

Universität Stuttgart

Institut für Parallele und Verteilte
Höchstleistungsrechner

Diplomarbeit Nr. 1418

**Eine Benutzeroberfläche für
APRICOTS in JAVA**

Jürgen Auwärter

Prüfer: Prof. Dr.-Ing. A. Reuter
Betreuer: Dipl.-Inform. F. Schwenkreis

CR-Klassifikation: D.2.2, H.5.2.
Begonnen am: 15.05.1996
Beendet am: 15.11.1996

Inhaltsverzeichnis

Inhaltsverzeichnis.....	i
Abbildungsverzeichnis.....	iv
Tabellenverzeichnis.....	v
1 Einleitung.....	1
1.1 Einführung.....	1
1.2 Aufgabenstellung	1
1.3 Gliederung der Arbeit.....	2
2 Überblick.....	3
2.1 ConTracts	3
2.1.1 Das ConTract-Skript	3
2.1.2 Die ConTract-Steps	3
2.2 APRICOTS.....	5
2.2.1 Das System Administration Interface.....	6
2.2.2 Das Definition Interface	6
2.2.3 Das Monitoring Interface	6
2.2.4 Das Tasklist Interface	6
2.2.5 Der ConTract Manager.....	6
2.2.6 Die Stepserver	7
2.2.7 Der Transaktions Manager	7
2.2.8 Der Context Manager	7
2.2.9 Der Ressource Manager	7
2.2.10 Der Storage Agent	7
2.2.11 Der Benutzer Agent.....	8
2.3 CORBA	9
2.3.1 Wie funktioniert CORBA ?.....	9
2.3.2 Warum CORBA ?	11
2.3.3 Der Naming Service von CORBA	11
3 Java.....	13
3.1 Was ist Java ?.....	13
3.2 Wie funktioniert Java ?	13
3.3 Die Vor- und Nachteile des Java Einsatzes.....	14
3.3.1 Die Eigenschaften von Java	14
3.3.2 Die Vorteile von Java.....	15
3.3.3 Die Nachteile von Java.....	17
3.4 Die Ausführungsgeschwindigkeit von Java-Programmen	18
3.5 Java und das Thema Sicherheit	19
4 Das Prädikat-Transitionsnetz (PTN).....	21
4.1 Was ist ein Prädikat-Transitionsnetz ?.....	21
4.1.1 Die einzelnen Komponenten eines PTN	22
4.1.1.1 Der ConTract.....	22

4.1.1.2 Der Step-Block.....	23
4.1.1.3 IF-Block.....	24
4.1.1.4 WHILE-Block	24
4.1.1.5 REPEAT-Block.....	24
4.1.1.6 FOR-Block	25
4.1.1.7 CASE-Block.....	25
4.1.1.8 Parallel-Block.....	25
4.1.2 Transaktionen	26
5 Graphentheoretische Überlegungen.....	27
5.1 Einführung in die Graphentheorie.....	27
5.2 Planarität und Kreuzungsfreiheit.....	28
5.3 Planaritätskriterien	28
5.3.1 Der Satz von Kuratowski	29
5.3.2 Der Satz von Whitney	30
5.3.3 Der Satz von McLane.....	30
5.3.4 Die Euler'sche Polyederformel	31
5.4 Beweis der Planarität der einzelnen Komponenten eines PTN.....	32
5.4.1 ConTract und Step-Block.....	32
5.4.2 Parallel Block	32
5.4.3 IF-Block und CASE-Block	33
5.4.4 Schleifen.....	33
5.4.5 Transaktionen und Konfliktauflösungen.....	34
5.5 Fazit.....	36
6 Die Bearbeitung eines ConTract-Skripts	37
6.1 Das Einlesen eines PTN mit Hilfe des Storage Agents.....	37
6.1.1 Die Schnittstellen des Storage Agents	37
6.1.1.1 Der Verbindungsaufbau zum Storage Agent.....	37
6.1.1.2 beginSession.....	38
6.1.1.3 getConTractState	38
6.1.1.4 getStartTransition	38
6.1.1.5 getStepObjects.....	38
6.1.1.6 getPredicates.....	39
6.1.1.7 getTransition.....	39
6.1.1.8 getTransactions.....	39
6.1.1.9 freeSession	39
6.1.2 Die interne Darstellung im Hauptspeicher	39
6.2 Die Bearbeitung des PTN.....	40
6.2.1 Die Eliminierung der True-Prädikate	41
6.2.2 Das Ausblenden der Abort-Pfade.....	41
6.3 Die Einbettung des PTN in die Ebene.....	42
6.3.1 Der Algorithmus von Hotz.....	42
6.3.2 Ein eigener Algorithmus	43
6.3.2.1 Die Methode calculate_x()	44
6.3.2.2 Die Methode calculate_y()	46
6.3.3 Komplexitätsbetrachtung	48
7 Oberflächengestaltung und Benutzungskonzept.....	49
7.1 Kriterien der Benutzerfreundlichkeit.....	49

7.2 Die Bedienung des Monitoring Interface	51
7.3 Funktionen zur Manipulation der ConTracts	52
7.4 Möglichkeiten der Kommunikation	54
7.4.1 Die synchrone Darstellung (Observe)	55
7.4.2 Die asynchrone Darstellung (Snapshot)	57
7.5 Fehlertoleranz und Restart-Verhalten	57
7.5.1 Ausfall des Naming Service	57
7.5.2 Ausfall des Monitor Agent bzw. der Engine	57
7.5.3 Ausfall des Storage Agents	58
7.5.4 Restart-Verhalten	58
8 Zusammenfassung und Ausblick	59

Anhang

A Glossar	60
B Fehler in Java	62
B.1 Fehler, die in der Sprache Java entdeckt wurden	62
C Schnittstellen zum System	64
C.1 basedef.idl	64
C.2 monitorInterface.idl	66
C.3 storageagent.idl	69
D Wichtige Konstanten im ConTract-Skript	78
D.1 Typen von Transitionen	78
D.2 Typen von Steps	78
D.3 Typen von Prädikaten	79
D.4 Status einer Transaktion	79
E Das Makefile für das Monitoring Interface	80
F Die Arbeitsumgebung des Monitoring Interface	82
F.1 Starten des Programms	82
F.2 Bekannte Fehler	82
G Die Klassen und Dateien des Monitoring Interface	83
G.1 Beschreibung der Klassen	83
G.2 Beschreibung der restlichen Dateien	84
H Literatur	85

Abbildungsverzeichnis

Abbildung 1: Beispielskript mit fünf Steps.....	4
Abbildung 2: Der Aufbau von APRICOTS	5
Abbildung 3: Kommunikation mit CORBA	9
Abbildung 4: Interface- und Implementation Repository.....	10
Abbildung 5: Der Namensraum von APRICOTS	12
Abbildung 6: Struktur des WWW mit Java	14
Abbildung 7: Der Verifizierungsprozeß für Java Programme	19
Abbildung 8: Komponenten eines Prädikat-Transitionsnetzes	22
Abbildung 9: Ein ConTract.....	22
Abbildung 10: Die Struktur einer Sequenz	22
Abbildung 11: Die Struktur eines Blockes.....	23
Abbildung 12: Do-Nothing-Step.....	23
Abbildung 13: Step mit Konfliktbehebung	23
Abbildung 14: Der IF-Block	24
Abbildung 15: Der WHILE-Block	24
Abbildung 16: Der REPEAT-Block.....	24
Abbildung 17: Der FOR-Block.....	25
Abbildung 18: Der CASE-Block.....	25
Abbildung 19: Der Parallel-Block.....	25
Abbildung 20: Struktur der Sequenz mit transaktionalen Blöcken.....	26
Abbildung 21: Die Struktur eines TA-Blocks	26
Abbildung 22: Beispiele für Graphen	27
Abbildung 23: Die beiden Graphen des Satzes von Kuratowski	29
Abbildung 24: Der Graph eines Parallel-Blocks	33
Abbildung 25: Graph einer REPEAT-Schleife	34
Abbildung 26: Nicht transaktionaler IF-Block.....	34
Abbildung 27: Transaktionaler IF-Block	35
Abbildung 28: Einlesen eines Prädikat-Transitionsnetzes	40
Abbildung 29: Beispiel für calculate_x()	45
Abbildung 30: Beispiel für calculate_y()	47
Abbildung 31: Das Hauptfenster des Monitoring Interface	51
Abbildung 32: Zustände eines ConTracts vor und während der Bearbeitung.....	54
Abbildung 33: Synchrone Darstellung eines ConTracts	56

Tabellenverzeichnis

Tabelle 1: Unterschiede zwischen Java und C++.....	15
Tabelle 2: Typen von Transitionen	78
Tabelle 3: Typen von Steps.....	78
Tabelle 4: Typen von Prädikaten.....	79
Tabelle 5: Status einer Transaktion.....	79

1 Einleitung

1.1 Einführung

Im Bereich der Computertechnologie und speziell im Bereich der Datenbanksysteme gibt es heutzutage viele Situationen, in denen die Bearbeitung eines bestimmten Vorgangs nicht sofort abgeschlossen werden kann, weil er über einen längeren Zeitraum andauert. Diese sogenannten „langlebigen Abläufe“ erstrecken sich meist über Tage, Wochen oder sogar Monate, bevor sie ein endgültiges Ergebnis liefern. Ab und zu muß auch ein Teil des Vorgangs wieder zurückgesetzt (kompensiert) und durch andere Aktionen ersetzt werden.

Ein typisches Beispiel hierfür ist die Buchung eines Urlaubs in einem Reisebüro: Nachdem das Reiseziel feststeht, müssen in der Regel Flüge gebucht, Hotels reserviert und vielleicht noch ein Mietwagen oder ähnliches angefordert werden. Alle diese Vorgänge können parallel zueinander ausgeführt werden, jedoch dauert es meist einige Tage, bis die einzelnen Ergebnisse vorliegen. Falls die Flugbuchung fehlschlägt, die Hotelreservierung hingegen bereits erfolgt ist, muß man sich entweder nach einem anderen Flug umsehen oder aber die Hotelreservierung rückgängig machen.

Das Prinzip der Transaktionen, das besonders im Bereich der Datenbanksysteme eingesetzt wird, eignet sich nur bedingt, um solche langlebigen Abläufe zu beschreiben. Erstens wird im Falle eines Fehlers die Transaktion immer komplett zurückgesetzt, wodurch alle bisherigen Ergebnisse verloren gehen und zweitens können mehrere Transaktionen nicht auf dieselben Objekte zugreifen, da diese für die Dauer eines Zugriffs einer Transaktion gesperrt sind.

Besonders bei langlebigen Abläufen wirken sich diese Eigenschaften negativ auf die Verarbeitungsgeschwindigkeit eines Vorgangs aus und führen dazu, daß die Bearbeitung des Vorgangs länger dauert als nötig. Aus diesem Grund wurde das ConTract-Modell [ReuWä90] entwickelt, das eine Erweiterung des klassischen Transaktionsmodells darstellt. Im ConTract-Modell wird ein Vorgang in einzelne Teilschritte, sogenannte Steps, zerlegt, die unter Einhaltung des ACID-Prinzips¹ unter Transaktionsschutz ablaufen. Dadurch kann eine fehler-tolerante Ausführung gewährleistet werden und einzelne Teilschritte können durch vorher festgelegte Maßnahmen zurückgesetzt werden.

1.2 Aufgabenstellung

APRICOTS (A **P**rototype **I**mplementation of a **C**on**T**ract **S**ystem) ist ein System zur zuverlässigen Ausführung langlebiger Abläufe (Workflows) und wird zur Zeit am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart entwickelt [Schw93] [Schw95]. Das Projekt existiert bereits seit mehreren Jahren und soll nun im Rahmen von mehreren Studien- und Diplomarbeiten zu einem neuen, lauffähigen Gesamtsystem ausgebaut werden.

In diesem Zusammenhang soll als ein Teil des Projektes eine graphische Benutzeroberfläche in Java erstellt werden, die eine der Schnittstellen des Systems zum Benutzer darstellt. Der

¹ Eigenschaften einer Transaktion: **A**tomicity, **C**onsistency, **I**solation, **D**urability

Benutzer soll die Möglichkeit haben, sich den aktuellen Verarbeitungszustand eines von ihm gestarteten Prozesses (ConTracts) sowohl synchron als auch asynchron darstellen zu lassen und diesen gegebenenfalls manipulieren zu können. Synchron Darstellung bedeutet hier, daß der aktuelle Abarbeitungszustand eines ConTracts ständig überwacht und angezeigt wird, während bei der asynchronen Darstellung nur der Zustand des ConTracts angezeigt wird, wie er kurz vor dem Aufruf dieser Funktion war. Zustandsänderungen, die später eintreten, werden nicht nachgeführt, d.h. die Darstellung „veraltet“ mit fortschreitender Zeit.

Als Manipulationsmöglichkeiten sind vorgesehen, die Abarbeitung eines ConTracts anzustoßen (start), anzuhalten (suspend) oder abzubrechen (stop), den ConContract weiterlaufen zu lassen (continue bzw. resume), oder den ConContract zurückzusetzen (compensate).

1.3 Gliederung der Arbeit

Kapitel 2 gibt einen Überblick über APRICOTS, CORBA und den Naming Service. Kapitel 3 beschäftigt sich mit der dem Monitoring Interface zugrunde liegenden Programmiersprache und analysiert ihre Stärken und Schwächen. In Kapitel 4 wird erklärt, wie ein Prädikat-Transitionsnetz aufgebaut ist, welches ein ConContract-Skript beschreibt. Kapitel 5 beschäftigt sich mit den graphentheoretischen Grundlagen, die zur graphischen Darstellung eines ConContract-Skriptes notwendig sind. Außerdem werden die Planaritätseigenschaften der einzelnen Elemente eines Prädikat-Transitionsnetz untersucht. Das sechste Kapitel zeigt, wie ein ConContract-Skript eingelesen und bearbeitet wird. Insbesondere der verwendete Algorithmus zur Einbettung des Skriptes in die Ebene wird näher beschrieben. Im siebten Kapitel wird auf die Oberflächengestaltung und das Benutzungskonzept des Monitoring Interface eingegangen. Abschließend liefert Kapitel 8 eine Zusammenfassung über die vorliegende Arbeit und gibt einen kleinen Ausblick, was in Zukunft noch getan bzw. verbessert werden könnte.

2 Überblick

2.1 ConTracts

Der Begriff ConTract steht für „**C**ontroled Activities **C**ontroled **T**ransaction Compound“ und beschreibt ein Modell zur zuverlässigen Ausführung langlebiger Abläufe auf der Basis von ACID-Transaktionen. Das ACID-Prinzip gewährleistet hierbei die Einhaltung der Konsistenz der Daten durch die Eigenschaften Atomicity, Consistency, Isolation und Durability.

Für einen ConTract ist garantiert, daß seine Ausführung entweder in einem endlichen Zeitraum erfolgreich terminiert oder daß der Originalzustand oder ein dazu logisch äquivalenter Zustand wiederhergestellt wird[Schw93]. Dadurch wird sichergestellt, daß ein Vorgang - bzw. Teile desselben - mit minimalem Zeitaufwand zurückgesetzt (kompensiert) werden bzw. anschließend weiter ausgeführt werden kann. Im ConTract-Modell gliedert sich hierzu ein ConTract in zwei Teile, nämlich in das Skript und die Steps.

2.1.1 Das ConTract-Skript

Im ConTract-Skript wird der Gesamtablauf durch die Festlegung des Kontroll- und Datenflusses definiert. Hierzu stehen verschiedene blockorientierte Konstrukte wie zum Beispiel Verzweigungen, Sequenzen, Schleifen und Anweisungen zur Verfügung, wie sie auch aus den herkömmlichen Programmiersprachen bekannt sind. Auch zur parallelen Ausführung von Steps gibt es spezielle Konstrukte wie zum Beispiel die Parforeach-Anweisung, wodurch eine Anweisung mehrfach parallel auf beliebige Steps angewendet werden kann. Diese Anweisung wird allerdings in der vorliegenden Implementierung von APRICOTS noch nicht unterstützt, da sie sehr aufwendig zu programmieren ist.

Auch welche Aktionen unter Transaktionschutz ablaufen sollen, wird im Skript definiert. Dies ist eine wichtige Maßnahme zur Wahrung der Daten-Konsistenz, da die einzelnen Steps auf gemeinsam genutzte Daten zugreifen können. Transaktionen können übrigens beliebig tief geschachtelt werden, wobei unter einer Top-Level-Transaktion mehrere Sub-Transaktionen ablaufen können.

Invarianten dienen dazu, den Ablauf mehrerer paralleler ConTracts zu synchronisieren. Mit ihrer Hilfe können globale Daten vor dem gleichzeitigen Zugriff mehrerer ConTracts geschützt werden, so daß die Datenintegrität gewahrt bleibt. Durch die Angabe von Konflikt-Lösungen (conflict resolutions) kann vordefiniert werden, wie im Falle eines Konflikts weiter vorgegangen wird.

2.1.2 Die ConTract-Steps

Steps sind die elementaren Aktionen in einem Skript und stehen vollständig unter Transaktionsschutz. Ihre Programmierung erfolgt getrennt von der Skripterstellung und kann in jeder beliebigen Programmiersprache durchgeführt werden. Ein Step wird immer Schritt für

Schritt ausgeführt und die anfallenden Daten werden in einem sogenannten Kontext [BuRai90] gespeichert. Für jeden Step gibt es in der Regel Kompensationssteps, die im Falle einer Kompensation die Aktionen der bereits ausgeführten Steps rückgängig machen. Dies muß bereits im ConTract-Skript definiert werden.

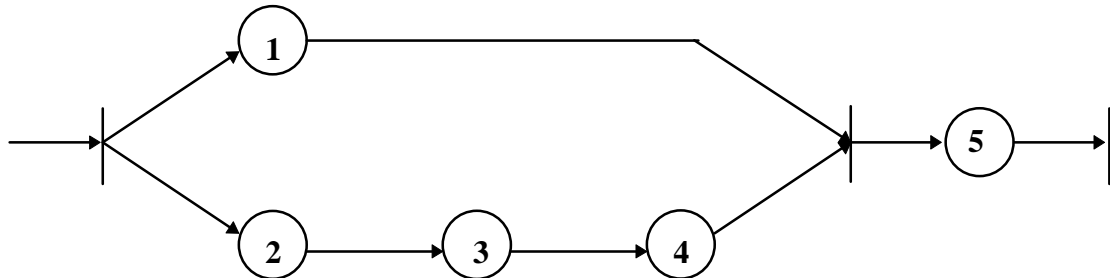


Abbildung 1: Beispielskript mit fünf Steps

Abbildung 1 zeigt ein Beispielskript mit fünf Steps. Die Steps 1 und 2 können parallel zueinander ausgeführt werden, wohingegen die Steps 2, 3 und 4 hintereinander abgearbeitet werden müssen. Erst wenn die Steps 1 und 4 beendet sind, kann der letzte Step 5 ausgeführt werden.

2.2 APRICOTS

APRICOTS steht für „**A** **P**rototype **I**mplementation of a **C**on**T**ract **S**ystem“ und wurde als Beispielanwendung für ein Workflow-System auf der Basis des ConTract-Modells am Institut für Parallele und Verteilte Höchstleistungsrechner der Universität Stuttgart entwickelt. Das System besteht aus mehreren einzelnen Komponenten, die mehr oder weniger unabhängig voneinander entwickelt wurden (und werden) und die über definierte Schnittstellen und ein zugrunde liegendes Kommunikationssystem Daten miteinander austauschen.

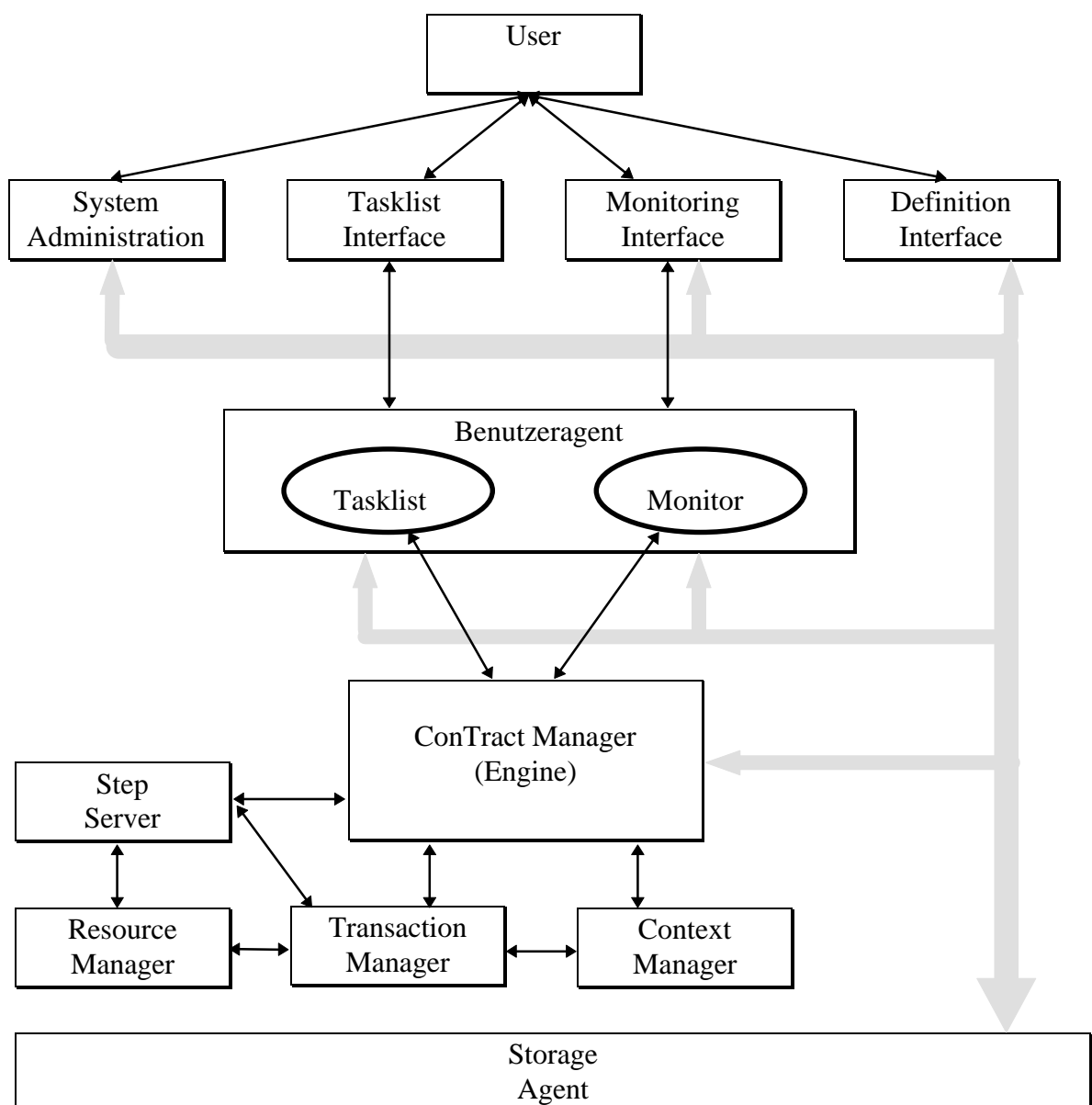


Abbildung 2: Der Aufbau von APRICOTS

2.2.1 Das System Administration Interface

Das System Administration Interface bietet Funktionen zur Verwaltung des Systems und seiner Benutzer. Es stellt alle für diesen Zweck relevanten Informationen graphisch dar und bietet dem Benutzer die Möglichkeit, aktiv in das System einzugreifen. So können zum Beispiel neue Komponenten in das System eingebunden werden oder der Benutzer kann im Falle eines Fehlers Maßnahmen ergreifen, um diesen zu beheben.

