurde relativ streng zwischen Step und Service unterschieden.

Die Gründe hierfür waren, daß ein Step mit Werten arbeitet, die aus dem Kontext geholt, bzw. dort abgelegt werden, während ein Service die für seinen Ablauf benötigten Vorgaben bei seiner Anforderung über das Kommunikationssystem „mitgeschickt“ bekommt, bzw. selbst wieder versendet.

Da das Holen / Ablagen der Kontextelemente für den Step und das Auspacken / Einpacken der Für den Service verschickten Werte in selbstständigen Funktionen vorgenommen wird, steht für einen zukünftigen Austausch der Step– und Service–Funktionen nichts im Wege.

Wird ein Step als Service verwendet, müssen die Parameter, die er benötigt (und bisher Kontextelementen entsprochen haben) jetzt eben an ihn verschickt und vor dem eigentlichen Aufruf entpackt werden.

Umgekehrt heißt dies für einen Service der als Step laufen soll, daß die benötigten Vorgabewerte als Kontextelemente vorhanden sein müssen, die dann vor den eigentlichen Funktionsaufruf aus dem Kontext geholt werden.

Folglicherweise sind für jeden Step / Service jeweils vier zusätzliche Funktionen bereitzustellen:
<*name*>`_get()`
<*name*>`_decode()`

<*name*>`_put()`
<*name*>`_encode()`

Ein solcher Austasuch wurde bisher nicht durchgeführt und bleibt zukünftigen Anwendungen vorbehalten.

8.2 Recovery Manager

Im APRICOT–System gibt es bislang noch keinen Recovery Manager; allerdings wurden bisher auch keine realen Aktionen von einen ConTract durchgeführt.

Dies ist ein weiterer Punkt, der einer Weiterentwicklung bedarf (und zum Teil in der Industrie schon angegangen wird).

Hier muß sich herausstellen, welche Fehler eines realen Ablaufs (eindeutig) erfaßt werden können und wie die zugehörigen Recovery Maßnahmen auszusehen haben und welche davon durchführbar sind.

Anhang A

Das Script „Auftragsabwicklung“

A.1 Scriptcode

```
ConTract Auftragsabwicklung

BEGIN_CONTEXT_DECL
kundennr        : long;
angebot         : long;
artnr           : long;
anzahl          : long;
angebot_ok      : long;
kosten          : long;
auftrag_ok      : long;
auftragnr       : long;
teile_ok        : long;
spedition       : string;
END_CONTEXT_DECL

BEGIN_SCRIPT
S1: Kundenanfrage (out: kundennummer : kundennr : long,
                   out: angebotsnummer : angebot : long,
                   out: bestellmenge : anzahl : long,
                   out: entscheidung : angebot_ok : long
                   /* 0 == ok, <>0 == nicht ok */
                  );
S2: IF (angebot_ok : long <> 0)
    S3: Auftragsablehnung (in: kundennummer : kundennr : long,
                           in: angebotsnummer : angebot : long
                          );
    ELSE
```

```
S4: Angebot_abgeben (in: angebotsnummer : angebot : long,
                     in: bestellmenge : anzahl : long,
                     out: preis : kosten : long
                    );
S5: Auftragserhalt (in: angebotsnummer : angebot : long,
                    out: auftrag : auftrag_ok : long
                    /* 0 == ok, <>0 == nicht ok */
                   );
S6: IF (auftrag_ok : long = 0)
    S7: PPS ();
        /* leerer Step */
    S8: PARBEGIN
        S9: SEQ
            S10: Auftragsbestaetigung
                          (in: angebotsnummer : angebot : long,
                           in: kundennummer : kundennr : long,
                           out: auftragsnummer : auftragnr : long
                          );
            S11: Speditionsauftrag
                          (in: auftragsnummer : auftragnr : long,
                           out: speditionsname : spedition : string
                          );
            ENDSEQ
        S12: SEQ
             S13: PARBEGIN
                  S14: Fertigungsvorbereitung ();
                       /* leerer Step */
                  S15: SEQ
                       S16: Lager
                            (in: angebotsnummer : angebot : long,
                             in: bestellmenge : anzahl : long,
                             out: artikelnummer : artnr : long,
                             /* Kennung des selbst zu fertigenden
                               Teils */
                             out: teile_vorraetig : teile_ok : long
                             /* 0== ok, <>0 == nicht ok */
                            );
                       S17: IF (teile_ok : long <> 0)
                            S18: Teilebeschaffung
                                 (in: angebotsnummer : angebot: long,
                                  in: bestellmenge : anzahl : long
                                 );
                            ENDIF
```

```
          S26: Leerer_Step()
               EXIT_INVARIANT
               ( EXECSQL
                  select count(*)
                  from Step_Lager
                  where :anzahl <=
                    (select sum(anzahl)
                     from Step_Lager
                     where artikelnummer = :artnr
                       and (zustand = 'roh'
                        or zustand = 'gedreht'
                        or zustand = 'gefraest'
                        or zustand = 'gebohrt'
                        or zustand = 'fertig')
                    )
               ENDSQL )
               POLICY: check_revalidate;
          ENDSEQ
     PAREND
S19: PARBEGIN
     S20: Fertigung
                (in: artikelnummer : artnr : long,
                 in: bestellmenge : anzahl : long
                )
     ENTRY_INVARIANT ( EXECSQL
       select count(*)
       from Step_Lager
       where :anzahl <=
            (select sum(anzahl)
             from Step_Lager
             where artikelnummer = :artnr
               and (zustand = 'roh'
                    or zustand = 'gedreht'
                    or zustand = 'gefraest'
                    or zustand = 'gebohrt'
                    or zustand = 'fertig')
            )
     ENDSQL );
     S21: Teilebereitstellung
                (in: angebotsnummer : angebot : long,
                 in: bestellmenge : anzahl : long
                );
          /* Teile bereitstellen, die nicht selbst
```

```
                              gefertigt werden */
                      PAREND
                 S22: Endmontage (in: angebotsnummer : angebot : long,
                                  in: bestellmenge : anzahl : long,
                                  in: artikelnummer : artnr : long
                                 );
                 ENDSEQ
            PAREND
            S23: Verpackung (in: auftragsnummer : auftragnr : long,
                             in: angebotsnummer : angebot : long,
                             in: bestellmenge : anzahl : long
                            );
            S24: Lieferschein (in: auftragsnummer : auftragnr : long,
                               in: kundennummer : kundennr : long,
                               in: preis : kosten : long
                              );
            S25: Versand (in: speditionsname : spedition : string,
                          in: auftragsnummer : auftragnr : long,
                          in: kundennummer : kundennr : long
                         );
        ENDIF
    ENDIF
END_SCRIPT

BEGIN_CONFLICT_RESOLUTION      /* fuer Steps mit Entry_Invariante */
S20: BEGIN
     S120: Teilenachbestellung (in: artikelnummer : artnr : long,
                                in: bestellmenge : anzahl : long
                               );
     END
END_CONFLICT_RESOLUTION

BEGIN_TRANSACTIONS
T1(S24, S25)
END_TRANSACTIONS

/***
BEGIN_COMPENSATIONS
S1: BEGIN
    C1: Leerer_Step ();
    END
S3: BEGIN
    C3: Leerer_Step ();
```

```
    END
S4: BEGIN
    C4: Leerer_Step ();
    END
S5: BEGIN
    C5: Auftragsstornierung (in: auftragsnummer : auftragnr : long
                           );
    END
S7: BEGIN
    C7: Neuplanung ();
    ** leerer Step; entspricht PPS ohne aktuellen Auftrag **
    END
S10: BEGIN
     C10: Auftrag_zuruecknehmen (in: auftragsnummer : auftragnr : long,
                                 in: kundennummer : kundennr : long
                                );
     ** Auftragsbestaetigung als ungueltig erklaeren **
     END
S11: BEGIN
     C11: Speditionsauftrag_stornieren
                          (in: speditionsname : spedition : string,
                           in: auftragsnummer : auftragnr : long
                          );
     END
S20: BEGIN
     C20: Teile_einlagern (in: artikelnummer : artnr : long,
                           in: bestellmenge : anzahl : long
                          );
     END
S21: BEGIN
     C21: Zukaufteile_einlagern (in: angebotsnummer : angebot : long,
                                 in: bestellmenge : anzahl : long
                                );
     END
S22: BEGIN
     C22: Demontage (in: angebotsnummer : angebot : long,
                     in: bestellmenge : anzahl : long
                    );
     END
S23: BEGIN
     C23: Lieferschein_ungueltig_stempeln
                          (in: lieferschein : liefernr : long
                          );
```

```
     END
S24: BEGIN
     C24: Auspacken (in: angebotsnummer : angebot : long,
                     in: bestellmenge : anzahl : long
                    );
     END
S25: BEGIN
     C25: Ruecktransport (in: speditionsname : spedition : string,
                          in: auftragsnummer : auftragnr : long
                         );
     END
S26: BEGIN
     C26: Leerer_Step ();
     END
END_COMPENSATIONS
***/
```