2.2.2 Das Definition Interface

Mit Hilfe des Definition Interface werden die ConTract-Skripte in graphischer Form in das System eingegeben. Aus diesen sogenannten Templates werden dann von einem Compiler auf Anforderung beliebig viele Instanzen erzeugt, die der ConTract Manager bearbeiten kann. Eine Instanz ist in APRICOTS übrigens nichts anderes als ein Prädikat-Transitionsnetz, das vom ConTract Manager (s.u.) interpretiert wird. Prädikat-Transitionsnetze werden in einem späteren Kapitel noch ausführlich beschrieben.

2.2.3 Das Monitoring Interface

Das Monitoring Interface ist Gegenstand dieser Diplomarbeit und wird in einem späteren Kapitel noch ausführlich beschrieben. Es stellt eine der Schnittstellen des Systems zum Benutzer dar und gibt Auskunft über den aktuellen Bearbeitungszustand eines ConTracts. Außerdem kann der Benutzer mit Hilfe des Monitoring Interface alle von ihm gestarteten ConTracts manipulieren.

2.2.4 Das Tasklist Interface

Das Tasklist Interface ist für die Verwaltung der Aufgaben zuständig, die der Benutzer im Rahmen der Abarbeitung eines ConTracts durchführen muß. Es bietet dem Benutzer eine Übersicht über alle bisher angefallenen Aufgaben und startet die zur Bearbeitung einer Aufgabe benötigte(n) Applikation(en), wenn der Benutzer eine Aufgabe auswählt. Dabei ist zu beachten, daß das Tasklist Interface nicht direkt mit der Engine kommuniziert, sondern wie das Monitoring Interface auch mit dem Benutzer Agenten, der zwischen der Engine und dem Tasklist Interface zwischengeschaltet ist.

2.2.5 Der ConTract Manager

Der ConTract Manager ist das Kernstück des APRICOTS - Systems, da er die zur Abarbeitung der ConTract-Skripte benötigten ConTract-Engines zur Verfügung stellt. Bevor ein Skript abgearbeitet werden kann, muß für dieses nämlich zuerst eine Engine erzeugt werden, die dann die zur Abarbeitung notwendigen Stepserver startet und insbesondere über eine zuverlässige Ausführung wacht. Nach Beendigung eines Skripts können dann die jeweiligen Engines wieder gelöscht werden.

2.2.6 Die Stepserver

Stepserver sind vom Benutzer in einer beliebigen Programmiersprache geschriebene Programme, die die Funktionalität der Steps eines Skriptes nachbilden. Der Step-Code wird hierbei rein sequentiell abgearbeitet, da jeder einzelne Step unter Transaktionsschutz steht. Trifft die Engine bei der Abarbeitung eines Skriptes auf einen Step, so übermittelt sie dem dazugehörigen Stepserver die benötigten Eingabdaten und wartet, bis der Stepserver die berechneten Ausgabedaten zurückliefert.

2.2.7 Der Transaktions Manager

Der Transaktions Manager ist für die Verwaltung der Transaktionen auf der Basis von OTS¹ zuständig. Er überwacht die Ressourcenintegration und das korrekte Terminieren einer Transaktion mit Hilfe des 2-Phasen-Commit-Protokolls. Durch den Einsatz von OTS sind auch geschachtelte Transaktionen möglich, wie sie im ConTract-Modell vorgesehen sind.

2.2.8 Der Context Manager

Mit Hilfe des Context Managers werden die bei der Abarbeitung eines Skriptes von den Stepservern erzeugten Ein- und Ausgabedaten verwaltet und für die gesamte Dauer der Abarbeitung persistent gespeichert. Dadurch kann zu jedem beliebigen Zeitpunkt auf einmal gespeicherte Daten zugegriffen werden, was besonders für die Kompensation eines ConTracts wichtig ist.

2.2.9 Der Ressource Manager

Der Ressource Manager verwaltet, wie der Name schon vermuten läßt, die zur Abarbeitung eines ConTracts benötigten Ressourcen, indem er Schnittstellen zu den einzelnen Ressourcen-Objekten anbietet. So können zum Beispiel die einzelnen Stepserver über den Ressource Manager und seine Funktionen auf eine Datenbank zugreifen, um sich dort benötigte Informationen zu beschaffen.

2.2.10 Der Storage Agent

Der Storage Agent ist eine Komponente, die ausschließlich für die persistente Speicherung der von den anderen Komponenten des Systems erzeugten Daten zuständig ist. Er bietet dazu eine Schnittstelle mit Input-/Output-Funktionen an, die von allen anderen Komponenten bequem und ohne Kenntnis der eigentlichen Implementierung genutzt werden kann. So speichert zum Beispiel das Definition Interface mit Hilfe des Storage Agents das durch den Benutzer eingegebene ConTract-Skript ab, oder das Monitoring Interface kann über den Storage Agent das vom Compiler aus diesem Skript erzeugte Prädikat-Transitionsnetz einlesen.

¹ OTS: Object Transaction Service von der Object Management Group

2.2.11 Der Benutzer Agent

Der Benutzer Agent besteht eigentlich aus zwei Komponenten und hat die Aufgabe, den Benutzer im System zu vertreten, da dieser sich nicht die ganze Zeit im System befinden kann.

- Die Komponente Tasklist ist eine Schnittstelle zwischen der Engine und dem Tasklist Interface und puffert im Prinzip die Aufgaben von der Engine, die eigentlich für den Benutzer gedacht waren. Da dieser aber nicht die ganze Zeit anwesend sein kann, werden alle Daten zwischengespeichert und erst an das Tasklist Interface ausgeliefert, wenn der Benutzer im System ist.
- Die Komponente Monitor ist eine Schnittstelle zwischen der Engine und dem Monitoring Interface und fängt zum Beispiel alle Meldungen ab, die von der Engine generiert werden und an den Benutzer gerichtet sind. Dies hat den Vorteil, daß die Engine nach einer Bestätigungsmeldung durch die Komponente Monitor sofort weiterarbeiten kann und nicht durch einen Benutzer blockiert wird, der zur Zeit gar nicht anwesend ist und daher keine Meldungen entgegennehmen kann.

2.3 CORBA

CORBA ist ein Standard der Object Management Group (OMG) und steht für **Common Object Request Broker Architecture**. Der Standard wurde 1991 von dem mehr als 500 Mitgliedern umfassenden Software-Konsortium ins Leben gerufen, um endlich eine geeignete Infrastruktur für verteilte objektorientierte Programme auf heterogenen Netzwerken zur Verfügung zu stellen. Ziel war es, nicht nur die transparente Verteilung von CORBA-Objekten über das zugrundeliegende Netzwerk zu erreichen, sondern auch den Zugriff auf jedes beliebige CORBA-Objekt zu ermöglichen, unabhängig davon, in welcher Programmiersprache es geschrieben wurde.

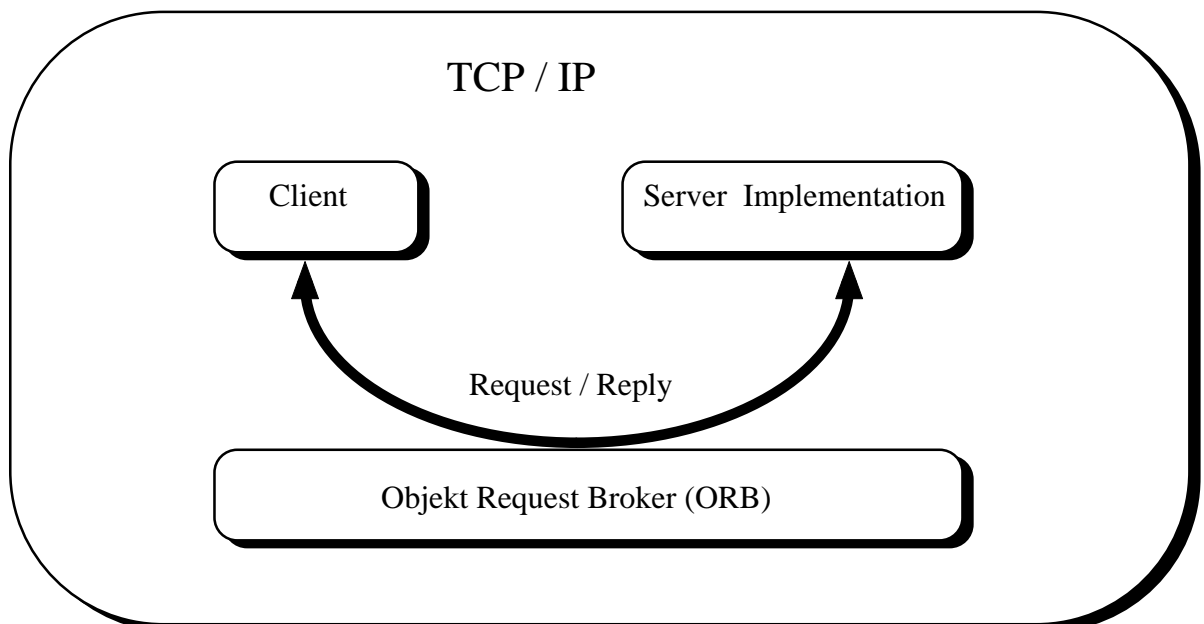


Abbildung 3: Kommunikation mit CORBA

2.3.1 Wie funktioniert CORBA ?

Zentraler Bestandteil der Architektur ist der Object Request Broker, kurz ORB. Seine Aufgabe ist es, Methodenaufrufe von einer Client-Applikation zu einem entfernten Objekt zu übermitteln, sowie die Ergebnisse und eventuelle Fehler rückzumelden. Jedes CORBA-Objekt ist Instanz eines sogenannten Interface-Typs. Ein Interface-Typ repräsentiert eine Klasse mit möglicherweise mehreren Elternklassen. Damit eine Applikation auf ein CORBA-Objekt zugreifen kann, muß die Interoperabilität zwischen dem aufrufenden und dem aufgerufenen Objekt gewährleistet sein. Zu diesem Zweck existiert in CORBA eine Schnittstellenbeschreibungssprache, kurz IDL (Interface Definition Language). Diese Schnittstellenbeschrei-

bungssprache enthält Elemente zur Datenbeschreibung und ist an die Syntax von C++ angelehnt. Soll im Netz ein CORBA-Objekt zur Verfügung gestellt werden, muß zuvor eine IDL-Definition im Interface Repository abgelegt werden, aus dem dann ein IDL-Compiler Code-Sequenzen (Stubs) und eine Art Skelett für die Implementierung (Skeleton) erzeugt. Die Informationen zur Implementierung werden zum Zeitpunkt der Installation im Implementation Repository abgelegt. Die Abbildung 4 verdeutlicht diesen Sachverhalt.

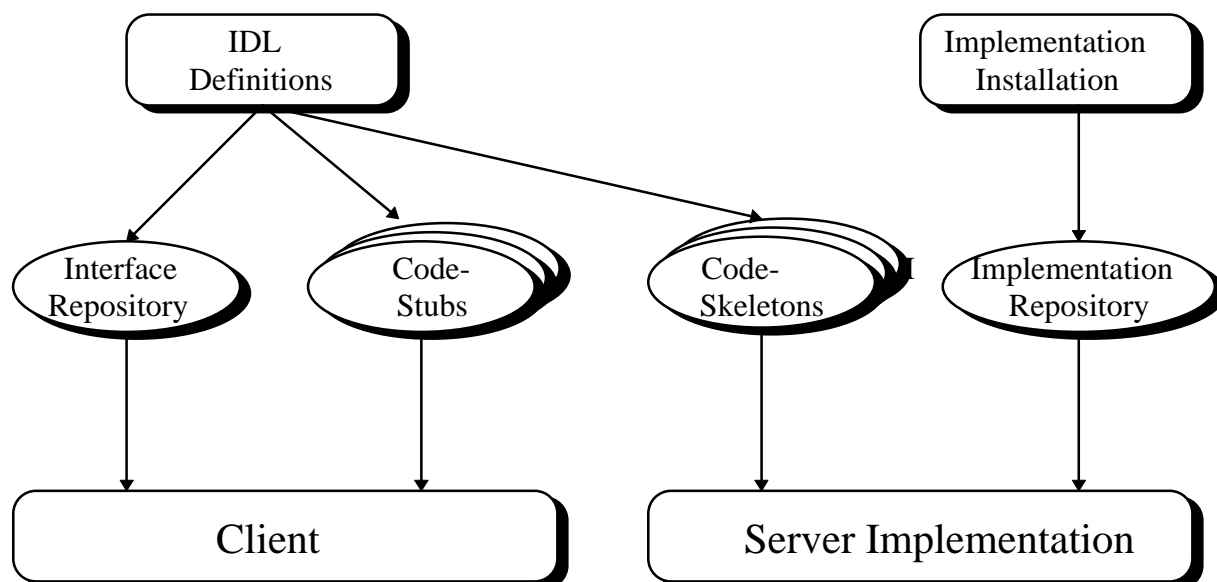


Abbildung 4: Interface- und Implementation Repository

Der IDL-Compiler bildet die IDL-Definition(en) auf den Code einer beliebigen Programmiersprache ab, zum Beispiel Java, Ada, C++ oder Smalltalk. Die generierten Code-Sequenzen (Stubs) bindet der Benutzer in sein eigenes Programm ein und kann dadurch transparent auf entfernte CORBA-Objekte zugreifen. Jedes CORBA-Objekt ist dabei durch eine eindeutige und typlose Objektreferenz bestimmt.

Damit man problemlos CORBA-Objekte benutzen kann, erzeugt der IDL-Compiler sogenannte Stellvertreterobjekte (Proxies), die lokal die Rolle der entfernten CORBA-Objekte übernehmen. Sobald eine Proxy-Methode aufgerufen wird, verpackt das Stellvertreterobjekt die übergebenen Argumente und schickt diese über das zugrundeliegende Netzwerk an das eigentliche CORBA-Objekt bzw. meldet eventuelle Ergebnisse zurück. Die Kommunikation erfolgt in der Regel auf der Basis von TCP/IP¹, da dieses als einziges im CORBA Standard 2.0 von 1994 genau spezifiziert ist.

¹ Transmission Control Protocol / Internet Protocol: Standard für den Austausch von Daten über das Internet

2.3.2 Warum CORBA ?

Für die Entwicklung des APRICOTS-Systems wurde die CORBA Implementierung Orbix von der irischen Firma IONA Technologies eingesetzt. Speziell für Java stand das Produkt OrbixWeb in der Version 1.0 und später dann in der Version 2.0 von derselben Firma zur Verfügung. Die Entscheidung für CORBA und OrbixWeb fiel aus folgenden Gründen:

- Da das APRICOTS-System aus mehreren Komponenten besteht, die in einer verteilten Umgebung laufen und die mit Hilfe von verschiedenen objektorientierten Programmiersprachen entwickelt wurden, bot sich eine Verwendung von CORBA geradezu an, da CORBA speziell für einen Einsatz in einer solchen Umgebung entwickelt wurde.
- Mit Hilfe von CORBA lassen sich alle Komponenten (theoretisch) problemlos zu einem einheitlichen Gesamtsystem vereinigen.
- Sprachanpassungen für die Sprachen Java und C++, die speziell im APRICOTS Projekt eingesetzt werden, waren für Orbix bereits vorhanden.
- Durch die Gemeinsamkeiten der Objektmodelle von CORBA und Java, beispielsweise dem Interface-Konzept, konnte das Objektmodell von CORBA fast nahtlos in Java umgesetzt werden, was die Programmierung wesentlich erleichterte.
- Die C++-Version von Orbix stand in einer multithread-fähigen Version zur Verfügung, was besonders für die Programmierung des ConTract Managers wichtig war.
- CORBA-Services wie zum Beispiel der Naming Service sind in Orbix bereits implementiert (siehe auch nächster Abschnitt).

2.3.3 Der Naming Service von CORBA

CORBA bietet neben den allgemein verfügbaren Funktionen des Object Request Brokers noch weitere zusätzliche Services an, zu denen unter anderem auch der Naming Service gehört. Mit Hilfe des Naming Services kann Objekten ein fast beliebiger logischer Name zugewiesen werden, der unabhängig vom physikalischen Standort des Objektes ist. Dadurch lassen sich die CORBA-Objekte im Netzwerk einfacher lokalisieren und ansprechen.

Durch die beliebige Vergabe von logischen Namen ist es möglich, einen hierarchischen Namensraum in der Form eines Baumes aufzubauen, in dem die verschiedenen Komponenten des APRICOTS-Systems, wie zum Beispiel Benutzer Agenten, Stepserver o.ä. ohne Kenntnis der physikalischen Adresse leicht aufgefunden werden können. Abbildung 5 zeigt den Aufbau des Namensraum von APRICOTS zum jetzigen Stand des Projekts. Später werden sich allerdings wahrscheinlich noch geringfügige Änderungen ergeben.

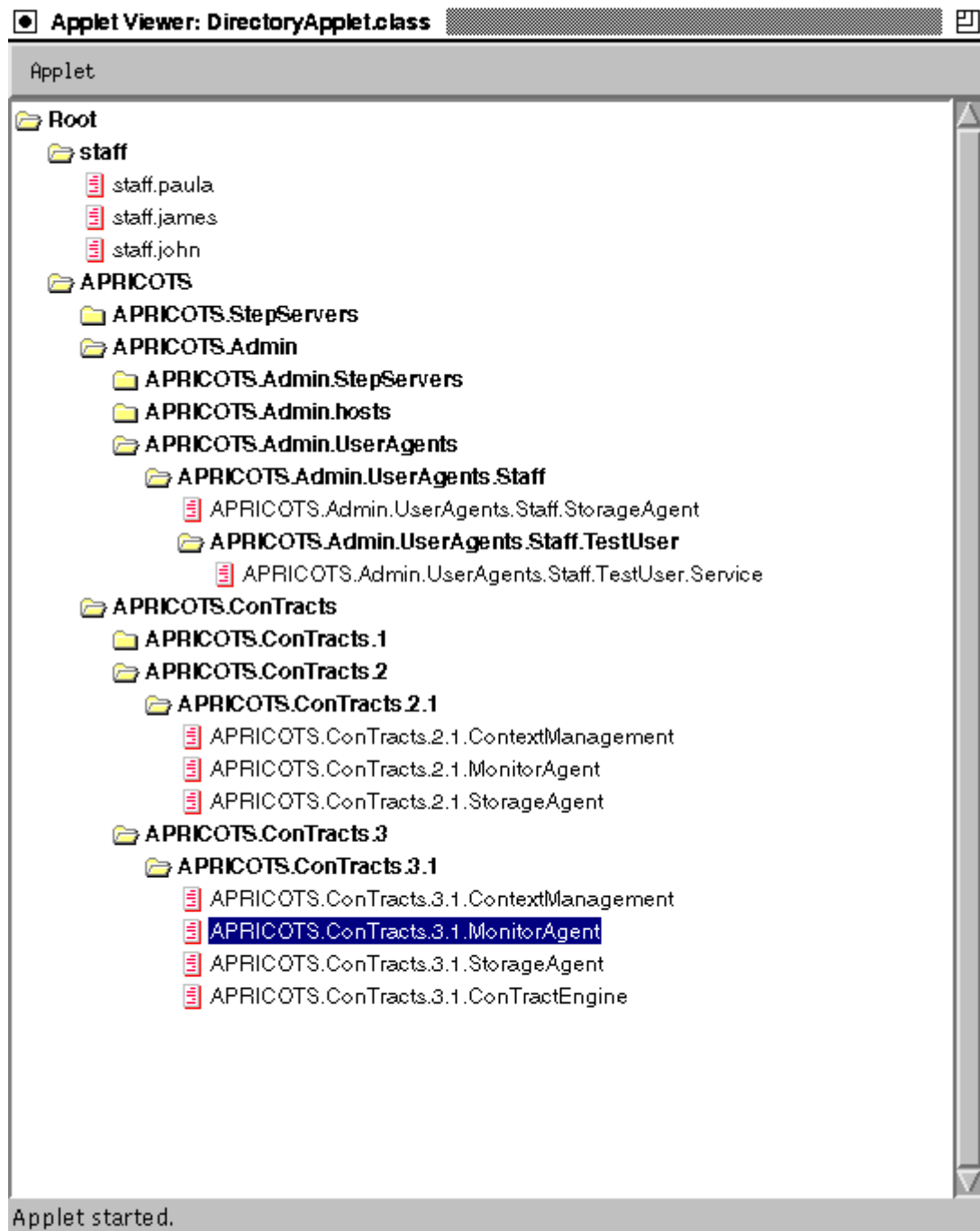


Abbildung 5: Der Namensraum von APRICOTS

Wie aus Abbildung 5 leicht zu erkennen ist, sind die Namen in APRICOTS hierarchisch in einer Baumstruktur angeordnet. Den Ausgangspunkt für das Monitoring Interface bildet das Verzeichnis APRICOTS, das unterhalb des Root-Directories angeordnet ist. Das Verzeichnis APRICOTS besteht aus weiteren Unterverzeichnissen, über die die relevanten Informationen gefunden werden können. So gibt es zum Beispiel ein Verzeichnis APRICOTS.ConTracts, unter dem sich alle ConTract-Templates befinden, unter denen sich wiederum die jeweiligen Instanzen befinden.

3 Java

Der Code für diese Diplomarbeit wurde mit dem JDK 1.0.2 von Sun vollständig in Java geschrieben. Genausogut hätte man aber auch C, C++, Smalltalk oder jede andere beliebige Programmiersprache verwenden können. Warum gerade Java zum Einsatz kam, und welche Vor- und Nachteile sich daraus ergaben, soll in diesem Kapitel diskutiert werden.

3.1 Was ist Java ?

Java ist eine noch relativ neue, objektorientierte Programmiersprache ähnlich C++ und wurde von der Firma Sun Microsystems entwickelt. Schwerpunkt der Sprache ist die Schaffung von ausführbaren Inhalten, die über Netzwerke und speziell über das Internet verteilt werden können und dann lokal unabhängig von der verwendeten Computerplattform ablaufen. Die Firma Sun spricht in diesem Zusammenhang sogar von „Write Once - Run Everywhere“, also von Applikationen, die einmal entwickelt auf beliebigen Plattformen ohne Portierung ablaufen können.

Durch Java soll dem World Wide Web die fehlende Interaktivität gegeben werden, die bisher trotz Skript-Programmierung und CGI¹-Gateways im Vergleich zu anderen nicht vernetzten multimedialen und hypermedialen Systemen nur in sehr geringem Maße vorhanden war. Java soll im WWW bisher nie dagewesene Spielräume für die Interaktivität freisetzen, denn durch den Einsatz von Java kann jetzt auch in verteilten Umgebungen all das unterstützt werden, was bisher nur auf lokalen Systemen möglich war; Animationen, Tutorien, interaktive Spiele, Berechnungen in Echtzeit - all das ist mit Java jetzt möglich.

3.2 Wie funktioniert Java ?

Java Applikationen können als eigenständige Anwendungen oder als sogenannte Applets entwickelt werden. Ein Applet ist ein Stück Code, das mit Hilfe von speziellen Tags in eine HTML²-Seite eingebettet wird und dann mit Hilfe eines Java fähigen Browsers³ ausgeführt werden kann.

Java Programme werden von einem Compiler nicht in Maschinensprache übersetzt, sondern in einen unabhängigen und portablen Zwischencode. Dieser sogenannte Bytecode kann theoretisch auf jedem beliebigen Zielsystem interpretiert werden, Voraussetzung ist lediglich die Verfügbarkeit einer systemspezifischen Laufzeitumgebung. Die Opcodes im Bytecode sind immer ein Byte lang (daher der Name Bytecode), gefolgt von einer variablen Anzahl von Operatoren und Daten. Für die Ausführung der Bytecodes ist die sogenannte Virtual Machine zuständig, die auf 32-Bit Basis und stackbasiert arbeitet, d.h. alle Parameter werden immer über den Stack übergeben. Die Register des jeweiligen Prozessors werden dafür nicht genutzt, weil die verschiedenen zu unterstützenden Prozessorarchitekturen in diesem Punkt zu weit voneinander abweichen.

¹ CGI: Common Gateway Interface, Schnittstelle zur Ausführung von Skripten

² HTML: HyperText Markup Language: Seitenbeschreibungssprache für das World Wide Web

³ Browser: Programm zur Darstellung von WWW-Seiten

Interessant ist, das die Java Virtual Machine bereits auf unterster Ebene den Unicode-Zeichensatz unterstützt, damit alle nationalen und internationalen Zeichensätze dargestellt werden können. Dadurch läßt sich Java zwar auf vielen unterschiedlichen Maschinen in unterschiedlichen Kulturkreisen einsetzen, die Ausführungsgeschwindigkeit der Programme leidet jedoch darunter, da der Unicode zwei Bytes anstatt wie bisher nur ein Byte an Speicher benötigt.

Der Performance-Verlust, der natürlich auch daher rührt, daß Java eine interpretierte Sprache ist, wird sich in Zukunft durch den Einsatz von sogenannten Just-In-Time-Compilern ausgleichen lassen. Just-In-Time-Compiler wandeln den Bytecode zur Laufzeit durch die Virtual Machine erst wieder in direkten plattformspezifischen Maschinencode um, bevor dieser ausgeführt wird. Geschwindigkeitseinbußen gegenüber compiliertem Code, wie er etwa von einem C++ Compiler erzeugt wird, werden dann kaum noch zu spüren sein.

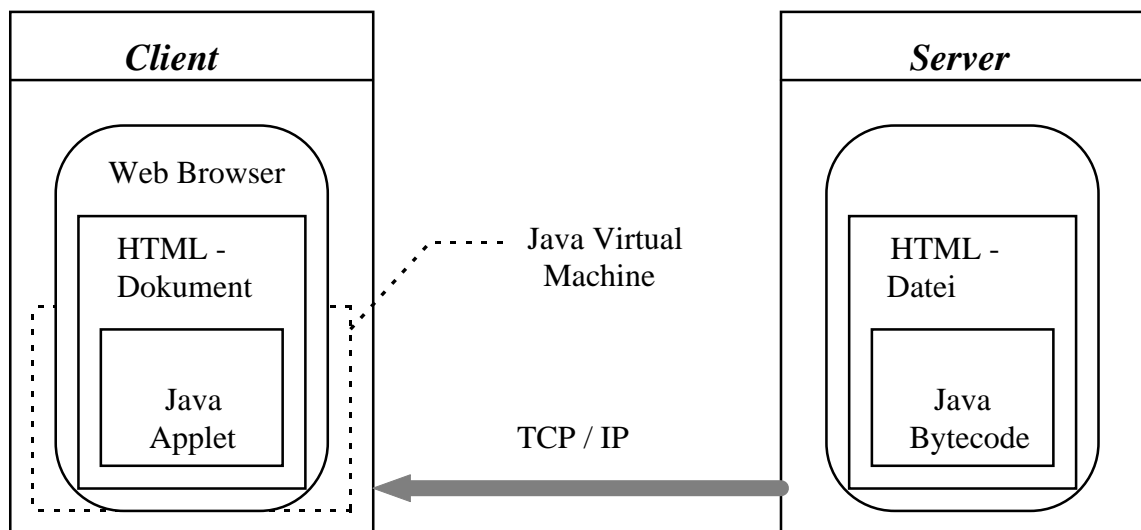


Abbildung 6: Struktur des WWW mit Java

Abbildung 6 verdeutlicht den Mechanismus einer verteilten Anwendung mit Java: Der Java Bytecode, der auf dem Server mit Hilfe von speziellen Tags in eine HTML-Seite eingebettet ist, wird via TCP/IP an den Client übertragen, der dann in seinem Java fähigen Web Browser das HTML-Dokument darstellt, während im Hintergrund die Java Virtual Machine den Bytecode interpretiert und das Applet ausführt.

3.3 Die Vor- und Nachteile des Java Einsatzes

3.3.1 Die Eigenschaften von Java

Java ist stark an die Programmiersprache C++ angelehnt, jedoch wurden nur die „guten“ Eigenschaften dieser Sprache übernommen. Zugunsten von Einfachheit, Robustheit und (Fehler-) Sicherheit wurden einige Funktionen weggelassen, die man aber durchaus verschmerzen kann. Insbesondere wurden in Java nicht implementiert:

- Zeiger
- Strukturen
- Verbunde
- Headerdateien
- Preprozessor Anweisungen
- Überladen von Operatoren
- Templates
- Implizite Konvertierung von Typen
- Mehrfachvererbung
- usw.

Feature	C / C++	Java
Arrays	Zeiger	echte Arrays mit Boundcheck
Boolean	muß definiert werden	ist vordefiniert
Kontrollfluß	if, while, do, for ergeben int	muß boolscher Wert sein
Sprünge	goto beliebiges Label	keine Sprünge erlaubt
Kommandozeile	argv[0] ist Programmname	argv[0] ist erstes Argument
Typ char	1 Byte	16-Bit Unicode
char-Arrays	Array von char zu 8 Bit	String- bzw. StringBuffer-Objekte
Einbinden	#include <Datei>	import <interface>
NULL	muß definiert werden	null ist Schlüsselwort
Speicherverwaltung	explizite Anforderung und Freigabe	Speicher muß nicht angefordert werden, Garbage Collection
Thread-Support	fehlt als Sprachelement	wird unterstützt
Mehrfachvererbung	wird unterstützt	wird nicht unterstützt

Tabelle 1: Unterschiede zwischen Java und C++

Tabelle 1 gibt einen Überblick über die Unterschiede von Java und C++. Wie man sieht, wurden einige sehr mächtige, aber auch sehr fehlerträchtige Konstrukte weggelassen, wie zum Beispiel das Zeiger-Konzept.

3.3.2 Die Vorteile von Java

Java bietet eine Menge Vorteile gegenüber C++ und anderen herkömmlichen Sprachen, die im folgenden erörtert werden:

Java ist

- *einfach*: Java baut auf der Programmiersprache C++ auf, ohne jedoch einige der selten genutzten und fehlerträchtigen Merkmale dieser Sprache zu implementieren. Viele dieser Funktionen sind zwar recht nützlich, verursachen aber bei unsauberer Anwendung große Probleme, die in Java von vornherein vermieden werden sollten; so zum Beispiel die Verwendung von Zeigern. Programme, in denen häufig auf die Benutzung von Zeigern zurückgegriffen wird (und das ist

bei C++ zum Beispiel fast immer der Fall), sind sehr fehleranfällig und außerdem schwer zu lesen. Da der größere Teil der Kosten bei der Softwareentwicklung auf den Unterhalt des Codes und nicht auf seine Erstellung entfällt, kann der Verzicht zugunsten eines robusteren und besser verstehbaren Codes zu deutlich niedrigeren Softwarekosten und meistens auch zu kürzeren Entwicklungszeiten führen.

- *verteilt:* Im Gegensatz zu anderen Sprachen ist Java hauptsächlich dafür konstruiert worden, innerhalb einer vernetzten Umgebung zu funktionieren. Java verfügt über eine riesige Bibliothek von Klassen zur Kommunikation mittels des TCP/IP-Protokolls, zu dem auch Protokolle wie HTTP¹ oder FTP² gehören. Der Java Code kann Ressourcen über URLs³ ebenso leicht manipulieren, wie es Programmierer gewöhnt sind, auf ein lokales Dateisystem mit C oder C++ zuzugreifen.
- *robust:* Java schränkt den Programmierer beim Schreiben des Quellcodes zwar stärker ein, fördert dadurch aber die Programmierung einer robusteren, das heißt weniger fehleranfälligen Software. So kennt Java zum Beispiel keine implizite Typkonvertierung oder führt zum Beispiel automatisch einen Bounds-Check bei der Verwendung von Arrays durch.
- *sicher:* Da Java in vernetzten Umgebungen arbeitet und Java Applets meist über das Internet aus mehr oder weniger bekannten oder unbekanntem Quellen heruntergeladen werden, wurde bei der Entwicklung von Java besonderen Wert auf Sicherheit gelegt. So kennt Java zum Beispiel keine Zeiger, um zu verhindern, daß findige Programmierer mit einigen Tricks und ein bißchen Zeigerakrobatik auf geschützte Speicherbereiche zugreifen oder den Hauptspeicher überschreiben können. Außerdem wird bereits mit Nachdruck an der Integration von öffentlichen Verschlüsselungsmethoden in Java gearbeitet. Mehr zum Thema Sicherheit folgt in einem der nächsten Abschnitte.
- *architektur-neutral:* Da Java Programme in einen Bytecode umgewandelt werden, dessen Interpretation unabhängig von der zugrundeliegenden Plattform ablaufen kann, können sie quasi auf jedem Rechnersystem ausgeführt werden, welches eine Java Unterstützung bietet. Sie müssen nicht extra neu kompiliert werden.
- *multithreaded:* Java ist eine Sprache, die von vorneherein die quasi parallele Ausführung mehrerer Teilprozesse ermöglicht. Auf der Grundlage eines Systems von Routinen, die mehrere sogenannte Threads einkalkulieren, die auf dem Überwachungs- und Bedingungsparadigma von C.A.Hoare beruhen, bietet Java dem Programmierer einen Weg, um in Programmen ein interaktives Echtzeit-Verhalten zu unterstützen. Insbesondere bietet Java benutzbare Synchronisationseigenschaften, die das Programmieren einfacher gestalten.

¹ HTTP: **H**yper**T**ext **T**ransport **P**rotocol

² FTP: **F**ile **T**ransfer **P**rotocol

³ URL: **U**niform **R**essource **L**ocator

- *interpretierbar:* Da der Java Compiler eine Klassendatei von Java in Bytecode übersetzt, kann diese auf jedem Rechner, der eine Java Virtual Machine zur Verfügung stellt, ablaufen. Dadurch kann der Java Sourcecode plattformneutral geschrieben werden, wobei die Kompilier- und Testphasen entfallen, da der Bytecode für einen vorgegeben Computer nicht spezifiziert, sondern lediglich interpretiert wird.
- *übertragbar:* Die Neutralität der Java-Architektur ermöglicht bereits ein hohes Maß an Übertragbarkeit. Java Programme können theoretisch für eine beliebige Plattform geschrieben werden und laufen dann ohne Änderung auch auf anderen Plattformen. Dies ist bei anderen Programmiersprachen meist nicht der Fall. Bei C und C++ kann der Quellcode auf unterschiedlichen Hardwareplattformen aufgrund der Art und Weise, wie diese Plattformen arithmetische Operationen implementieren, jeweils etwas anders ablaufen. Bei Java ist das anders. Ein ganzzahliger Typ in Java, ein int, ist immer eine ganze 32-Bit-Zahl. Eine Gleitkommazahl, ein float, ist immer ein 32-Bit langer Fließkommawert, wie er im IEEE-754-Standard definiert ist.
- *dynamisch:* Im Gegensatz zum C++-Code, der häufig eine vollständige Neukompilierung erfordert, sobald zum Beispiel eine Elternklasse verändert wird, verwendet Java zur Verringerung der Abhängigkeit ein System von Schnittstellen. Daher können Java Programme neue Methoden und Variablen in eine Objektbibliothek einbeziehen, ohne daß die davon abhängigen Client Objekte betroffen sind.

Die oben genannten Vorteile waren bei der Entscheidung für Java als Programmiersprache für diese Diplomarbeit nicht der alleinige Grund. Da Java eine noch relativ junge Sprache ist, sollte bewußt untersucht werden, was mit dieser Sprache möglich ist und was nicht und wo vielleicht noch Schwächen liegen. Auf die gefundenen Fehler wird detailliert im Anhang dieser Diplomarbeit eingegangen.

3.3.3 Die Nachteile von Java

Java hat neben seinen vielen Vorteilen leider auch einige gravierende Nachteile, die allerdings teilweise darauf zurückzuführen sind, daß Java noch eine sehr junge Programmiersprache ist, deren Entwicklung noch nicht abgeschlossen ist.

Einige der gravierendsten Nachteile sind:

- Für Java sind zur Zeit noch kaum wirklich ausgereifte Entwicklungswerkzeuge erhältlich. Zahlreiche Hersteller haben zwar Produkte angekündigt, doch diese befanden sich zu Beginn dieser Diplomarbeit noch in einem Beta-Stadium und waren daher nur beschränkt einsatzfähig bzw. recht fehleranfällig.

So wurde zum Beispiel zu Beginn dieser Diplomarbeit versucht, den Java WorkShop von Sun Microsystems, der in einer Betaversion Dev 5 kostenlos erhältlich war, unter einer

Windows 95 Umgebung als Entwicklungswerkzeug einzusetzen. Dieses Vorhaben mußte jedoch schnell wieder aufgegeben werden, da der WorkShop besonders unter Windows 95 noch massiv mit Fehlern behaftet war und außerdem auch auf einem Pentium System mit 32 MB Hauptspeicher unendlich langsam arbeitete.

- Java selbst ist zum Teil noch mit kleineren Fehlern behaftet, die oft nur sehr schwer zu entdecken sind und sich auch nur unter bestimmten Bedingungen bemerkbar machen. Und gerade die Behebung von Fehlern, deren Auftreten nicht reproduzierbar ist, machen dem Programmierer den meisten Aufwand.
- Java ist noch sehr langsam. Da der Javacode bzw. der vom Compiler erzeugte Bytecode interpretiert wird, haben Java Programme naturgemäß eine deutlich schlechteres Laufzeitverhalten als andere Hochsprachen, die von einem Compiler in reinen plattformspezifischen Maschinencode umgewandelt werden. Auch der Einsatz von Just-In-Time-Compilern, die den Bytecode zur Laufzeit in Maschinencode umsetzen, haben bis jetzt noch keine rechte Abhilfe gebracht. Berichten zufolge ist die Laufzeit mancher Java-Programme beim Einsatz eines Just-In-Time-Compilers sogar schlechter (!) als bei einer reinen Interpretation. Dies liegt vermutlich daran, daß die Compiler zum Teil nur in einer Beta-Versionen verfügbar und noch nicht ausgereift sind. In späteren Versionen müßte sich die Laufzeit eines Java-Programms an die eines C++-Programmes zumindest annähern lassen.
- Java ist eine interpretierte Sprache und daher kann der Bytecode leicht durch einen Recompiler wieder in den ursprünglichen Sourcecode zurückverwandelt werden. Deshalb wird es schwierig werden, Java-Programme kommerziell zu vermarkten, da theoretisch jeder Einsicht in den einem Programm zugrundeliegenden Sourcecode nehmen kann und damit wertvolle Entwicklungszeit spart.

3.4 Die Ausführungsgeschwindigkeit von Java-Programmen

Zu Beginn dieser Diplomarbeit war es noch nicht klar, ob die Ausführungsgeschwindigkeit der interpretierten Sprache Java den Anforderungen an das Laufzeitverhalten der zu implementierenden Benutzeroberfläche genügen würde. Das Monitoring Interface stellt jedoch keine großen Anforderungen an die Ausführungsgeschwindigkeit. Lediglich das Einlesen eines ConTracts und die anschließende Berechnung der Bildschirmdarstellung ist ein komplexer und rechenintensiver Vorgang, der schon einige Zeit dauern kann. Da der Benutzer von der eigentlichen Berechnung so gut wie nichts mitbekommt, ist es wichtig, diese Prozedur so schnell wie möglich zu halten, um den Benutzer nicht zu lange warten zu lassen. Da die eigentliche Verarbeitungsgeschwindigkeit der Java Virtual Machine nicht beeinflußt werden konnte, mußte durch geschickte Programmierung und durch die Wahl eines Algorithmus mit maximal quadratischem Aufwand das zeitliche Problem gelöst werden, was auch gelungen ist. Siehe auch hierzu das Kapitel „Komplexitätsbetrachtung“. Im großen und ganzen läßt sich sagen, daß Java den Anforderungen hinsichtlich der Abarbeitungsgeschwindigkeit durchaus genügt.

3.5 Java und das Thema Sicherheit

Da Java Programme meistens über ein Netzwerk wie zum Beispiel das Internet geladen werden und ihre Herkunft nicht immer hundert prozentig verifiziert werden kann, ist der Faktor Sicherheit ein wichtiger und nicht zu vernachlässigender Aspekt sowohl für den Benutzer als auch für den Java-Programmierer.

Gerade in der heutigen Zeit, wo der Trend eindeutig weg von einer lokalen Umgebung hin zu verteilten heterogenen Netzwerken geht, kann das Thema Sicherheit nicht stark genug beachtet werden. Es gibt bereits mehrere tausend mehr oder weniger bösartige Viren, die von harmlosen Späßen bis hin zu schwerwiegenden Zerstörungen alles auf den befallenen Systemen verursachen können. Auch ist eine vernetzte Umgebung und das Internet ein geradezu idealer Ort für Datenspionage oder -manipulation. Gerade bei Daten, die über unzählige nicht nachvollziehbare Stationen über das Netz laufen, ist eine Gefahr der Manipulation groß, und deshalb muß besonders auf eine gesicherte im Sinne von verifizierbare Datenübertragung geachtet werden.

Die Entwickler von Java waren sich dieser Probleme durchaus bewußt, und deshalb wurde alles getan, um den Einsatz von Java so sicher wie möglich zu gestalten. Eine Verifikationsroutine in der Java Virtual Machine überprüft, ob der durch den Java-Compiler erzeugte Bytecode während der Übertragung über das Netz verändert wurde oder ob vordefinierte Sprachkonstrukte verletzt werden. Die Verifikationsroutine soll sicherstellen, daß der Code selbst keine Daten fälscht oder auf gesperrten Speicher oder Objekte zugreift, die nicht definiert sind. Die Prüfung sorgt übrigens ebenfalls dafür, daß die Methodenaufrufe die richtige Anzahl von Argumenten des richtigen Typs enthalten und es zu keinen Stack-Überläufen kommt.

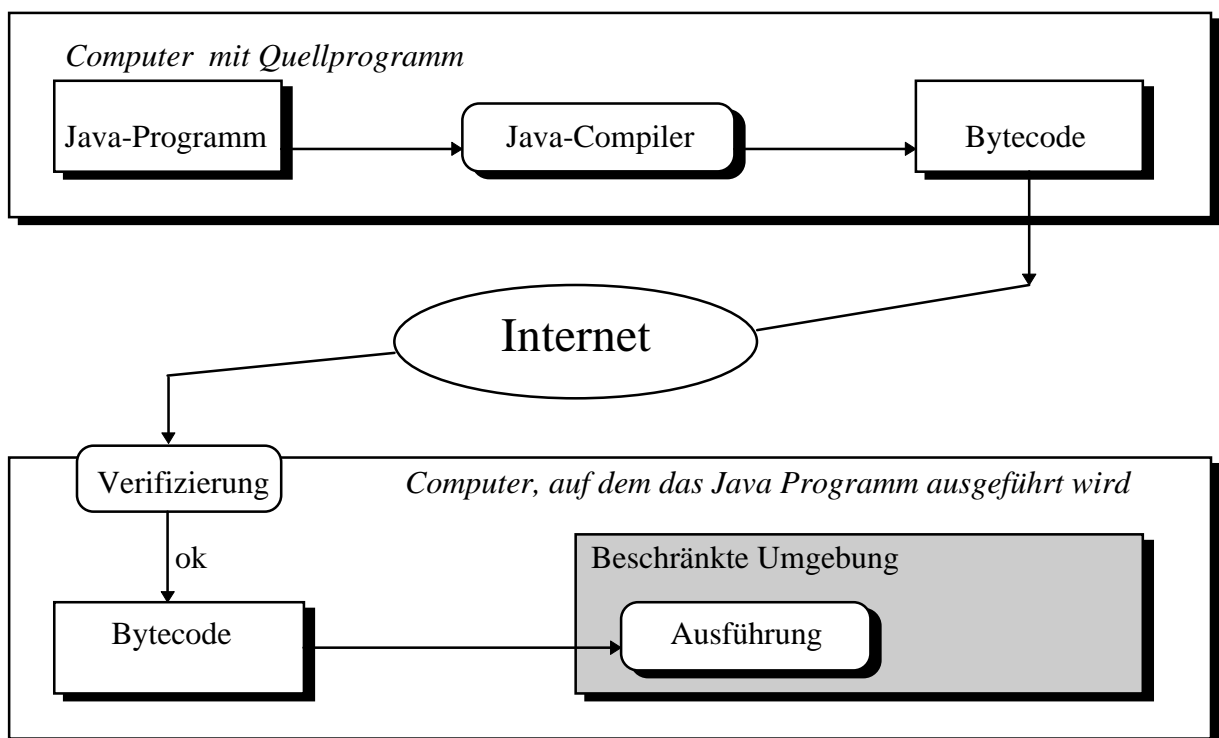


Abbildung 7: Der Verifizierungsprozeß für Java Programme

Abbildung 7 zeigt, wie der aus dem Netz geladene Bytecode zuerst auf mögliche Schutzverletzungen hin überprüft wird, bevor er auf dem lokalen Computer innerhalb einer beschränkten Umgebung ausgeführt werden kann. Da der Zugriff auf Methoden und Variablen innerhalb von Java nur über Namen und nicht wie sonst üblich über Adressen erfolgt, kann leicht festgestellt werden, welche Methoden und Funktionen tatsächlich benutzt werden. Und da im Bytecode außerdem zusätzliche Informationen über Typen enthalten sind, die der Überprüfung der Zulässigkeit des Programms dienen, kann sichergestellt werden, daß der Bytecode nicht unbefugt verändert werden kann.

Zusätzlich zu der Verifikationsroutine für den Bytecode wurden in Java noch folgende Sicherheitsmaßnahmen eingebaut:

- Java Programme laufen grundsätzlich in einer beschränkten Umgebung ab, d.h., sie haben nur begrenzten Zugriff auf Speicherbereiche und lokalen Festwertspeicher. Diese Beschränkungen schließen das Löschen der Zeigerarithmetik und illegaler Cast-Operatoren mit ein.
- Java besitzt ein Sicherheitssystem für Schnittstellen, das auf verschiedenen Ebenen unterschiedliche Sicherheitsvorkehrungen verwendet.
- Wenn auf der Ebene des Dateizugriffs versucht wird, illegal auf eine Datei zuzugreifen, wird dem Benutzer durch ein Dialogfeld die Möglichkeit gegeben, die Aktion zu stoppen.
- Bei der Ausführung des Bytecodes werden in diesem enthaltene Informationen dazu verwendet, zu entscheiden, welche Fähigkeiten der Code besitzt und woher er kommt. So kann zum Beispiel festgestellt werden, ob der Bytecode aus dem Bereich eines Firewall stammt oder nicht und gegebenenfalls im Programm festgelegt werden, ob der Zugriff auf Programme und Daten, die nicht aus dem gesicherten Bereich stammen, verboten wird.
- Wie bereits erwähnt gibt es Bemühungen, mit Hilfe von Public-Key-Verschlüsselung oder anderen Verschlüsselungstechniken die Herkunft des Codes und seiner Integrität durch eine Art elektronischer Unterschrift zu überprüfen. Exportbeschränkungen und internationale Gesetze, die den Einsatz von bestimmten Verschlüsselungstechniken verbieten, machen allerdings in diesem Bereich noch Probleme.