Literaturverzeichnis

[BuRa90] Holger Burkert, Bernd Raichle
Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext
Studienarbeit, IPVR, Universität Stuttgart, 1990

[Laube92] Dieter Laube
Transaktionsorientierte Kommunikationsmechanismen für das verteilte Transaktionssystem APRICOTS
Diplomarbeit, IPVR, Universität Stuttgart, 1992

[LoSch87] P. C. Lockemann, J. W. Schmidt (Hrsg.)
Datenbank–Handbuch
Springer–Verlag, 1987

[Masman91] Ioannis Masmanidis
ConTract Programmierung in APRICOTS: Sprachbeschreibung und Programmieranleitung
Arbeitspapier, IPVR, Universität Stuttgart, 1991

[ReuSch91] Andreas Reuter, Ursula Schmidt
„Transactions in Manufacturing Applications"
in: 4th Int. Workshop on High Performance Transaction Systems,
Asilomar, Sept. 1991

[Schmidt92] Ursula Schmidt
„A Framework For Automated Recovery in FMS"
in:
Proc. 2nd Int. Conf. on Automation, Robotics and Computer Vision,
Singapore, 1992

[WäReu91] Helmut Wächter, Andreas Reuter
„The ConTract Model"
in: A. K. Elmagarmid (Ed.):
Transaction Models for Advanced Database Applications,
Morgan Kaufmann Publishers, 1991

[Wahl91] Peter Wahl
Interpretative Abwicklung von Prädikat–Transaktionsnetzen
Studienarbeit, IPVR, Universität Stuttgart, 1991

[Zellmer91] Oliver Zellmer
Terminal_Output / _Input
Arbeitspapier, IPVR, Universität Stuttgart, 1991

[Zellmer92/1] Oliver Zellmer
APRICOTS User Interface – Manuel (Arbeitstitel)
Arbeitspapier, IPVR, Universität Stuttgart, 1992 (in Vorbereitung)

[Zellmer92/2] Oliver Zellmer
Eine graphische Oberfläche für das verteilte Transaktionssystem APRICOTS
Studienarbeit, IPVR, Universität Stuttgart, 1992