Wie man sieht, gibt es große Bemühungen, Java so sicher wie möglich zu machen. In fast allen Java Versionen wurden jedoch bisher Fehler bzw. Schlupflöcher gefunden, die es ermöglichen, einige der Sicherheitsvorkehrungen zu umgehen. Solange dies nicht behoben ist, ist der kommerzielle und professionelle Einsatz von Java in Unternehmen noch in weiter Ferne. Für das Monitoring Interface und das APRICOTS-System sind die Sicherheitsmechanismen von Java jedoch völlig ausreichend, da bisher noch keine sicherheitsrelevanten Daten übertragen werden müssen. Dies kann sich in Zukunft jedoch ändern, weshalb man sich in zukünftigen Arbeiten zum Thema APRICOTS durchaus Gedanken über diese Thematik machen sollte.

4 Das Prädikat-Transitionsnetz (PTN)

Wenn ein neuer ConTract eingegeben wird, so geschieht dies im APRICOTS-System mit Hilfe des Definition Interface, das in Kapitel 2 bereits kurz erläutert wurde. Das Definition Interface speichert einen ConTract in Form eines ConTract-Template ab, einer Art Schablone, die das ConTract-Skript genau beschreibt. Bevor das Skript jedoch vom ConTract Manager bearbeitet werden kann, muß aus dem ConTract-Template zuerst von einem Compiler eine Instanz erzeugt werden, die verändert werden darf. Diese sogenannte ConTract-Instanz ist nichts anderes als ein Prädikat-Transitionsnetz, das im folgenden beschrieben wird.

4.1 Was ist ein Prädikat-Transitionsnetz ?

Prädikat-Transitionsnetze, abgekürzt PTN, sind eine Erweiterung der von C.A.Petri 1962 entwickelten Petri-Netze und eignen sich besonders gut zur Modellierung von nebenläufigen Prozessen und zur Darstellung von asynchronen Ereignissen. Wie die Petri-Netze ist ein PTN ein gerichteter Graph mit bestimmten Markierungen, der allerdings nicht mehr bipartit ist, da ihm Prädikate hinzugefügt wurden. Auf Petri-Netze soll an dieser Stelle nicht näher eingegangen werden, weiterführende Informationen finden sich in der Dissertation des Erfinders der Petri-Netze, C.A.Petri [Petri62].

Prädikat-Transitionsnetze bestehen wie die Petri-Netze aus den folgenden Komponenten, zu denen noch die Prädikate hinzugekommen sind:

- **Transitionen:** Transitionen entsprechen den Übergängen in einem Petri-Netz und werden im folgenden durch senkrechte Striche dargestellt.
- **Steps:** Steps entsprechen den Stellen in einem Petri-Netz. Sie repräsentieren die elementaren Aktionen in einem Skript und werden im folgenden durch Kreise dargestellt.
- **Prädikate:** Prädikate haben keine Entsprechung in einem Petri-Netz, da sie die eigentliche Erweiterung darstellen. Sie werden dazu benutzt, um bedingte Verzweigungen formulieren zu können und evaluieren entweder zu wahr oder falsch. Es gibt mehrere verschiedene Typen von Prädikaten, die im folgenden in ihrer Grundstruktur immer als Dreiecke dargestellt werden.
- **Kanten:** Kanten sind gerichtete Verbindungen zwischen Transitionen, Steps und Prädikaten. Es dürfen jedoch niemals zwei Transitionen, zwei Steps oder zwei Prädikate miteinander verbunden werden. Die Reihenfolge der Komponenten in einem Prädikat-Transitionsnetz ist immer Transition, Step, Prädikat, Transition. Kanten werden im folgenden in Form von Pfeilen dargestellt.

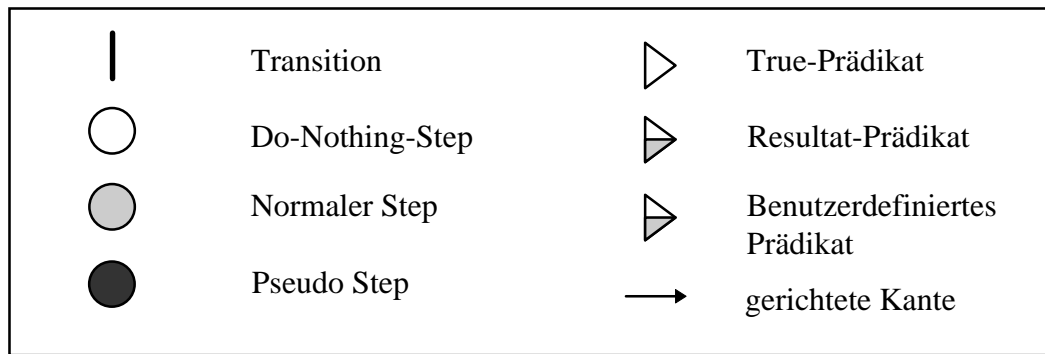


Abbildung 8: Komponenten eines Prädikat-Transitionsnetz

Abbildung 8 zeigt die einzelnen Komponenten eines Prädikat-Transitionsnetz. Do-Nothing Steps sind Steps, die vom Compiler eingefügt wurden um die Regeln des Prädikat-Transitionsnetz nicht zu verletzen und die keinerlei Aktionen beinhalten. Resultat- und Benutzerdefinierte Prädikate sind Prädikate, die im Gegensatz zu reinen True-Prädikaten zwei Ausgänge haben. Der Wahr-Ausgang dieser Prädikate wird durch den weißen Teil des Dreiecks dargestellt, der Falsch-Ausgang durch den schwarzen Teil.

4.1.1 Die einzelnen Komponenten eines PTN

Im folgenden werden nun die einzelnen (nicht transaktionalen) Komponenten eines Prädikat-Transitionsnetz vorgestellt. Das Prädikat-Transitionsnetz eines Skriptes wird von vielen Komponenten des APRICOTS-Systems eingelesen und bearbeitet, so zum Beispiel vom ConTract Manager oder vom Monitoring Interface. Die Kenntnis des Aufbaus eines PTN ist daher grundlegend für das Verständnis dieser Komponenten.

4.1.1.1 Der ConTract

Die folgenden Abbildungen zeigen die einzelnen Komponenten, aus denen ein Prädikat-Transitionsnetzes aufgebaut ist.

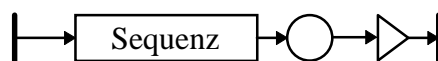


Abbildung 9: Ein ConTract

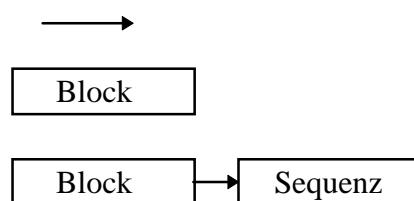


Abbildung 10: Die Struktur einer Sequenz

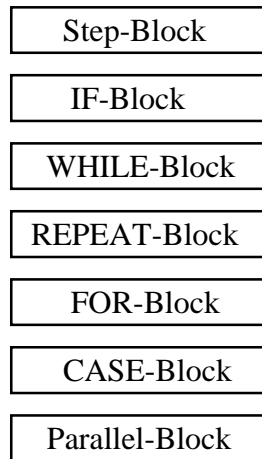


Abbildung 11: Die Struktur eines Blockes

Abbildung 9 zeigt einen ConTract. Der ConTract ist minimal, wenn die Sequenz leer ist, bzw. nur aus einer Kante besteht. Eine Sequenz kann aber auch aus einem Block oder einem Block gefolgt von einer weiteren Sequenz bestehen. Es lassen sich also beliebig viele Blöcke ineinander schachteln. Ein Block kann dabei wie Abbildung 11 zeigt, entweder aus einem Step, mehreren Schleifenkonstrukten oder aus einer Parallel-Anweisung bestehen. Die Parforeach-Anweisung wurde hier bewußt weggelassen, da sie in der gegenwärtigen APRICOTS-Implementierung noch nicht unterstützt wird.

4.1.1.2 Der Step-Block

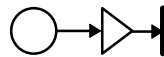


Abbildung 12: Do-Nothing-Step

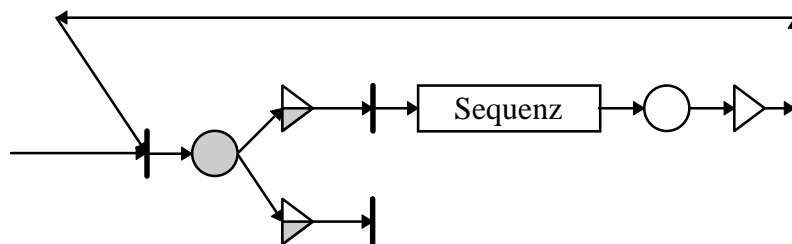


Abbildung 13: Step mit Konfliktbehebung

Abbildung 12 zeigt einen Do-Nothing-Step. Diese Art von Steps werden vom Compiler bei der Umwandlung eines ConTract-Templates in eine ConTract-Instanz eingefügt, um die Integrität des Prädikat-Transitionsnetz zu bewahren. Sie haben sonst keine Funktion. Abbildung 13 zeigt einen Step mit eingebauter Konfliktbehebung. Die beiden Prädikate nach dem ersten Step sind Resultat-Prädikate, die auf das Ergebnis der Step-Bearbeitung reagieren. Schlägt die Bearbeitung fehl, so wird der Step kompensiert und nochmal ausgeführt.

4.1.1.3 IF-Block

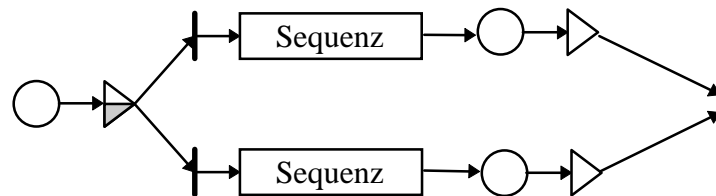


Abbildung 14: Der IF-Block

Abbildung 14 zeigt einen IF-Block. Je nachdem, wie die Auswertung des Prädikates ausfällt, wird zu einer bestimmten Sequenz verzweigt.

4.1.1.4 WHILE-Block

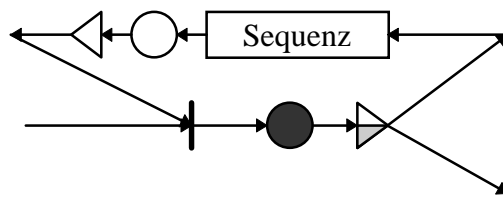


Abbildung 15: Der WHILE-Block

Abbildung 15 zeigt eine WHILE-Schleife. Je nachdem, wie das Prädikat ausgewertet wurde, wird der normale Programmablauf fortgesetzt oder der Schleifenrumpf durchlaufen. Der Schleifenrumpf wird dabei so lange ausgeführt, bis das Prädikat zu falsch evaluiert. Ein WHILE-Block entspricht dem WHILE ... DO Konstrukt in herkömmlichen Programmiersprachen.

4.1.1.5 REPEAT-Block

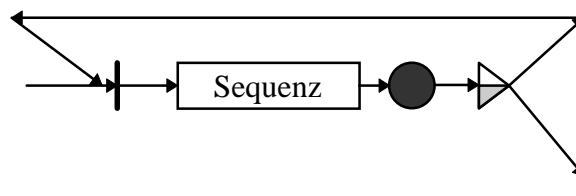


Abbildung 16: Der REPEAT-Block

Im Gegensatz zur WHILE-Schleife wird der Rumpf einer REPEAT-Schleife mindestens einmal ausgeführt. Erst danach wird das Prädikat ausgewertet und entschieden, ob die Schleife nochmals durchlaufen werden soll oder nicht. Ein REPEAT-Block entspricht damit dem DO...WHILE bzw. REPEAT ... UNTIL Konstrukt in herkömmlichen Programmiersprachen.

4.1.2 Transaktionen

Neben den im vorigen Abschnitt vorgestellten nicht transaktionalen Komponenten gibt es im Prädikat-Transitionsnetz auch die entsprechenden transaktionalen Komponenten. Diese laufen unter Transaktionsschutz (sind also als atomare Einheiten zu betrachten) und haben genau definierte Ein- und Ausgänge. Nur über diese kann eine Transaktion betreten oder wieder verlassen werden. Es ist nicht erlaubt, mitten aus einer Transaktion herauszuspringen.

Der Beginn einer Transaktion wird durch eine Transition vom Typ TK_TA_BEGIN eingeleitet, verlassen kann man eine Transaktion nur über die Transitionen vom Typ TK_TA_COMMIT oder TK_TA_ABORT. Die Commit-Transition wird gewählt, wenn die Ausführung des transaktionalen Blockes erfolgreich war, bei einem Fehler kann man die Transaktion über einen von theoretisch beliebig vielen Abort-Pfaden verlassen, so daß festgestellt werden kann, welchen Grund der Abbruch hatte.

Eine ConTract-Sequenz besteht also nicht nur aus Kanten und Blöcken, sondern auch aus den dazugehörigen Transaktions-Blöcken, kurz TA-Block.

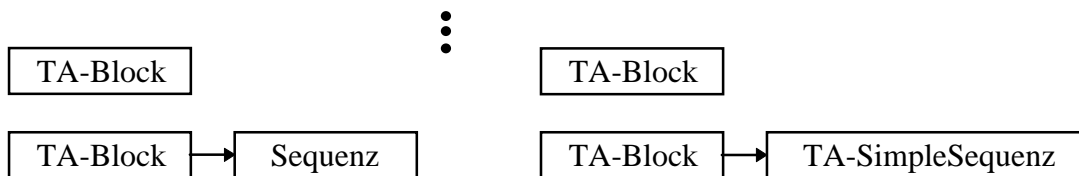


Abbildung 20: Struktur der Sequenz mit transaktionalen Blöcken

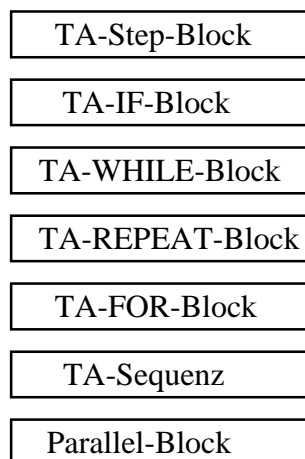


Abbildung 21: Die Struktur eines TA-Blocks

Alle im vorigen Abschnitt aufgeführten nicht transaktionalen Konstrukte haben eine transaktionale Entsprechung, die hier allerdings aus Platzgründen nicht mehr aufgeführt werden. Der Aufbau ist auch fast identisch, bis auf den Unterschied, daß Begin-, Commit- und Abort-Pfade hinzugefügt wurden. Der Unterschied zwischen transaktionalen und nicht transaktionalen Komponenten wird im nächsten Kapitel deutlich, welches sich mit dem Beweis der Planarität befaßt.

5 Graphentheoretische Überlegungen

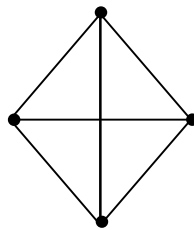
Die graphische Darstellung eines ConTracts bzw. einer ConTract-Instanz war eine der schwierigsten und komplexesten Aufgaben dieser Diplomarbeit. Nachdem das ConTract-Skript eingelesen und bearbeitet wurde, mußte eine möglichst kreuzungsfreie und für den Benutzer des Monitoring Interface übersichtliche Darstellung gefunden werden, die sich zudem noch mit minimalem Zeitaufwand berechnen und aufbauen ließ. Zu diesem Zweck war eine umfassende Einarbeitung in die Graphentheorie erforderlich, da sich dieses Gebiet unter anderem genau mit der gestellten Aufgabe befaßt.

5.1 Einführung in die Graphentheorie

Die Wurzel der Graphentheorie liegt in der Untersuchung topologischer Probleme, die sich durch eine Anzahl von Punkten (Knoten) und deren Verbindungen untereinander (Kanten) charakterisieren lassen. Ein Graph G besteht aus einer endlichen, nicht leeren Menge V von Knoten zusammen mit einer Menge E von zweielementigen Teilmengen von V . Jedes ungeordnete Paar $e = \{u, v\}$ von Knoten, das zu E gehört, ist eine Kante von G und verbindet u mit v .

Definition Graph: Ein Graph G ist ein Paar $G=(V,E)$ aus einer endlichen Menge $V \ll \emptyset$ von Knoten und einer Menge E von zweielementigen Teilmengen von V , den Kanten.

a) planarer Graph G



b) ebener Graph G'

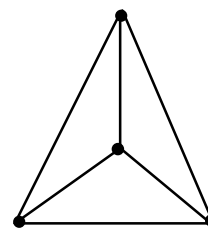


Abbildung 22: Beispiele für Graphen

Ersetzt man nun alle Steps, Transitionen und Prädikate in einem ConTract-Skript durch Knoten, und betrachtet man alle Verbindungen als gerichtete Kanten, dann ist ein ConTract-Skript bzw. das zugrundeliegende Prädikat-Transitionsnetz nichts anderes als ein gerichteter Graph G , für den alle Gesetze und Definitionen der Graphentheorie Gültigkeit haben.

Definition gerichtet: Ein Graph $G=(V,E)$ heißt gerichtet, wenn die Teilmenge E aus geordneten Paaren $e=(u,v)$ besteht, d.h., ein Paar $e=(u,v)$ mit $u,v \in V$ ist nicht gleich $e=(v,u)$ mit $v,u \in V$.

5.2 Planarität und Kreuzungsfreiheit

Häufig tritt in der Praxis das Problem auf, Graphen kreuzungsfrei in die Ebene einzubetten. Dies ist zum Beispiel besonders wichtig beim Entwurf von elektrischen Schaltungen oder bei der Darstellung von Netzwerken auf einem Computer-Bildschirm. Auch bei der Darstellung eines ConTract-Skriptes stößt man auf dieses Problem.

Definition planar: Ein Graph G heißt planar oder plättbar, wenn er sich in die Ebene einbetten läßt. Die Einbettung von G bezeichnet man dabei als ebenen Graphen G' .

Definition einbettbar: Ein Graph G heißt in die Ebene P einbettbar, wenn er so auf P gezeichnet werden kann, daß sich keine zwei Kanten schneiden.

Abbildung 21 b) zeigt einen ebenen Graphen G' zu dem planaren Graphen G von Abbildung 21 a). Man beachte den folgenden, sehr wesentlichen Unterschied zwischen dem Begriff Graph (speziell planarer Graph) und dem Begriff ebener Graph: Während sich ein Graph nur aus zwei verschiedenen Elementen aufbaut, nämlich den Knoten und den Kanten, besitzt ein ebener Graph drei verschiedene Arten von Elementen. Zu den Knoten und Kanten - genauer gesagt, zu deren Einbettungen - kommen noch die *Gebiete* hinzu, in die die Ebene durch den Graphen zerlegt wird.

Es ist ebenfalls wichtig zu wissen, daß man zwar beweisen kann, ob ein Graph planar ist oder nicht. Aus dem Beweis der Planarität kann man aber noch nicht ableiten, wie die Knoten und Kanten in der Ebene anzuordnen sind, um eine überschneidungsfreie Darstellung (Einbettung) zu erhalten. Außerdem ist es für den Beweis der Planarität unerheblich, welche Orientierung der vorliegende Graph hat, d.h., ob er gerichtet ist oder nicht. Ebenso spielen Schlingen (Anfangs- und Endknoten eines Bogens fallen zusammen) und Mehrfachkanten (verschiedene Kanten haben gleiche Anfangs- und Endknoten) keine Rolle, so daß man alle Graphen als schlicht voraussetzen kann. Denn falls ein vorgegebener Graph nicht schlicht ist, entfernt man einfach alle Schlingen und ersetzt die Mehrfachkanten durch eine einzige. Jetzt untersucht man den neu entstandenen Graphen auf Planarität. Ist dieser planar, so ist es auch der Originalgraph. Ist er nicht planar, dann ist es auch nicht der Originalgraph.

Diese Eigenschaften sind besonders wichtig für die Anwendung der nachfolgenden Planaritätskriterien auf ein ConTract-Skript.

5.3 Planaritätskriterien

Wie findet man nun heraus, ob ein Graph planar ist oder nicht? Eine Möglichkeit wäre, durch Probieren zu versuchen, eine kreuzungsfreie Darstellung zu finden, was bei Graphen mit wenigen Knotenpunkten sicher auch relativ schnell gelingen würde. Im Falle vieler Knotenpunkte und Kanten wäre eine solche Vorgehensweise allerdings sehr mühsam, und eine Entscheidung, ob ein vorgegebener Graph einbettbar ist oder nicht, könnte kaum getroffen werden. Außerdem würde auch ein extrem schneller Rechner für das Durchpro-

bieren aller (endlich vielen) Möglichkeiten eine nicht akzeptable Zeitspanne brauchen, so daß dieser Ansatz von vornherein verworfen wurde.

5.3.1 Der Satz von Kuratowski

Ein Kriterium für die Planarität von Graphen liefert der Satz von Kuratowski. Vor der Angabe dieses Satzes muß allerdings erst noch der Begriff *Unterteilung* definiert werden:

Definition Unterteilung: Ein Graph H heißt *Unterteilung* eines Graphen G , falls H aus G dadurch entsteht, daß man Knotenpunkte in das Innere von Kanten aus G einfügt. Im Fall $G=H$ ist der Graph H ebenfalls *Unterteilung* von G .

Kuratowski benutzt in seinem Theorem die beiden Graphen K_5 und $K_{3,3}$, die in Abbildung 23 dargestellt werden.

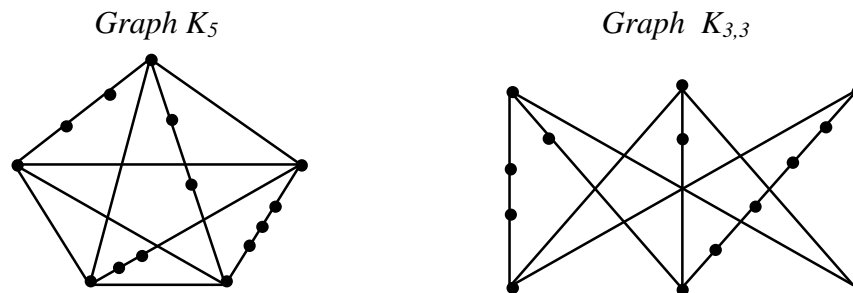


Abbildung 23: Die beiden Graphen des Satzes von Kuratowski

Keiner der beiden Graphen K_5 und $K_{3,3}$ ist planar, was man durch einfaches Probieren schnell erkennen kann. Damit ist auch keine Unterteilung eines dieser Graphen planar, weil ein Graph zu jeder seiner Unterteilungen äquivalent ist [Sachs72]. Weiter ist klar, daß jeder Untergraph eines planaren Graphen ebenfalls planar ist. Daraus folgt, daß ein Graph, der eine Unterteilung eines der Graphen aus Abbildung 23 enthält, sicher nicht planar ist. Der Satz von Kuratowski besagt, daß hiervon auch die Umkehrung gilt:

Satz von Kuratowski: Ein Graph ist genau dann nicht planar, wenn er eine Unterteilung des vollständigen Graphen K_5 mit fünf Knotenpunkten oder des vollständigen Graphen $K_{3,3}$ mit je drei Knotenpunkten in den beiden Klassen enthält.

Wie man unschwer erkennen kann, bringt die Anwendung des Satzes von Kuratowski bei praktischen Planaritätsuntersuchungen natürlich die Schwierigkeit mit sich, in einem Graphen Unterteilungen der beiden Graphen K_5 und $K_{3,3}$ zu erkennen. Dies erweist sich in der Praxis als sehr schwierig und als für die vorliegende Aufgabe nicht geeignet. Deshalb wurde in dieser Arbeit auf eine Anwendung des Satzes verzichtet und nach besser geeigneten Planaritätskriterien gesucht.

5.3.2 Der Satz von Whitney

Ein weiteres Planaritätskriterium liefert der Satz von Whitney. Zu seinem Verständnis sind jedoch wieder einige Definitionen erforderlich, die im folgenden gegeben werden:

Definition Schnitt: Eine Kantenmenge β eines ungerichteten Graphen G heißt Schnitt von G , wenn die folgenden Bedingungen erfüllt sind:

- 1. Nach Entfernen der Kanten von β aus G ist der Restgraph nicht mehr zusammenhängend.*
- 2. Die Kantenmenge β ist minimal, d.h. keine echte Teilmenge $\beta' \subset \beta$ erfüllt Bedingung 1.*

Definition Kreis: Ein gerichteter Graph bildet einen Kreis, wenn er eine geschlossene Kantenfolge besitzt, in der alle Ecken außer der ersten und der letzten verschieden sind (Zyklus) und in der alle Kanten im Sinne des Zyklusdurchlaufs orientiert sind.

Definition dual: Sei G ein mindestens zweifach zusammenhängender Graph. Ein Graph DG heißt zu G dual, wenn es eine eineindeutige Abbildung der Kanten von G auf DG gibt, die der folgenden Bedingung genügt:

Eine Menge von Kanten von G bildet genau dann einen Schnitt, falls die ihr in DG entsprechende Kantenmenge einen Kreis bildet.

Nach Kenntnis der obigen Definitionen kann nun der Satz von Whitney geliefert werden:

Satz von Whitney: Ein Graph G ist genau dann planar, wenn es einen zu ihm dualen Graphen DG gibt.

Es erweist sich, daß der zu einem planaren Graphen G duale Graph DG ebenfalls planar ist. Man könnte nun den Satz von Whitney dazu benutzen, nachzuprüfen, ob ein Graph planar ist oder nicht, indem man einen zu ihm dualen Graphen sucht. Das für die Aufgabenstellung dieser Diplomarbeit dem Graphen zugrundeliegende ConTract-Skript ist aber leider nicht zweifach zusammenhängend, was jedoch bei der Definition des Begriffs dual gefordert wird. Deshalb eignet sich auch dieser Satz nicht zur Lösung der vorliegenden Aufgabe.

5.3.3 Der Satz von McLane

Der dritte allgemein bekannte Satz zur Lösung des Planaritätsproblem ist der Satz von McLane. Er baut auf dem Begriff der Kreisbasis auf, die im folgenden definiert wird:

Definition Kreisbasis: Eine Teilmenge S aller Kreise eines Graphen G heißt Kreisbasis, sofern sich ein beliebiger Kreis des Graphen durch Addition geeigneter Kreise von S erzeugen läßt und keine echte Untermenge von S jeden Kreis zu erzeugen gestattet.

Es gilt der folgende Satz:

Satz: *Jeder Graph besitzt (wenigstens) eine Kreisbasis. Jede Kreisbasis besteht aus genau $m - n + 1$ Kreisen, wobei m die Anzahl der Kanten des Graphen und n die Anzahl der Knoten des Graphen bezeichnet.*

Mit Hilfe der obigen Definition und des obigen Satzes kann nun der Satz von McLane formuliert werden:

Satz von McLane: *Ein Graph G ist genau dann planar, wenn er eine solche Kreisbasis S besitzt, daß keine Kante des Graphen in mehr als zwei Kreisen von S liegt.*

Eine Kreisbasis, die den Forderungen des Satzes von McLane genügt, kann einfach gefunden werden, allerdings muß auch hierzu der Graph zweifach zusammenhängend sein. Das heißt, der Graph ist selbst zusammenhängend und bleibt es auch, wenn man einen beliebigen Knoten und die mit ihm zusammenhängenden Kanten entfernt. Wie vorher beim Satz von Whitney schon erwähnt, ist der Graph eines ConTract-Skriptes jedoch nicht zweifach zusammenhängend, was man sofort beweisen kann, wenn man aus einem beliebigen ConTract-Skript einen geeigneten Knoten entfernt. Deshalb kann auch der Satz von McLane leider nicht dazu benutzt werden, ein ConTract-Skript auf Planarität zu untersuchen. Es kann jedoch schon jetzt vorweg genommen werden, daß die im nächsten Abschnitt beschriebene Euler'sche Polyederformel endlich das gewünschte Ergebnis liefert.

5.3.4 Die Euler'sche Polyederformel

Einen letzten Hinweis auf die Eigenschaften planarer Graphen liefert die Euler'sche Polyederformel, die besagt:

Satz: *Für jedes sphärische Polyeder mit V Ecken, E Kanten und F Flächen gilt:*

$$V - E + F = 2 \quad (1)$$

Ein solches Polyeder kann man als spezielle Einbettung des aus den Ecken und Kanten des Polyeders bestehenden Graphen in die Kugeloberfläche ansehen, und die Formel (1) kann man auch auf beliebig zusammenhängende ebene (plättbare) Graphen anwenden [Sachs72]. Aus der Formel läßt sich schnell errechnen, daß ein schlichter planarer Graph weniger als $3n$ Kanten besitzt, wenn n die Knotenanzahl ist. Sofern also die Kantenzahl $3n - 5$ übersteigt, ist der Graph mit Sicherheit nicht mehr planar.

Setzt man $p = V$, $q = E$ und $r = F$ für eine ebene Landkarte (das ist ein zusammenhängender, ebener Graph zusammen mit allen seinen Gebieten), so erhält man die zur Euler'schen Polyederformel bis auf die Bezeichner identische Formel:

$$p - q + r = 2 \quad (2)$$

Aus dieser zweiten Formel geht ein wichtiger Korollar für (p,q) -Landkarten hervor:

Korollar 1: Wenn G eine ebene (p,q) -Landkarte mit p Ecken und q Kanten ist, in der jedes Gebiet ein n -Eck, d.h., ein Kreis der Länge n ist, dann gilt:

$$q = n \frac{(p-2)}{(n-2)} \quad (3)$$

Mit Hilfe obiger Formel (3) kann man beweisen, ob G ein ebener Graph ist oder nicht (also planar und in die Ebene einbettbar ist oder nicht). Leider kann man die Formel nicht auf jeden beliebigen gegebenen Graphen anwenden, da komplexere Graphen viele verschiedene Gebiete haben, woraus folgt, daß es natürlich auch viele verschiedene n -Ecke gibt. Auf einfachen und überschaubaren Graphen liefert die Formel jedoch brauchbare Ergebnisse.

Satz: Ein Graph ist genau dann plättbar, wenn jedes seiner Glieder plättbar ist.

Mit Kenntnis der obigen Formel (3) und des obigen Satzes gelangt man nun zu einem ganz anderen Ansatz, um die Plättbarkeit eines Graphen bzw. eines ConTract-Skripts nachprüfen zu können:

Anstatt jedesmal zu überprüfen, ob ein gegebenes ConTract-Skript plättbar ist, genügt es zu beweisen, daß jede mögliche Komponente (Glieder) eines ConTract-Skripts bzw. Prädikat-Transitionsnetzes plättbar ist. Ist dies nämlich der Fall, dann muß auch jedes aus diesen Komponenten erzeugte Prädikat-Transitionsnetz plättbar sein. In Kapitel 4 wurden die einzelnen Komponenten bereits ausführlich beschrieben, so daß im folgenden nur noch der Beweis für ihre Plättbarkeit angetreten wird.

5.4 Beweis der Planarität der einzelnen Komponenten eines PTN

Der Beweis der Planarität der einzelnen Komponenten eines Prädikat-Transitionsnetzes wurde bereits in einer früheren Arbeit erbracht [Eitl92]. Allerdings wurden damals weder Prädikate, noch Transaktionen oder Schleifen wie zum Beispiel FOR- oder REPEAT-Schleifen betrachtet. Prädikate spielen keine besondere Rolle, da sie im nachfolgenden Beweis ebenso wie Steps und Transitionen auf Knoten eines Graphen abgebildet werden. Auf die anderen Konstrukte muß jedoch besonders eingegangen werden, da sie aufgrund ihrer Konstruktion dazu geeignet scheinen, die Planaritätskriterien zu verletzen.

5.4.1 ConTract und Step-Block

Die Planarität eines ConTracts, der nur aus einer Kante (Linie) besteht, auf der mehrere Knoten linear angeordnet sind, ist offensichtlich. Ebenso die Planarität eines Step-Blocks. Auf einen Beweis wird daher verzichtet.

5.4.2 Parallel Block

Zum Beweis der Planarität eines Parallel-Blocks fassen wir diesen als Graph auf, wobei wir die Steps, Transitionen und Prädikate des Blocks als Knoten des Graphen darstellen. Auch die in den Block eingebetteten Sequenzen können wir als einfache Knoten betrachten, sie werden

jedoch der Übersichtlichkeit halber immer noch als Sequenz dargestellt. Mit dieser Einschränkung sieht der Graph eines Parallel-Blocks wie folgt aus:

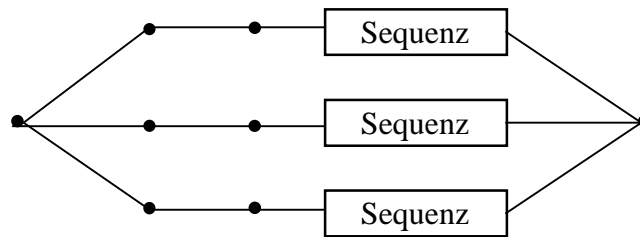


Abbildung 24: Der Graph eines Parallel-Blocks

Ein Parallel-Block mit m Zweigen ($m > 1$) hat also immer $p = 3m + 2$ Ecken, $q = 4m$ Kanten und $(m-1)$ Kreise der Länge $n=8$. Eingesetzt in unsere gefundene Formel (3) ergibt dies den folgenden Ausdruck:

$$4m = 8 * \frac{3m+2-2}{8-2} = 8 * \frac{3m}{6} = 8 * \frac{m}{2} = 4m \quad \text{wahr}$$

Für den Graph in Abbildung 24 gilt zum Beispiel: $p = 11$, $q = 12$ und der Graph besitzt zwei Kreise der Länge $n = 8$. Eingesetzt in unsere Formel ergibt dies:

$$12 = 8 * \frac{11-2}{8-2} = 8 * \frac{9}{6} = 12 \quad \text{wahr}$$

$12 = 12$ ist eine wahre Aussage, ebenso wie der allgemeine Ausdruck für Parallel-Blöcke mit m Zweigen, $4m = 4m$. Ein Parallel-Block ist also sicherlich immer planar.

5.4.3 IF-Block und CASE-Block

Der Graph eines IF-Blocks oder eines CASE-Blocks ist mit dem Graphen des Parallel-Blocks fast identisch, daher verläuft der Beweis für die Planarität fast analog. Man muß lediglich wissen, daß der CASE-Block in weitere Einzelkomponenten zerlegt werden kann, für die die Planarität gesondert bewiesen werden muß. Auf eine Ausführung des Beweises wird aufgrund seiner Ähnlichkeit zu obigem Beweis an dieser Stelle verzichtet.

5.4.4 Schleifen

Wie schon erwähnt, scheint der Beweis der Planarität bei den Schleifen-Konstrukten auf den ersten Blick schwieriger. Da der Graph nicht exakt in Gebiete abgegrenzt werden kann bzw. da es nicht abgeschlossene Außengebiete gibt, kann unsere Formel (3) nicht so ohne weiteres angewendet werden. Abbildung 25 zeigt als Beispiel den Graph der REPEAT-Schleife.

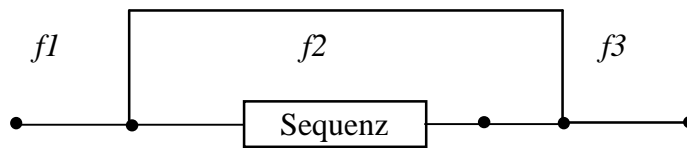


Abbildung 25: Graph einer REPEAT-Schleife

Abbildung 25 zeigt die drei Gebiete $f1$, $f2$ und $f3$ des Graphen einer REPEAT-Schleife. Nur das Gebiet $f2$ ist durch einen Kreis begrenzt, die Gebiete $f1$ und $f3$ sind Außengebiete. Aus der Graphik ist jedoch offensichtlich, daß die Gebiete $f1$ und $f3$ planar eingebettet werden können, daher kann man das Gebiet $f2$ als einfachen Block betrachten, auf den wir wieder unsere Formel anwenden können.

Der innere Block einer REPEAT-Schleife besitzt immer $p = 4$ Ecken, $q = 4$ Kanten und einen Kreis der Länge $n = 4$. Eingesetzt in unsere Formel (3) ergibt dies:

$$4 = 4 * \frac{4-2}{4-2} = 4 * \frac{2}{2} = 4 \quad \text{wahr}$$

Es ist ja auch trivial, daß ein einfaches Viereck planar in eine Ebene eingebettet werden kann. Dasselbe gilt auch für FOR- und WHILE-Schleifen, so daß wir nun den Beweis der Planarität für Schleifen als erbracht ansehen können.

5.4.5 Transaktionen und Konfliktauflösungen

Wie schon im Kapitel über Transaktionen erwähnt wurde, sind die transaktionalen Komponenten eines Prädikat-Transitionsnetzes bis auf die genau definierten Ein- und Ausgänge, über die eine Transaktion betreten und auch wieder verlassen werden muß, identisch zu ihren nicht transaktionalen Entsprechungen, für die die Planarität bereits bewiesen wurde. Die Frage ist nun, ob auch die transaktionalen Komponenten den Planaritätseigenschaften genügen.

Das Problem kann aufgrund der Ähnlichkeit der Konstrukte auf die Frage reduziert werden, ob ein Graph, für den die Planarität bereits bewiesen wurde, durch Hinzufügen von einigen Verzweigungen (konkret: einem Commit- und einem bis mehreren Abort-Pfaden) so verändert werden kann, daß er nicht mehr planar ist. Und dies ist leider der Fall !

Betrachten wir zum Beweis den Graphen eines nicht transaktionalen Verzweigungsblocks und des zugehörigen transaktionalen Blocks.

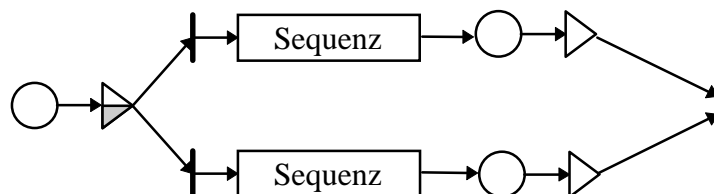


Abbildung 26: Nicht transaktionaler IF-Block

Abbildung 26 zeigt einen aus einem vorigen Kapitel schon bekannten nicht transaktionalen IF-Block, der nun durch Hinzufügen einer Begin-, Commit- und Abort-Transition zu einem transaktionalen Block umgebaut wird, wie die nächste Abbildung zeigt.

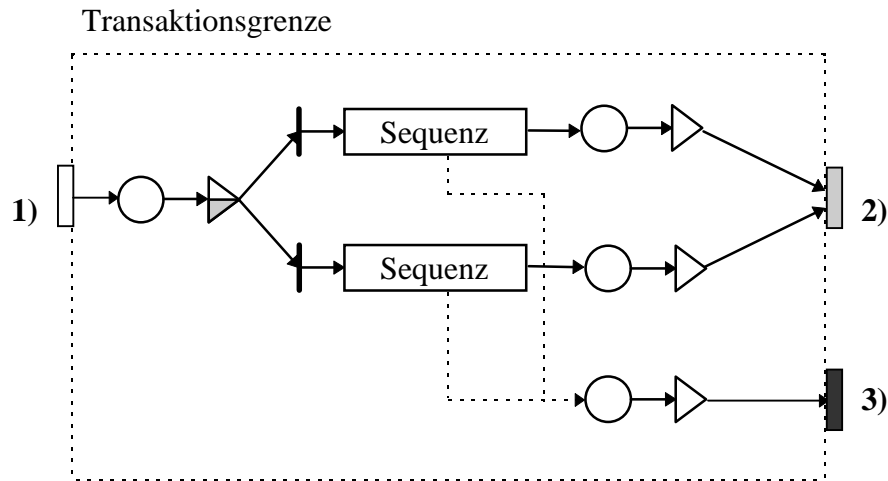


Abbildung 27: Transaktionaler IF-Block

Abbildung 27 zeigt einen transaktionalen IF-Block, der von einem gestrichelten Viereck zur Verdeutlichung der Transaktionsgrenze umgeben ist. Die Transaktion kann nur im Punkt 1) über die Begin-Transition betreten und in Punkt 2) über die Commit-Transition verlassen werden. Im Falle eines Fehlers kann die Transaktion auch am Punkt 3) über die Abort-Transition verlassen werden. Es sind theoretisch beliebig viele Abort-Transitionen möglich.

Wenn wir es nun schaffen, den Graphen aus Abbildung 27 so zu zeichnen, daß keine Linie die Transaktionsgrenze verläßt oder schneidet und sich keine zwei Linien überschneiden, dann ist der Graph planar und in die Ebene einbettbar. Dies ist jedoch schlicht unmöglich, weil die Abort-Pfade nie so gezeichnet werden können, daß sich keine zwei Linien schneiden.

Der mathematische Beweis ergibt sich aus einem weiteren Korollar zur Euler'schen Polyederformel:

Korollar2: Für jeden plättbaren (p,q) -Graphen G mit $p \geq 3$ gilt:

$$q \leq 3p - 6 \quad (1)$$

Wenn G keine Dreiecke enthält, gilt sogar:

$$q \leq 2p - 4 \quad (2)$$

Für den vereinfachten Graph, der sich aus Abbildung 27 ergibt, gilt: $p = 6$ und $q = 10$. Da der Graph keine Dreiecke besitzt, wenden wir Formel (2) an:

$$10 \leq 2 * 6 - 4$$

$$\Leftrightarrow 10 \leq 12 - 4$$

$$\Leftrightarrow 10 \leq 8 \quad \textit{falsch !}$$

5.5 Fazit

Wie wir soeben gezeigt haben, ist der transaktionale IF-Block aus Abbildung 27 nicht planar, weil seine Abort-Pfade die Planaritätskriterien verletzen und er deshalb nicht kreuzungsfrei in der Ebene gezeichnet werden kann. Aus diesem Grunde können wir uns eine Untersuchung weiterer transaktionaler Komponenten ersparen, da es für das Aufweichen der Planarität eines ConTract-Skripts bereits genügt, wenn eine einzige der Komponenten nicht plättbar ist.

Wir haben jedoch immerhin bewiesen, daß ein ConTract-Skript, das keine Transaktionen enthält, wie gewünscht kreuzungsfrei in die Ebene gezeichnet werden kann. Und falls man die Abort-Pfade ausblendet, kann man auch ein Skript mit Transaktionen kreuzungsfrei zeichnen. Dies ist mit ein Grund, warum im Monitoring Interface bei der Darstellung eines ConTracts die Abort-Pfade zuerst ausgeblendet werden.

6 Die Bearbeitung eines ConTract-Skriptes

6.1 Das Einlesen eines PTN mit Hilfe des Storage Agents

Eine der Hauptaufgaben des Monitoring Interface ist das Einlesen und die anschließende graphische Aufbereitung einer ConTract-Instanz, die ja nichts anderes ist, als das in Kapitel 4 beschriebene Prädikat-Transitionsnetz. Zum Einlesen bedient sich das Monitoring Interface einer weiteren Komponente des APRICOTS-Systems, dem Storage Agent. Der Storage Agent ist für die persistente Speicherung der von den anderen Komponenten des Systems erzeugten Daten zuständig und bietet dazu eine Schnittstelle mit Input-/Output-Funktionen an, die von anderen Komponenten bequem benutzt werden können.

6.1.1 Die Schnittstellen des Storage Agents

Der Umgang mit dem Storage Agents soll hier nur kurz anhand von einigen Beispielen erläutert werden. Eine vollständige Beschreibung der Schnittstellen findet sich im Anhang dieser Diplomarbeit.

6.1.1.1 Der Verbindungsaufbau zum Storage Agent

Die Kommunikation mit dem Storage Agent läuft wie bei allen anderen Komponenten des APRICOTS-Systems auch über CORBA bzw. Orbix ab. Um den Storage Agent ansprechen zu können, muß über den Naming Service von CORBA zuerst eine Verbindung zu diesem aufgebaut werden. Dies geschieht in der vorliegenden Diplomarbeit durch einen Aufruf der Methode *io.bind_to_storageagent(ciid)* der Klasse *io*. In der Klasse *io* werden im Sinne eines objektorientierten Ansatzes sämtliche Methoden zur Kommunikation mit anderen Agenten gekapselt, so daß sie theoretisch von anderen Klassen und Komponenten benutzt werden kann. Beim Aufruf der Methode wird gleich die *ciid (ConTractInstanceID)* der ConTract Instanz mitgegeben, mit deren Hilfe eine Instanz eindeutig identifiziert wird. Die Methode *bind_to_storageagent* ist wie folgt implementiert:

```
public void bind_to_storageagent(APRICOTS_Definitions.ConTractInstanceID ciid) {  
  
    NamingHelper nHelper;  
    String cid = String.valueOf(ciid.cid);  
    String cidInst = String.valueOf(ciid.cidInst);  
  
    mon1.statusleiste.setText("Connecting to StorageAgent ...");  
    String path = "APRICOTS.ConTracts."+cid+"."+cidInst+".StorageAgent";  
  
    try  
    {  
        nHelper = new NamingHelper(path); // use NamingHelper  
        ref = nHelper.getObjectRef();  
    }  
    catch (IE.Iona.Orbix2.CORBA.SystemException ex) // catch exceptions
```

```
    {
    ...
    }

    try
    {
        sAgent = APRICOTS_StorageAgent._narrow(ref); // narrow
    }
    catch (Exception ex) // catch exceptions
    {
    ...
    }

    mon1.statusleiste.setText("");
} // end of bind_to_monitoragent(ciid)
```

6.1.1.2 beginSession

Die meisten Methoden des Storage Agents erfordern als Eingabeparameter eine *SessionID*, mit deren Hilfe eine Session (also eine Verbindung) genau identifiziert werden kann. Nach erfolgreichem `_bind` an den Storage Agent muß also als erstes eine Session aufgemacht werden. Dies geschieht durch den Aufruf der Methode *io.beginSession()*, die bei erfolgreicher Ausführung eine SessionID in der Variable *session* der Klasse *io* ablegt.

6.1.1.3 getConTractState

Durch einen Aufruf der Methode *io.getConTractState()* kann jetzt der Status des ConTract bzw. der ConTract-Instanz ausgelesen werden. Die Methode liefert Angaben über die ConTract Engine, die die Instanz zuletzt bearbeitet hat, den neuen Zustand der Instanz, das Datum der letzten persistenten Speicherung und den Inhalt des logFile, in dem alle Änderungen mitprotokolliert werden.

6.1.1.4 getStartTransition

Mit Hilfe der Methode *io.getStartTransition()* wird nun die erste Transition des Prädikat-Transitionsnetzes eingelesen, die den Ausgangspunkt für alle weiteren Lesevorgänge darstellt. Sie wird in der Variable *transition* der Klasse *io* zur weiteren Verarbeitung zwischengespeichert.

6.1.1.5 getStepObjects

Nach den Regeln des Prädikat-Transitionsnetz kommen nach einer Transition nur Steps als Folgeobjekte in Frage. Diese können mit der Methode *io.getStepObjects(transID)* eingelesen werden. Die Methode liest dabei alle Steps ein, die über maximal eine Kante von der Start-Transition mit der ID *transID* aus erreicht werden können und speichert sie in der Variablen *sos* ab, die vom Typ *stepObjectSeq* ist.

6.1.1.6 getPredicates

Nach einem Step können dann mit Hilfe der Methode *io.getPredicates(roid)* alle Prädikate eingelesen werden, die dem Step mit der ID *roid* folgen. Die Prädikate werden in der Variablen *ps* zwischengespeichert, die vom Typ PredicateSeq ist.

6.1.1.7 getTransition

Nach einem Prädikat folgt dann nach den Regeln des Prädikat-Transitionsnetzes wieder eine Transition, die mit der Methode *io.getTransition(transID)* eingelesen wird. Die Methode ist fast identisch mit der schon bekannten Methode *io.getStartTransition()*, nur wird hier eben eine Transition nach einem beliebigen Prädikat gelesen, und nicht die Start-Transition.

6.1.1.8 getTransactions

Nachdem nun alle Komponenten des Prädikat-Transitionsnetzes eingelesen werden können, kann man mit Hilfe der Methode *io.getTransactions(parent)* noch alle Transaktionen mit der Elterntxaktion *parent* einlesen, damit man weiß, welche Objekte transaktional sind und welche nicht.

6.1.1.9 freeSession

Nach erfolgreichem Einlesen sollte man durch einen Aufruf der Methode *io.freeSession()* die zuvor angeforderte Session wieder beenden und den allokierten Speicher freigeben. Wenn die Session noch aktiv ist, wird ein sogenannter Rollback auf alle zuvor gemachten Änderungen durchgeführt, das heißt, der ursprüngliche Zustand wird wieder hergestellt.

6.1.2 Die interne Darstellung im Hauptspeicher

Nachdem nun die für das Monitoring Interface wichtigsten Schnittstellen des Storage Agents beschrieben wurden, soll an dieser Stelle erläutert werden, wie genau beim Einlesen eines Prädikat-Transitionsnetzes in den Hauptspeicher vorgegangen wird und welche Speicherstruktur dabei aufgebaut wird.

Das eigentliche Einlesen passiert innerhalb der Routine *goReadConTract()*. Durch einen Aufruf der schon bekannten Methode *io.getStartTransition()* wird die erste Transition ermittelt und in den Vector *z* eingelesen. Ein Vector ist eine spezielle Klasse in Java und implementiert ein Array aus Objekten, das bei Bedarf in seiner Größe wachsen kann. Dies hat den Vorteil, daß man nicht schon von vornherein wissen muß, wie viele Objekte eingelesen werden. Der Speicher kann also dynamisch allokiert werden und es wird nur gerade so viel Speicher belegt, wie nötig.

Da in einem Vector nur Objekte gespeichert werden können, wurde eine Klasse *Element* angelegt, die sämtliche Daten zu den verschiedenen Objekten aufnimmt, wie zum Beispiel Angaben über den Typ des Objekts (Transition, Step oder Prädikat), die Vorgängerobjekte, die Nachfolgerobjekte, die x- und y-Position der Objekte auf dem Bildschirm und noch viele weitere Informationen.

Die Daten der Start-Transition werden also in einer Instanz der Klasse *Element* gespeichert und diese wird dann dem Vector z zugefügt. Anschließend wird geprüft, von welchem Typ das zuletzt gelesene Objekt war. Davon abhängig werden entweder die nachfolgenden Stepobjekte, Prädikate oder Transitionen eingelesen. Deren Daten werden dann wiederum in Objekten verpackt dem Vector z zugefügt. Dieser Vorgang wird so lange wiederholt, bis alle Objekte des Prädikat-Transitionsnetzes eingelesen wurden. Am Ende des Lesevorgangs enthält der Vector z also alle für eine weitere Bearbeitung erforderlichen Informationen über das Prädikat-Transitionsnetz.

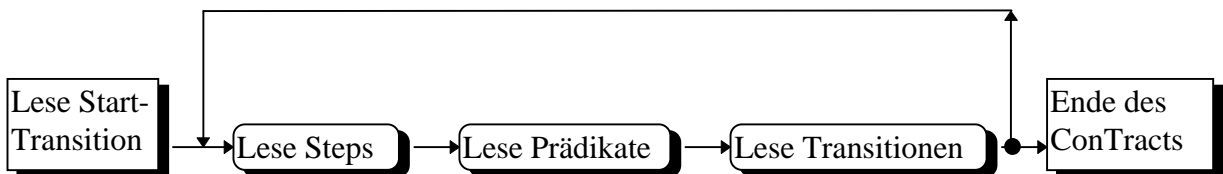


Abbildung 28: Einlesen eines Prädikat-Transitionsnetz

An dieser Stelle sei noch erwähnt, daß das Einlesen des Prädikat-Transitionsnetzes in den Hauptspeicher je nach Anzahl der Objekte und Auslastung des Systems einige Zeit dauern kann. Für die weitere Bearbeitung des Prädikat-Transitionsnetzes und vor allem für die Berechnung der graphischen Darstellung ist es jedoch unabdingbar, daß eine Struktur mit allen Objekten vollständig im Hauptspeicher verfügbar ist, die nach Belieben verändert werden kann. Würde man nur gerade gebrauchte Teile des Prädikat-Transitionsnetzes einlesen, so ginge der Überblick über die Gesamtstruktur verloren und außerdem hätte diese Vorgehensweise zur Folge, daß ständig auf den Storage Agent zugegriffen werden müßte, was zu höherer Netzbelastung und deutlichen Geschwindigkeitseinbußen führen würde.

6.2 Die Bearbeitung des PTN

Nachdem das Prädikat-Transitionsnetz eingelesen wurde, wird der Storage Agent nicht mehr benötigt und die Verbindung kann freigegeben werden. Jetzt befinden sich alle benötigten Informationen im Vector z im Hauptspeicher und müssen nur noch für den Benutzer aufbereitet werden, da diesen nicht alle eingelesenen Informationen interessieren. Die Do-Nothing-Steps zum Beispiel sind für die graphische Darstellung des ConTracts nicht notwendig und haben auch für den Benutzer keinerlei Bedeutung. Deshalb müssen sie genau wie die Abort-Pfade, die wie im vorigen Kapitel beschrieben die Planaritätskriterien verletzen, und noch einige andere Objekte, die nicht von Bedeutung sind, ausgefiltert werden. Dies kann allerdings erst jetzt geschehen, da für die Filterung die interne Struktur im Hauptspeicher benötigt wird.

Das Herausfiltern der nicht benötigten Objekte aus dem Vector z geschieht in mehreren Durchläufen und nach Art der Objekte getrennt. Es wurden mehrere Methoden dafür entwickelt, die im folgenden erläutert werden.

6.2.1 Die Eliminierung der True-Prädikate

True-Prädikate sind für die graphische Darstellung eines ConTracts uninteressant und werden deshalb vor der Berechnung des eigentlichen Graphen herausgefiltert. Dies übernimmt die Methode *true_filter()*, die einfach ohne Parameter aufgerufen wird und sämtliche True-Prädikate aus dem Vector z entfernt. Der Algorithmus funktioniert dabei wie folgt:

1. *Beginnend mit dem ersten Element von z wird geprüft, ob das aktuelle Element ein True-Prädikat ist.*
2. *Ist dies der Fall, so werden alle Vorgänger- und Nachfolger-Objekte des Elements gesucht. In der Nachfolgerliste der Vorgänger-Objekte und in der Vorgängerliste der Nachfolger-Objekte wird das Element gelöscht bzw. durch die neuen Vorgänger und Nachfolger ersetzt.*
3. *Das Element wird im Vector z gelöscht.*
4. *Dieses Verfahren wird so lange angewandt, bis das letzte Element des Vectors z erreicht wird. Nun sind alle True-Prädikate eliminiert.*

Anzumerken ist noch, daß die Eliminierung der Do-Nothing-Steps und der Transitionen, die im Graph generell nur dann dargestellt werden, wenn sie Beginn oder Ende eines Parallel-Konstrukts sind, auf genau die gleiche Art und Weise erfolgt. Auf eine Erläuterung der Methoden *dn_filter()* und *trans_filter()* wird deshalb an dieser Stelle verzichtet.

6.2.2 Das Ausblenden der Abort-Pfade

Abort-Pfade werden nur im Fehlerfall ausgeführt und verzweigen oft im Graphen rückwärts, um dort die Ausführung eines oder mehrerer Steps nochmals zu veranlassen. Diese Rückwärtssprünge bereiten jedoch Probleme bei der kreuzungsfreien Darstellung eines Graphen, wie wir im Kapitel „Beweis der Planarität der einzelnen Komponenten eines PTN“ bereits gesehen haben. Abort-Pfade von bestimmten Transaktionen können die Planaritätseigenschaften des Gesamtgraphen verletzen. Daher werden Abort-Pfade nur auf Wunsch des Benutzers dargestellt. Sie müssen also aus der normalen Darstellung erst einmal ausgeblendet werden.

Das Ausblenden der Abort-Pfade übernimmt die Funktion *abort_filter()*, die erst nach der erfolgreichen Eliminierung aller anderen nicht benötigten Objekte ausgeführt werden sollte, da sie auf die vorigen Routinen aufbaut. Der Algorithmus funktioniert wie folgt:

1. *Beginnend mit dem ersten Element von z wird geprüft, ob das aktuelle Element eine Transition vom Typ `TK_TA_ABORT` ist.*

2. Ist dies der Fall, so wird überprüft, ob der Pfad rückwärts verzweigt. Wenn ja, dann wird mit 3. fortgefahren um die Transition auszublenden, wenn nein, dann wird das nächste Element von z geprüft.
3. Verzweigt der Pfad rückwärts, so wird der komplette Pfad aus dem Vector z ausgehängt und die Vorgänger- und Nachfolgerlisten werden ähnlich wie beim dem Algorithmus zur Eliminierung der True-Prädikate aktualisiert. Die Informationen über den Rücksprung werden allerdings gespeichert, damit sie auf Wunsch wieder einblendet werden können.
4. Dieses Verfahren wird so lange angewandt, bis das letzte Element des Vector z erreicht wird. Nun sind alle Abort-Pfade ausgeblendet.

6.3 Die Einbettung des PTN in die Ebene

Hat man die Planarität eines Graphen bzw. eines ConTract-Skripts bewiesen, ist damit noch lange nicht geklärt, wie man das Skript in die Ebene einbetten muß, damit sich keine zwei Linien kreuzen. Zu diesem Zweck gibt es spezielle Einbettungs- oder Graphlayoutalgorithmen, die in günstigen Fällen die nötigen Berechnungen in quadratischer oder sogar in linearer Zeit durchführen können. Es gibt Dutzende dieser Algorithmen, die alle jedoch sehr komplex und sehr lang sind. Deshalb beschränken wir uns an dieser Stelle auf den Algorithmus von Hotz, dessen grundlegende Funktionsweise zum besseren Verständnis anhand eines Verbalalgorithmus erklärt wird. Auf einen Abdruck des vollständigen, wenig aussagekräftigen Listings wird verzichtet. Die Lösung des Einbettungsproblems wurde in dieser Diplomarbeit ohnehin durch einen selbstentwickelten Algorithmus durchgeführt, der im Anschluß an den Algorithmus von Hotz ausführlich diskutiert wird.

6.3.1 Der Algorithmus von Hotz

Der Algorithmus von Hotz beruht auf dem Satz von Kuratowski. Gegeben ist ein primitiver Graph G und (p,q) , eine beliebige Kante von G . Gesucht ist der Graph G_n , der in die Ebene E eingebettet werden kann. W bezeichnet einen Weg zwischen zwei Knoten.

Algorithmus:

Neben der Kante (p,q) gibt es mindestens zwei weitere Wege W_1 und W_2 , die die Punkte p und q verbinden und welche außer p und q keine Knotenpunkte gemeinsam haben. Weiter gibt es mindestens einen Weg W_3 , welcher einen inneren Knotenpunkt von W_1 mit einem inneren Knotenpunkt von W_2 verbindet und weder p noch q enthält. Konstruiere aus p , q , W_1 , W_2 und W_3 den Untergraphen U_1 .

Konstruiere eine (abbrechende) Folge von Untergraphen U_n von G und deren Einbettungen G_n in die Ebene E wie folgt:

Erzeuge einen Graph H_n durch Entfernen aller zu U_1 gehörenden Kanten aus G .

Erweitere U_n zu U_{n+1} und G_n zu G_{n+1} durch Hinzunahme der Knoten X und Y , die wie folgt gefunden werden:

- a) *Es gibt genau ein Gebiet T von U_n , auf dessen Rand sowohl X als auch Y liegen.*
- b) *X und Y sind in H_n durch einen Weg W verbunden, der keinen Knoten von U_n als inneren Knotenpunkt enthält.*

Wenn es in U_n eine Kante $k = (u,v)$ und in H_n einen u mit v verbindenden Weg W^ gibt, der keinen Knotenpunkt von U_n als inneren Knotenpunkt hat, dann konstruiere U_{n+1} aus U_n , indem man die Kante k in U_n löscht und an ihrer Stelle den Weg W^* einfügt.*

Entsprechend erhält man G_{n+1} aus G_n , indem man die Einbettung k der Kante k' durch die Einbettung W^ des Weges W^* ersetzt.*

Es ist klar, daß bei jeder der beiden obigen Operationen die Anzahl der eingebetteten Kanten zunimmt. Das Verfahren bricht ab, wenn keine der Operationen mehr ausgeführt werden kann. Das tritt spätestens dann ein, wenn H_n keine Kante mehr enthält. In diesem Falle gilt dann $U_n = G$ und G_n ist die gesuchte Einbettung von G in die Ebene E .

Bemerkung:

Bricht das Verfahren früher ab, so ist eine Einbettung von G in die Ebene E unmöglich, das heißt G ist nicht planar.

6.3.2 Ein eigener Algorithmus

Der Algorithmus von Hotz ist ein sehr allgemeiner Einbettungsalgorithmus und nur zur Einbettung planarer Graphen geeignet, da das Verfahren wie bereits schon erwähnt abbricht, falls eine kreuzungsfreie Einbettung nicht gefunden werden kann. Außerdem können bei diesem Algorithmus an den Außenseiten gebogene Kanten auftreten.

Da ConTract-Skripte, wie im vorigen Abschnitt gezeigt, nicht immer planar sein müssen und bei der Darstellung der ConTract-Skripte im Monitoring Interface gebogene Linien vermieden werden sollen, wurde ein eigener, speziell auf ConTract-Skripte zugeschnittener Algorithmus entwickelt, der aus mehreren Methoden besteht. Die grundlegende Funktionsweise des Algorithmus ist folgende:

1.) *Berechne die korrekte x -Position aller Elemente.*

Durchlaufe hierzu den Vector z , in den das ConTract-Skript eingelesen wurde, und verschiebe alle Elemente eines Teilbaumes in Abhängigkeit der Position der anderen Elemente des Teilbaumes.

Wiederhole dieses Verfahren für alle Teilbäume.

2.) *Berechne die korrekte y-Position aller Elemente.*

Durchlaufe hierzu den Vector z und überprüfe für jedes Element, ob Konflikte hinsichtlich ihrer y-Position mit anderen Elementen auftreten. Falls ja, verschiebe die konfliktbehafteten Elemente in positive y-Richtung und prüfe erneut.

Führe dieses Verfahren solange durch, bis keine Konflikte mehr auftreten.

Dies ist nur eine grobe Beschreibung des Algorithmus. Zum besseren Verständnis seiner Arbeitsweise werden im folgenden bei der Erklärung der einzelnen Methoden einige kleine graphische Beispiele gegeben, die alles etwas anschaulicher machen.

6.3.2.1 Die Methode calculate_x()

Die Methode calculate_x() ist für die Berechnung der korrekten x-Position der Knoten eines ConTract-Skriptes verantwortlich. Sie ist relativ kurz, da sie sich zur eigentlichen Berechnung der rekursiv implementierten Methode rec(id) bedient und davon ausgeht, daß bereits beim Einlesen des ConTract-Skriptes die Knoten ihrer Reihenfolge entsprechend in der Ebene angeordnet wurden.

```
public void calculate_x() {
    int i,j;
    Element e;

    for (i=0;i<z.size();i++)
    {
        e = (Element)z.elementAt(i);
        if (e.draw == true && e.par_begin==true)
            rec(e.id)
    }
}
```

Die Methode durchsucht den Vector z, in den, wie schon im vorigen Abschnitt erklärt, das ConTract-Skript mit allen benötigten Daten eingelesen wurde. Ist ein Knoten auf dem Bildschirm sichtbar (e.draw == true) und verzweigen von dem Knoten zugleich mehrere Kanten (e.par_begin == true), dann wird die ID des Knotens an die Methode rec(id) übergeben, die berechnet, ob ein Knoten in x-Richtung verschoben werden muß oder nicht.

```
public void rec(int id) {
    ...

    max_count = 0; // max. Breite eines Pfades

    e1 = (Element)z.elementAt(id); // lese Element id
    old_xpos = e1.xpos; // merke alte x-Position
```

```

for (i=0; i<e1.succ_list.size(); i++)           // durchsuche Nachfolgerliste
{
    e2 = (Element)z.elementAt(i);
    if (e2.par_begin == true)                 // bei weiterer Verzweigung
        rec(e2.id);                          // rekursiver Aufruf der Methode
    count++;                                  // erhöhe Breitenzähler

    if (e2.par_end == true)                   // ist die Verzweigung zu Ende ?
        new_xpos = e2.xpos;                  // merke neue x-Position
    ...
}

if (count > max_count)                         // ermittle die breiteste Stelle
    max_count = count;

size = new_xpos - old_xpos;                   // berechne Verschiebung
r_move = max_count - size;

...
moveTree_X(e2.id, r_move);                    // verschiebe Element + Zweig
}
}

```

Die Methode `rec(id)` berechnet in einem Zweig die Breite jedes einzelnen Pfades. Trifft sie während der Berechnung auf eine weitere Verzweigung, dann ruft sie sich selbst rekursiv auf und startet eine neue Berechnung. Am Ende der Berechnung ist genau bekannt, wieviele Elemente sich zum Beispiel in jedem Pfad einer Parallel-Verzweigung befinden, woraus man die Breite jedes Pfades und damit auch die korrekten x-Positionen der einzelnen Elemente des Pfades berechnen kann. Steht ein Element mit der `id` `e.id` an der falschen Position, dann wird es mit Hilfe der Methode `moveTree_X(e.id, r_move)` um `r_move` Positionen an seine richtige Position verschoben. Die Elemente im selben Zweig werden beim Aufruf der Methode übrigens mit verschoben.

Beispiel:

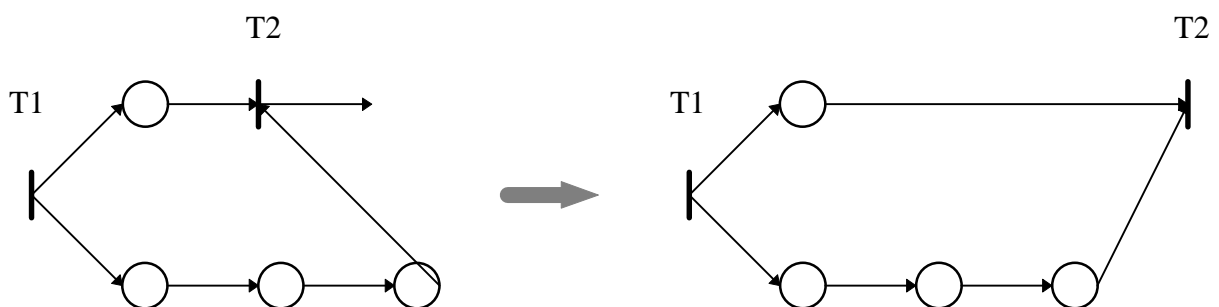


Abbildung 29: Beispiel für `calculate_x()`

Abbildung 29 zeigt ein Beispiel für die Wirkungsweise der Methode `calculate_x()`. Die Transition `T2` befindet sich nicht an der korrekten `x`-Position und wird daher um `r_count = 2` Positionen nach rechts verschoben.

6.3.2.2 Die Methode calculate_y()

Die Methode calculate_y() ist für die Berechnung der korrekten y-Position der Knoten eines ConTract-Skriptes verantwortlich. Die Methode überprüft für jedes Element des Vectors z, ob die y-Position des Elements möglicherweise Konflikte mit einem vorigen Element verursacht, das auf gleichem y-Niveau liegt und dessen x-Position kleiner ist. Ist dies der Fall, so wird das y-Niveau des vorigen Elements und aller Folgeelemente mit Hilfe der Methode moveTree_Y(id, 1) um eins erhöht. Das heißt, das Element und der komplette an diesem Element hängende Zweig wandert in der graphischen Darstellung um eine Ebene tiefer.

```
public void calculate_y() {
    Element e1,e2,e3;
    boolean stop;
    int i,j,k;
    ...

    old_loop = loop;                                // loop Zähler

    for(i=0;i<z.size();i++)                          // durchsuche Vector z
    {
        e1 = (Element)z.elementAt(i);                // lese Element e1 an der Stelle i

        for(j=0;j<z.size();j++)                      // zweite Schleife für Vector z
        {
            e2 = (Element)z.elementAt(j);            // lese Element e2 an der Stelle j
            ly_old = e2.ly;                           // speichere alte y-Position und id
            id_old = e2.id;

            if ( (e2.ly == e1.ly) && (e2.xpos < e1.xpos) ) // existiert Konflikt ?
            {
                stop=false;                           // ja, nicht abbrechen

                do
                {
                    if (e2.par_begin == true)          // Verzweigung ?
                    {
                        e2 = (Element)z.elementAt(r_int); // lese Element
                    }
                    else
                    {
                        integer = (Integer)e2.succ_list.elementAt(0);
                        r_int = integer.intValue();

                        if (r_int == -2)                // -2 bedeutet: kein Nachfolger
                            stop=true;                // abbrechen
                        else
                            e2 = (Element)z.elementAt(r_int); // lese Element
                    }
                }

                if (e2.ly < ly_old)                    // wenn y-Level kleiner,
```

```

    {
        stop=true;                // dann fertig
        loop++;                   // Anzahl der Durchläufe

        moveTree_Y(id_old, 1);    // verschiebe Element + Zweig
    }
} while (stop == false)        // fertig ?
}
}
}

```

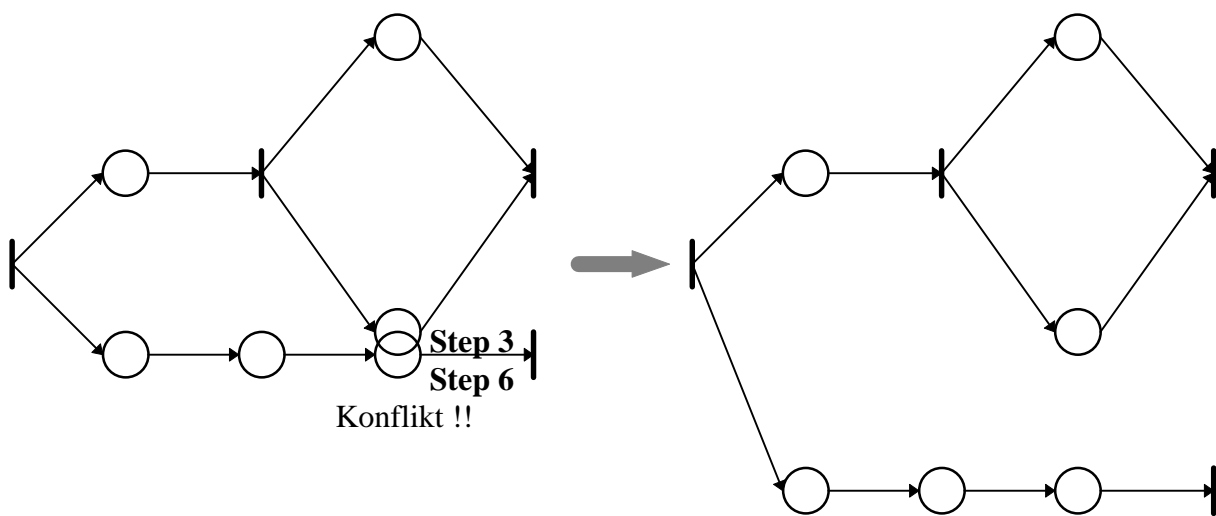
Beispiel:**Abbildung 30: Beispiel für calculate_y()**

Abbildung 30 zeigt ein Beispiel für die Wirkungsweise der Methode `calculate_y()`. Die Steps 3 und 6 befinden sich an derselben Position und stehen daher im Konflikt zueinander. Der Konflikt wird dadurch gelöst, daß der komplette Zweig mit Step 6 um eine Position in positiver y-Richtung nach unten verschoben wird.

Anmerkung:

Der Graph, der durch den in diesem Abschnitt beschriebenen Einbettungsalgorithmus aufgebaut wird, ist ein Streckengraph, d.h., er hat nur gerade Kanten. Besitzt das ConTract-Skript keine Transaktionen bzw. sind die Abort-Pfade ausgeblendet, so kann das Skript kreuzungsfrei dargestellt werden, bis auf eine Ausnahme:

Es kann theoretisch passieren, daß für eine kreuzungsfreie Darstellung ohne gebogene Kanten bestimmte Elemente sehr sehr weit auseinandergezogen werden müßten. Dadurch ginge jedoch der Überblick über den ConTract völlig verloren, weshalb in diesem Fall zugunsten der Übersichtlichkeit auf eine rein kreuzungsfreie Darstellung verzichtet wird.

Dieser Fall tritt jedoch relativ selten auf und die nicht kreuzungsfreie Darstellung einer oder zweier Kanten fällt kaum ins Gewicht.

6.3.3 Komplexitätsbetrachtung

Für die Berechnung der graphischen Darstellung war es wichtig und notwendig, daß dieser Vorgang so schnell wie möglich erfolgt, damit der Benutzer nicht zu lange vor dem Bildschirm warten muß. Gute, im Sinne von schnellen Algorithmen erledigen diese Aufgabe in maximal quadratischer Zeit, und auch der in dieser Diplomarbeit entwickelte, speziell auf ConTract-Skripte zugeschnittene Algorithmus hat die (Zeit-) Komplexität $O(n^2)$.

Dies ist auch leicht einzusehen, da in der Routine *calculate_x()* nur zwei for-Schleifen ineinander geschachtelt sind, die beide von 1 bis $n = z.size()$ laufen und die innere Schleife wird sogar noch von einer if-Bedingung eingeschränkt. Die Komplexität kann also hier nur maximal quadratisch sein. In der zweiten Routine *calculate_y()* sind ebenfalls zwei for-Schleifen, die von 1 bis $n = z.size()$ laufen, ineinander geschachtelt. In der inneren Schleife gibt es noch eine while-Schleife, die allerdings nur maximal über die Breite eines Parallel-Zweiges läuft, und das auch nur, wenn vorher eine if-Bedingung zu wahr evaluiert. Da die Breite eines Parallel-Zweiges immer konstant ist, kann bei der Komplexitätsbetrachtung die while-Schleife als konstanter Faktor vorgezogen werden, und es ergibt sich eine Komplexität von $c \cdot n^2$ mit c als konstantem Faktor. Auch hier ist also die Komplexität maximal quadratisch, woraus folgt, daß der gesamte Algorithmus eine Komplexität von $O(n^2)$ besitzt.

7 Oberflächengestaltung und Benutzungskonzept

Das Monitoring Interface ist eine der Schnittstellen des APRICOTS-Systems zum Benutzer, und für den Benutzer stellt sich ein System meist über seine Benutzerschnittstelle dar. Man könnte sogar sagen, für viele Benutzer ist die Benutzerschnittstelle weitgehend identisch mit dem System. Aus diesem Grund ist die benutzerfreundliche Gestaltung einer Benutzeroberfläche ein sehr wichtiger Punkt im Softwareentwicklungs-Zyklus. Der Anteil am gesamten Programmcode einer Anwendung für die Benutzeroberfläche wurde schon 1978 auf etwa 60 % geschätzt [Balz88], und die Tendenz ist weiter steigend. Das Ziel einer jeden Anwendungs-Entwicklung sollte also sein, eine Benutzeroberfläche zu entwickeln, die allen Anforderungen des Benutzers an die Bedienung des Systems genügt.

7.1 Kriterien der Benutzerfreundlichkeit

Eine Oberfläche, die den heutigen Anforderungen an die Softwareergonomie genügt, sollte nach DIN-Norm 66234/Teil 8 mindestens folgende Eigenschaften besitzen:

- Transparenz
- Ähnlichkeit
- Toleranz
- Konsistenz
- Unterstützung
- Flexibilität / Individualisierbarkeit

Transparenz:

Mit dem Begriff Transparenz ist gemeint, daß der Benutzer erkennen sollte, ob ein eingegebenes Kommando behandelt wird oder ob das System auf weitere Eingaben wartet. Außerdem sollte das System bei längeren Vorgängen Zwischenstandsmeldungen ausgeben.

- ⇒ Das Monitoring Interface gibt nach jedem Kommando eine Bestätigung aus, ob das Kommando ausgeführt wird oder nicht. Außerdem gibt es bei länger dauernden Vorgängen wie zum Beispiel beim Einlesen eines Prädikat-Transitionsnetzes oder bei der Berechnung des ConTract-Layouts in der Statuszeile Zwischenmeldungen aus, die über den Abarbeitungszustand des aktuellen Kommandos informieren.

Ähnlichkeit:

Das System sowie dessen Antwortverhalten sollte für den Benutzer transparent und konsistent sein, d.h., ähnliche Aktionen sollten zum Beispiel ähnliche Ausführungen bewirken.

- ⇒ Die Kommandos des Monitoring Interface wurden ähnlich ihrer realsprachlichen Verwendung benannt. So dient zum Beispiel das Kommando *start* zum Starten eines ConTract-Skripts, mit dem Kommando *stop* wird ein Vorgang gestoppt usw.

Toleranz:

Das System sollte sich dem Benutzer gegenüber tolerant verhalten, d.h., es darf nicht davon ausgegangen werden, daß der Benutzer keine für das System unsinnigen Kommandos auszuführen versucht.

- ⇒ Das Menü und die Button-Leisten des Monitoring Interface wurden so implementiert, daß dem Benutzer nur jeweils diejenigen Kommandos zur Verfügung stehen, die zum aktuellen Zeitpunkt aus Sicht des Systems Sinn machen bzw. durchführbar sind. Aus dem Aussehen eines Buttons oder eines Menüeintrages kann der Benutzer sofort erkennen, ob ein Kommando zum gegenwärtigen Zeitpunkt ausführbar ist oder nicht.

Konsistenz:

Bei der Darstellungsform für Einzelinformationen sollte auf Übereinstimmung mit eventuellen entsprechenden gedruckten Vorlagen oder Unterlagen geachtet werden. Sprache und begriffliche Komplexität des Dialoges sollten sich an den Kenntnissen des spezifischen Benutzerkreises orientieren.

- ⇒ Die Darstellung der Informationen des Monitoring Interface und die Dialogboxen orientieren sich so weit wie möglich an den anderen Komponenten des APRICOTS-Systems, so daß der Benutzer in einer homogenen und vertrauten Umgebung arbeiten kann.

Unterstützung:

Vom Benutzer sollten Dialoghilfen sowohl zu inhaltsbezogenen als auch zu vorgangsbezogenen Aktionen abrufbar sein. Das System sollte ferner den Benutzer so weit wie möglich führen, damit auch unerfahrene Benutzer mit dem System umgehen können.

- ⇒ Im Monitoring Interface ist ein Menüpunkt *Hilfe* bereits vorgesehen, seine Funktionalität wurde allerdings nicht implementiert, da der Aufwand dafür zu groß gewesen wäre und für diese Diplomarbeit davon ausgegangen werden konnte, daß nur erfahrene Benutzer mit dem System arbeiten werden. Für die zukünftige Entwicklung des APRICOTS-Systems wäre aber eine Implementierung von Hilfsfunktionen in allen Komponenten des Systems durchaus sinnvoll.

Flexibilität / Individualisierbarkeit:

Eine Oberfläche sollte unterschiedliche Möglichkeiten zu ihrer Bedienung vorsehen und die Abfolge einzelner Arbeitsschritte sollte so wenig wie möglich vorgegeben sein. Auch sollte die Oberfläche in gewissem Maße an die Bedürfnisse des jeweiligen Benutzers angepaßt werden können.

- ⇒ Das Monitoring Interface bietet mehrere Möglichkeiten zur Ausführung eines Kommandos. Befehle können zum Beispiel mit der Maus wahlweise über ein Menü oder schnell und einfach durch einen Klick auf eine Button-Leiste ausgewählt werden. Auf eine Auswahl der Befehle über die Tastatur wurde verzichtet, da dies von der derzeitigen Java Version noch nicht auf allen Plattformen unterstützt wird. Im ConTract-Skript Fenster kann der Benutzer über das Menü Optionen wählen, zu welchen Elementen auf dem Bildschirm die ID eingeblendet werden soll, außerdem kann er Elemente auf dem Bildschirm hin und her schieben. Damit kann er die Darstellung eines ConTract-Skriptes nach seinen Wünschen beeinflussen.

7.2 Die Bedienung des Monitoring Interface

Nach dem Start öffnet sich ein etwa 700 x 400 Bildpunkte großes Hauptfenster in der Mitte des Bildschirms, in dem der Benutzer seine Eingaben vornehmen kann.

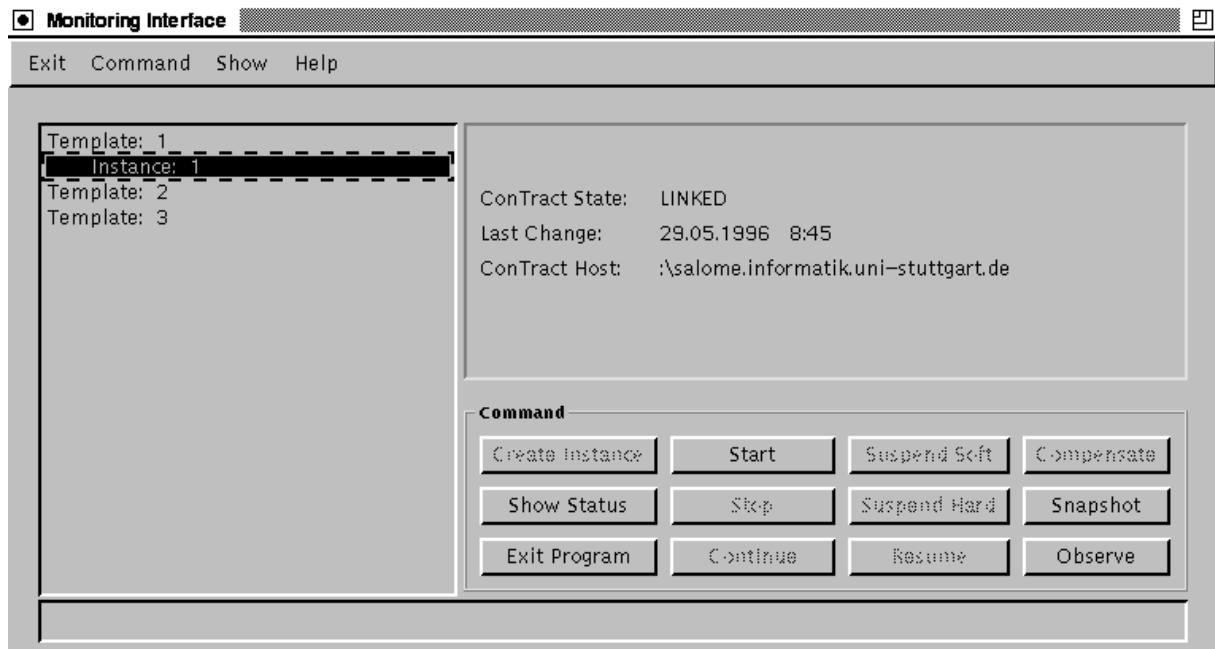


Abbildung 31: Das Hauptfenster des Monitoring Interface

Im linken Teil des Hauptfensters befindet sich eine Listbox, in die unmittelbar nach dem Start ohne Aufforderung durch den Benutzer alle ConTract-Templates eingelesen und angezeigt werden. Der Benutzer kann mit der Maus ein Template anklicken und erhält dann nach kurzer

Verzögerung, die durch das Suchen im Namensservice bedingt ist, eine Liste aller Instanzen, die zu dem angeklickten Template existieren.

Klickt der Benutzer nun eine ConTract-Instanz an, so erscheinen im Textfeld rechts oben im Hauptfenster einige grundlegende Informationen zu dieser Instanz. Diese Informationen beinhalten zum Beispiel den aktuellen Status der Instanz, das Datum, wann sie zuletzt bearbeitet wurde, auf welchem Host die Engine läuft usw.

Über die Button-Leiste in der rechten unteren Hälfte des Hauptfensters hat der Benutzer Zugriff auf Funktionen, mit denen er die Instanz bzw. den ConTract manipulieren kann. Ebenso kann er dort das Anlegen einer neuen Instanz initiieren oder sich einen ConTract synchron oder asynchron anzeigen lassen. Alle diese Funktionen können übrigens auch über die Menüleiste ausgeführt werden. Die Wahl bleibt dem Benutzer überlassen.

In der Statusleiste am unteren Rand des Hauptfensters werden jeweils aktuelle Meldungen ausgegeben bzw. Fehler, die nach dem Aufruf einer Funktion durch den Benutzer aufgetreten sind, näher erläutert.

7.3 Funktionen zur Manipulation der ConTracts

Die Funktionen zur Manipulation eines ConTracts können grundsätzlich nur auf Instanzen ausgeführt werden, niemals auf Templates. Ein Template ist nämlich nur eine Art Schablone, aus der von einem Compiler Instanzen erzeugt werden. Diese können dann individuell bearbeitet und verändert werden, ohne daß dies Auswirkungen auf das Original hat.

Eine Instanz kann aus einem Template durch Anklicken des Buttons *Create instance* erzeugt werden, woraufhin sich ein Dialogfenster öffnet, in das der Name der neuen Instanz eingegeben werden kann. Die Daten werden dann vom Monitoring Interface an den Monitor Agent geschickt, der für die weitere Abwicklung und insbesondere für die Kommunikation mit dem ConTract Manager verantwortlich ist. Es ist zu beachten, daß der Aufruf von *Create instance* synchron erfolgt, was bedeutet, daß keine weiteren Aktionen unternommen werden können, solange der Vorgang nicht abgeschlossen ist.

Start:

Mit Hilfe des Start-Kommandos kann die Bearbeitung eines ConTracts bzw. einer ConTract-Instanz gestartet werden. Dies geht natürlich nur, wenn zuvor durch einen Mausklick in die Listbox eine gültige Instanz selektiert wurde und die Instanz noch nicht in Bearbeitung, das heißt im Zustand *'Initial'* ist. Nach erfolgreicher Ausführung des Kommandos befindet sich der ConTract dann im Zustand *'Running Forward'*.

Stop:

Durch das Kommando *'Stop'* wird die Bearbeitung einer bereits gestarteten Instanz angehalten. Dieses Kommando sollte nur benutzt werden, wenn die Ausführung des ConTract für kurze Zeit unterbrochen werden soll, da sich das System nach einem Stop in dem nicht persistenten Zustand *'Stopped'* befindet.

Continue:

Mit diesem Kommando kann nach einem 'Stop' die Bearbeitung eines ConTracts wieder aufgenommen werden. Der ConTract befindet sich nach erfolgreicher Ausführung des Kommandos somit wieder in dem Zustand '*Running Forward*'.

Suspend:

Das Kommando 'Suspend' wird dazu benutzt, die Bearbeitung eines ConTracts für längere Zeit zu unterbrechen. Hierfür wird die Bearbeitung im Gegensatz zu dem Kommando 'Stop' in einem konsistenten Zustand angehalten und persistent abgespeichert. Das Kommando 'Suspend' kennt zwei unterschiedliche Modi:

- *Suspend Hard*: Bei einem 'Suspend Hard' werden alle Transaktionen sofort abgebrochen und zurückgesetzt. Dann wird das System in dem persistenten Zustand '*Suspended*' angehalten.
- *Suspend Soft*: Bei einem 'Suspend Soft' werden im Gegensatz zu dem Kommando 'Suspend Hard' alle angefangenen Transaktionen noch zu Ende gebracht, bevor das System in dem persistenten Zustand '*Suspended*' angehalten wird. Dies hat natürlich zur Folge, daß das Anhalten des Systems wesentlich länger dauern kann, da sich das Beenden einzelner Transaktionen über einen nicht vorher definierbaren Zeitraum erstrecken kann.

Resume:

Mit diesem Kommando kann nach einem 'Suspend' die Bearbeitung eines ConTracts wieder aufgenommen werden. Der ConTract befindet sich nach erfolgreicher Ausführung des Kommandos wieder im Zustand '*Running Forward*'.

Compensate:

Mit dem Kommando 'Compensate' kann ein ConTract kompensiert (zurückgesetzt) werden, d.h., alle während der Ausführung eines ConTracts durchgeführten Aktionen werden rückgängig gemacht, sofern dies möglich ist. Nach erfolgreicher Kompensation befindet sich der ConTract wieder in seinem Anfangszustand bzw. in einem dazu logisch äquivalenten Zustand.

Bei der Ausführung dieses Kommandos muß beachtet werden, daß die eigentliche Kompensation erst durch ein 'Prepare Compensation' vorbereitet werden muß. Diese Arbeit übernimmt jedoch der Monitor Agent für den Benutzer, d.h. bei einem Aufruf des Kommandos 'Compensate' durch den Benutzer leitet der Monitor Agent zwei Kommandos, nämlich 'Prepare Compensation' und 'Compensate' an den ConTract Manager weiter. Nach erfolgreicher Ausführung der Kompensation befindet sich der ConTract in dem Zustand '*Compensated*'.

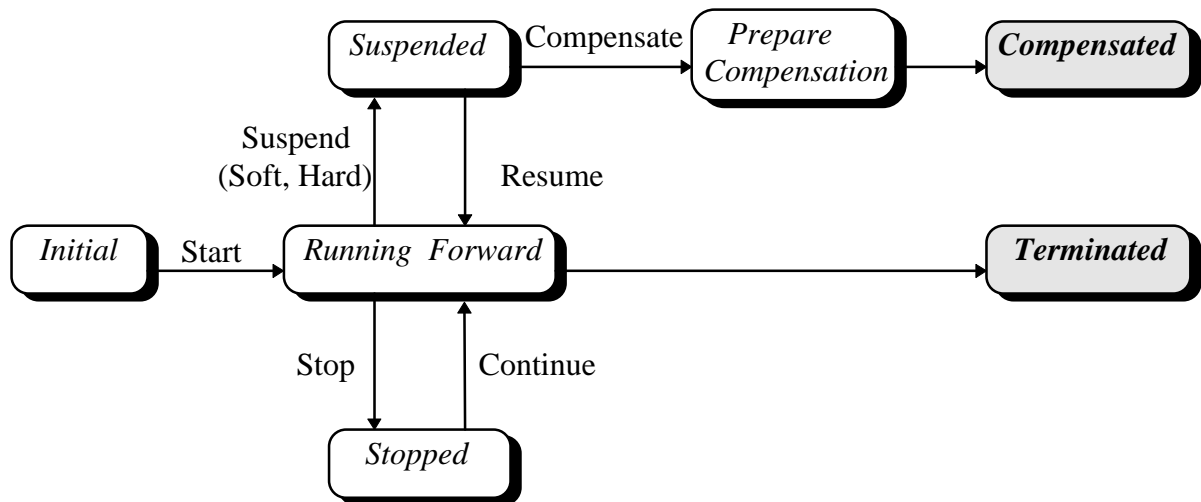


Abbildung 32: Zustände eines ConTracts vor und während der Bearbeitung

Abbildung 32 verdeutlicht in einer etwas vereinfachten Grafik nochmals die Zustände, die ein ConTract während der Bearbeitung durchlaufen kann. Anfangs ist der ConTract immer im Zustand *Initial*. Nach erfolgreichem Start geht er in den Zustand *Running Forward* über, von dem aus er durch ein *Suspend* bzw. *Stop* in die Zustände *Suspended* bzw. *Stopped* versetzt werden kann. Diese beiden Zustände können durch die Kommandos *Resume* bzw. *Continue* wieder aufgehoben werden. Eine Kompensation wird aus dem Zustand *Suspended* heraus durch das Kommando *Compensate* initiiert, worauf der ConTract in den Zustand *Prepare Compensation* und anschließend in den Endzustand *Compensated* übergeht. Wird der ConTract normal beendet, so geht er in den Endzustand *Terminated* über.

7.4 Möglichkeiten der Kommunikation

Das Monitoring Interface bietet die Möglichkeit, ConTracts sowohl synchron als auch asynchron anzeigen zu lassen. Synchron bedeutet, daß stets der aktuelle Bearbeitungszustand überwacht (observe) und angezeigt wird, während bei der asynchronen Darstellung nur der momentane Stand angezeigt wird. Es wird sozusagen ein Bild (Snapshot) des Systems zu einem bestimmten Zeitpunkt gezeigt. Für beide Möglichkeiten werden die hierfür relevanten Informationen über den Monitor Agent eingelesen. Doch wie geht die Kommunikation mit dem Monitor Agent vor sich ?

Prinzipiell gibt es mehrere Möglichkeiten, die benötigten Informationen vom Monitor Agent einzulesen:

- Beim asynchronen Fall, dem Snapshot, ist die Vorgehensweise relativ einfach. Nach der Aufforderung durch den Benutzer wird einmalig mit Hilfe von CORBA eine Verbindung zum Monitor Agent aufgebaut und die Informationen werden gelesen. Danach kann die Verbindung abgebrochen werden, da beim Snapshot der Abarbeitungszustand des ConTract-Skripts nicht nachgeführt wird, also keine weiteren Informationen mehr benötigt werden.

- Der synchrone Fall ist ungleich schwieriger. Da alle Änderungen am ConTract-Skript synchron, d.h., möglichst ohne Verzögerung nachgeführt werden sollen, stellt sich die Frage, wie mit dem Monitor Agent kommuniziert werden kann, so daß alle Änderungen sofort bemerkt werden können. Hierzu bieten sich zwei Möglichkeiten an:

Polling

Beim Polling agiert das Monitoring Interface als Client und fragt ständig in genau definierten Zeitabständen beim Monitor Agent (dem Server) nach, ob neue Informationen vorliegen. Wenn ja, dann werden diese abgeholt, wenn nein, dann war die Anfrage umsonst.

Der Nachteil dieser Vorgehensweise ist offensichtlich: Da Änderungen im ConTract-Skript teilweise in sehr kurzen Abständen auftreten können, muß natürlich das Abfrageintervall entsprechend klein gewählt werden. Andererseits handelt sich bei den Vorgängen um langlebige Abläufe, bei denen Stunden und Tage vergehen können, ehe etwas passiert. Das heißt, das Monitoring Interface fragt während dieser Zeit ständig umsonst nach und das Netz wird unnötigerweise belastet. Außerdem wird viel Rechenzeit für das Polling verbraucht, die sinnvoller eingesetzt werden kann.

Callbacks

Bei der Verwendung von Callbacks kann sich der Client, also das Monitoring Interface, beim Server registrieren lassen, und eine Art Rückruf vereinbaren. Das heißt, der Monitor Agent übermittelt dem Monitoring Interface eine Nachricht, sobald er neue Informationen vorliegen hat. Das Monitoring Interface holt sich diese dann im Gegenzug ab, sobald es Zeit für die Bearbeitung des Calls hat.

Diese Methode ist wesentlich besser zur Durchführung der vorliegenden Aufgabe geeignet als das Polling, da hier nur dann eine Kommunikation stattfindet, wenn wirklich neue Informationen vorliegen. Aus diesem Grund wurde der synchrone Fall der Darstellung eines ConTract-Skripts mit Hilfe von Callbacks realisiert.

7.4.1 Die synchrone Darstellung (Observe)

Die synchrone Darstellung wird durch einen Mausklick auf den Button 'Observe' eingeleitet. Dadurch wird eine Verbindung zum Storage Agent aufgebaut und das ConTract-Skript wird in Form eines Prädikat-Transitionsnetzes eingelesen. Dieser Vorgang wurde bereits genauer in Kapitel 4 beschrieben. Nach dem Einlesen des Skriptes wird intern der zugehörige Graph aufgebaut und beim Monitor Agent ein Callback für neu eintreffende Zustandsmeldungen registriert. Außerdem wird beim ersten Verbindungsaufbau der aktuelle Bearbeitungsstand des ConTracts abgeholt. Bereits abgearbeitete Elemente des Skriptes und die ausgewählten Pfade werden farbig markiert und der ConTract wird nun auf dem Bildschirm dargestellt. Wenn sich am aktuellen Bearbeitungsstand des ConTracts etwas ändert, d.h., ein Call vom Monitor Agent empfangen wird, dann wird auf diesen sofort reagiert und die Änderungen werden am Bildschirm nachgeführt, so daß der Benutzer stets über den aktuellen Stand auf dem Laufenden ist. Über die Menüzeile des ConTract-Fensters kann der Benutzer die schon aus dem vorigen Abschnitt bekannten Funktionen zur Manipulation des ConTracts ausführen. Es

sind übrigens nur die Funktionen anwählbar, die beim aktuellen Bearbeitungszustand des ConTracts Sinn machen. Weiterhin kann der Benutzer mit der linken Maustaste Elemente des ConTract-Graphen ergreifen und seinem Wunsch entsprechend verschieben. Wird ein Element, wie z.B. ein Step, dagegen mit der rechten Maustaste angeklickt, so öffnet sich nach kurzer Wartezeit ein Fenster, das nähere Informationen zu diesem Element enthält.

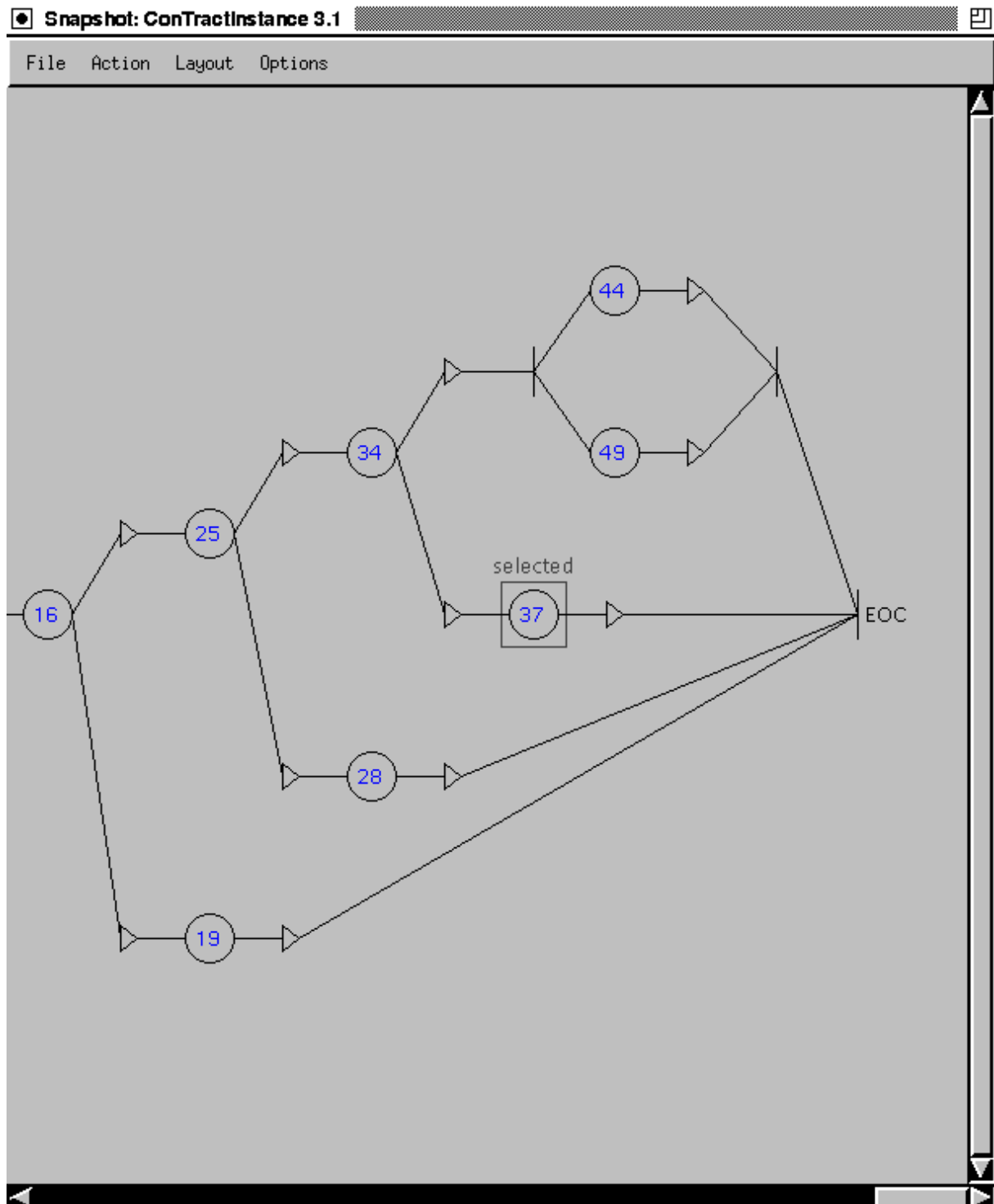


Abbildung 33: Synchroner Darstellung eines ConTracts

7.4.2 Die asynchrone Darstellung (Snapshot)

Die asynchrone Darstellung eines ConTracts wird durch einen Mausklick auf den Button 'Snapshot' eingeleitet und ist mit der synchronen Darstellung fast identisch. Der einzige Unterschied besteht darin, daß kein Callback registriert werden muß, da Veränderungen am Bearbeitungszustand des ConTracts nicht auf dem Bildschirm nachgeführt werden. Das heißt, der 'Snapshot' des ConTracts veraltet mit zunehmender Zeit. Dies ist jedoch in diesem Fall nicht von Bedeutung, da die asynchrone Darstellung lediglich dazu dient, sich kurz einen Überblick über den ConTract bzw. über dessen Bearbeitungsstand zu verschaffen.

Wie auch bei der synchronen Darstellung kann der ConTract über das Menü des Fensters manipuliert werden. Auch können Elemente wieder mit der linken Maustaste verschoben werden und durch einen Klick mit der rechten Maustaste erhält man weiterführende Informationen.

7.5 Fehlertoleranz und Restart-Verhalten

Eine wichtige Eigenschaft des APRICOTS-Systems ist seine Fehlertoleranz. Damit ist nicht etwa das fehlertolerante Verhalten des Systems gegenüber Benutzereingaben gemeint, sondern vielmehr das Verhalten bei kritischen Systemfehlern oder dem Wegfall einzelner APRICOTS-Komponenten. In einem solchen Fall ist gefordert, daß sich das System nach Beseitigung der Störung oder nach einem Neustart wieder in einem konsistenten Zustand befindet und auch alle Daten, die bis kurz vor der Störung bearbeitet wurden, noch erhalten sind. Da auch das Monitoring Interface mit Komponenten des APRICOTS-Systems kommuniziert, soll an dieser Stelle untersucht werden, wie sich ein Ausfall dieser Komponenten oder spezieller Dienste auf das Monitoring Interface auswirkt.

7.5.1 Ausfall des Naming Service

Das Monitoring Interface benutzt den Naming Service von CORBA, um andere Komponenten im Namensraum von APRICOTS aufzufinden und anzusprechen. Ist dieser Dienst nicht verfügbar, so kann keine Kommunikation stattfinden und das Monitoring Interface ist nicht einsatzfähig. Da gleich beim Start des Monitoring Interface der Naming Service dazu benutzt wird, den zum Benutzer gehörenden Monitor Agent zu finden, kann sich der Benutzer erst gar nicht in das System einloggen, wenn der Naming Service nicht ordnungsgemäß funktioniert. Fällt der Naming Service erst zu einem späteren Zeitpunkt aus, dann läuft die Oberfläche zwar weiter, es werden jedoch alle Kommandos, die irgendwie den Naming Service betreffen, solange mit einer Fehlermeldung quittiert, bis dieser wieder verfügbar ist.

7.5.2 Ausfall des Monitor Agent bzw. der Engine

Ein Ausfall des Monitor Agents ist gleichbedeutend mit einem Ausfall des ConTract-Managers bzw. der ConTract-Engine. In diesem Fall kann der Benutzer keine Kommandos, die eine Instanz betreffen, mehr ausführen. Es können insbesondere keine zu einem Template gehörigen Instanzen eingelesen und auch keine neuen Instanzen erzeugt werden. Alle Kommandos zur Manipulation eines ConTracts werden ignoriert bzw. mit einer

Fehlermeldung quittiert. Beim synchronen Fall der ConTract-Skript Darstellung werden keine Veränderungen mehr nachgeführt, da vom Monitor Agent keine Nachrichten mehr eintreffen. In einem solchen Fall sollte das aktuelle bzw. alle betroffenen Fenster geschlossen und erst dann wieder geöffnet werden, wenn alle Komponenten wieder verfügbar sind. Dies wird vom Monitoring Interface nämlich nicht automatisch erkannt, da sich der Monitor Agent nicht beim Monitoring Interface meldet, wenn er wieder einsatzfähig ist.

7.5.3 Ausfall des Storage Agents

Bei einem Ausfall des Storage Agents kann das Monitoring Interface keine ConTract-Skripte mehr einlesen und diese daher verständlicherweise auch nicht auf dem Bildschirm darstellen. Theoretisch funktionieren jedoch alle Funktionen zur Manipulation eines ConTracts. Theoretisch allerdings nur deshalb, weil zwar ein Kommando angenommen und über den Monitor Agent an den ConTract Manager weitergeleitet wird, von diesem aber dann eine Fehlermeldung zurückkommt, da auch dieser auf das Vorhandensein und Funktionieren eines Storage Agents angewiesen ist.

7.5.4 Restart-Verhalten

Stürzt aus irgendeinem Grund das Monitoring Interface selbst ab, so hat dies keine größeren Auswirkungen auf das Gesamtsystem. Da das Monitoring Interface selbst keine Daten speichert, mußte auch nicht auf eine persistente Datenhaltung geachtet oder ein Recovery implementiert werden. Nach einem Restart des Monitoring Interface werden alle Daten neu eingelesen und von neuem verarbeitet. Ein Restart unterscheidet sich also nicht von einem normalen Start.

Bei einem Restart des ConTract Managers bzw. des Monitor Agents verhält sich der Fall etwas anders. Der Wegfall bzw. zumindest das Wiederanlaufen dieser Komponenten wird vom Monitoring Interface nicht automatisch registriert, so daß sich die Oberfläche erst dann wieder in einem aktuellen und konsistenten Zustand befindet, wenn der Benutzer alle vor dem Absturz der anderen Komponenten aktiven Fenster geschlossen und danach bei Bedarf wieder geöffnet hat. Erst dann werden alle Daten neu eingelesen und aktualisiert. Bei der synchronen Darstellung eines ConTract-Skripts wird zuerst ein Snapshot ausgeführt, um vom Monitor Agent ein aktuelles Bild des Zustandes nach dem Absturz zu erhalten. Danach wird der schon bekannte Callback-Mechanismus installiert, um über alle zukünftigen Änderungen informiert werden zu können. Ab diesem Zeitpunkt stellt sich das Monitoring Interface dem Benutzer wieder so wie vor dem Absturz dar und der Benutzer kann normal weiterarbeiten.

8 Zusammenfassung und Ausblick

Das Ziel dieser Diplomarbeit war es, für das APRICOTS-System eine Oberfläche zur Manipulation und Überwachung der ConTracts zur Verfügung zu stellen. Hierzu war es erforderlich, mit Hilfe des Storage Agents Prädikat-Transitionsnetze einzulesen, die dann anschließend ausgewertet und aufbereitet werden mußten. Vor allem die graphische Darstellung der ConTracts war eine komplexe und schwierige Aufgabe, da einige nicht ganz triviale Probleme der Graphentheorie gelöst werden mußten. Durch die Entwicklung eines eigenen, speziell auf ConContract-Skripte bzw. Prädikat-Transitionsnetze in der vorliegenden Form zugeschnittenen Algorithmus konnte diese Aufgabe jedoch auch unter Performance-Gesichtspunkten gut gelöst werden.

Leider standen zum Zeitpunkt der Fertigstellung dieser Diplomarbeit noch nicht alle Komponenten des APRICOTS-Systems zur Verfügung bzw. waren nur eingeschränkt lauffähig. Insbesondere konnte das Definition Interface noch nicht benutzt werden, so daß keine ConContract-Skripte eingegeben werden konnten. Diese mußten in mühevoller und sehr fehlerträchtiger Kleinarbeit von Hand erzeugt werden, weshalb man sich auch auf drei ConTracts beschränkte, die für den Test des Systems benutzt wurden.

Mit den vorliegenden ConTracts funktionierte das Monitoring Interface einwandfrei zusammen, insbesondere konnten auch die durch eine Kompensation veränderten Skripte eingelesen und fehlerfrei dargestellt werden. Da jedoch längst nicht alle Fälle ausprobiert werden konnten (es gibt unendlich viele Möglichkeiten) und da die vorliegenden Skripte einige Konstrukte, die theoretisch unterstützt werden müßten wie zum Beispiel CASE-Blöcke, gar nicht enthielten, kann nicht gesagt werden, wie sich das Monitoring Interface bei der Darstellung dieser Konstrukte verhält. Theoretisch jedenfalls kann das Monitoring Interface mit allen für das APRICOTS-System vorgesehenen Konstrukten umgehen, bis auf die Parforeach-Anweisung, da diese auch nicht von der ConContract Engine unterstützt wird.

Für die Zukunft könnte man in Betracht ziehen, APRICOTS oder auch speziell das Monitoring Interface mit Mechanismen zur Sicherheitskontrolle auszustatten. Beim Start des Monitoring Interface ist bereits eine rudimentäre Passwortabfrage implementiert. Diese ist allerdings noch nicht richtig aktiviert und hat derzeit keinerlei Auswirkungen auf das System.

Außer einer Identifizierung durch Paßwörter könnte man noch daran denken, die ConTracts in Zugriffsgruppen einzuteilen oder mit bestimmten Benutzerrechten wie read, write oder execute auszustatten. Diese Rechte müßten jedoch alle Komponenten des APRICOTS-Systems unterstützen und die einzelnen Komponenten müßten sicherheitsrelevante Daten untereinander austauschen können. Da die einzelnen Komponenten in einer verteilten Umgebung ablaufen, sollten diese Daten auch nicht unverschlüsselt übers Netz gehen. Speziell die Programmiersprache Java wird in naher Zukunft Methoden und Klassen zur Verschlüsselung bereitstellen, die eine Programmierung dieser Sicherheitsmechanismen deutlich vereinfachen werden.

Abschließend bleibt noch zu wünschen, das das APRICOTS-Projekt nach seinem Abschluß als Erfolg gewertet werden wird, und daß die vorliegende Diplomarbeit ihren Teil dazu beigetragen hat.

Anhang

A Glossar

ACID:	Eigenschaften einer Transaktion: Atomicity, Consistency, Isolation, Durability.
APRICOTS:	A Prototype Implementation Of A ConTract System
Browser:	Programm zum Anzeigen von WWW-Seiten
CGI:	Common Gateway Interface, Schnittstelle zur Ausführung von Skripten
ConTract:	Langlebiger Ablauf im ConTract-Modell
ConTract-Modell:	Eine Erweiterung des klassischen Transaktionsmodells [ReuWä90]
CORBA:	Common Objekt Request Broker Architecture von der OMG
HTML:	Hypertext Markup Language, Seitenbeschreibungssprache für das WWW
IDL:	Interface Definition Language zur Beschreibung von Schnittstellen
Instanz:	Eine Art Kopie eines Templates, die vom ConTract Manager bearbeitet wird.
IPVR:	Institut für Parallele und Verteilte Höchstleistungsrechner der Uni Stuttgart
JAVA:	Programmiersprache von der Firma SUN Microsystems
JDK:	Java Development KIT der Firma SUN Microsystems
JIT:	Just In Time Compiler, beschleunigt JAVA-Programme
JWS:	Java WorkShop der Firma SUN Microsystems
Kontext:	Verwaltet die Daten der Steps [BuRai90]
NS:	Naming Service, ein CORBA-Service zur Verwaltung von Namen
OMG:	Object Management Group
ORB:	Object Request Broker, wichtiger Teil von CORBA
Orbix:	CORBA Implementierung der Firma IONA
OTS:	Object Transaction Service von der OMG

Petri-Netz:	Modell zur Darstellung von asynchronen Ereignissen [Petri62]
Prädikat:	Element im ConTract-Modell zur Auswertung von Bedingungen
PTN:	Prädikat-Transitionsnetz, wird zur Darstellung von ConTracts benötigt
PROXY:	Von einem IDL-Compiler erzeugtes Stellvertreterobjekt
Repository:	Sammelstelle für Daten
Step:	Element im ConTract-Modell, entspricht einer Stelle in einem Petri-Netz
Stub:	Rudimentäre Implementierung einer Schnittstelle
Template:	Schablone eines ConTract-Skripts, wird mit dem Definition Interface erzeugt
TCP/IP:	Transmission Control Protokoll / Internet Protokoll, wird im Internet benutzt
Transaktion:	Atomare Einheit im ConTract-Modell
Transition:	Element im ConTract-Modell, entspricht einem Übergang in einem Petri-Netz
Workflow:	Hier: Beschreibung eines langlebigen Ablaufs (ConTracts)
WWW:	World Wide Web, graphischer Teil der Internets

B Fehler in Java

Bei der Arbeit mit dem JDK 1.0.2 für Java von Sun Microsystems unter den Betriebssystemen Solaris bzw. Windows 95 zeigte sich, daß Java noch mit einigen kleineren Fehlern behaftet ist und daß einige spezielle Eigenschaften von Java bei der Programmierung Schwierigkeiten bereiten. Diese Fehler werden im folgenden erläutert. Falls eine Lösung für das Problem gefunden wurde, wird diese gleich mit angegeben.

B.1 Fehler, die in der Sprache Java entdeckt wurden

- *Der Java Compiler kann gleiche Klassennamen, die in verschiedenen Packages verwendet werden, nicht unterscheiden.*

Folgender Java-Code verursacht einen Fehler während der Compilierung:

```
package myPackage;
import java.util.Vector;

public class Vector {

    public void Vector() {

        ...
    }
}
```

Erklärung: Es wird versucht, in dem Package myPackage eine Klasse Vector zu erzeugen, die in dem Package java.util.Vector bereits existiert. Der Compiler meldet deshalb einen Fehler, obwohl dieses Vorgehen durchaus zulässig wäre.

Lösung: Die Verwendung von gleichlautenden Klassennamen vermeiden, bis der Fehler in einer der nächsten Versionen behoben wird.

- *Die Klasse Scrollbar weist einige Fehler auf:*

Die Methode **Scrollbar.setVisible** liefert nach einer Resize-Operation falsche Werte zurück. Manchmal wird der Wert auch gar nicht verändert, obwohl die Größe der Scrollbar definitiv verändert wurde.

Die Methode **Scrollbar.setPageIncrement** setzt zwar den richtigen Wert, dieser wird aber bei einem PageUp oder PageDown ignoriert, bzw. es wird immer nur um den Wert 1 gescrollt.

Das Greifen des Rollbalken mit der Maus und anschließende Verziehen funktioniert nicht richtig. Läßt man den Rollbalken los, wird nicht immer sofort an die neue Position gescrollt, oft erst nach einem zweiten Mausklick auf die Scrollbar.

- *Die Methode `Frame.setResizable()` funktioniert nicht richtig:*

Der Aufruf der Methode **`Frame.setResizable(false)`**, der eine Änderung der Größe eines Fensters unmöglich machen soll, funktioniert nicht.

Lösung: Denkbar wäre, die Ressourcen von X-Windows gleich von Anfang an so zu setzen, daß ein Fenster nicht mehr vergrößert werden kann. Darauf wurde aber in der vorliegenden Diplomarbeit verzichtet. Stattdessen wurden alle Fenster mit Hilfe des `GridBagLayouts` so angelegt, daß sie ihren Inhalt an die jeweilige Fenstergröße anpassen.

- *Beim Hinzufügen von Komponenten arbeitet die Klasse `Container` fehlerhaft*

Wird mit Hilfe der Methoden versucht, eine Komponente zu einer Containerklasse hinzuzufügen bzw. zu entfernen, kann es passieren, daß der Bildschirminhalt nicht erneuert wird. Die alte (gerade entfernte) Komponente wird nach wie vor noch dargestellt, allerdings kann man sie nicht mehr aktivieren. Oder die neue (gerade hinzugefügte) Komponente ist noch nicht sichtbar, obwohl sie es eigentlich sein müßte.

Lösung: Nach der Verwendung der Container-Methoden **`add()`** bzw. **`remove()`** sollte man die Methoden **`Container.validate()`** bzw. **`Container.pack()`** aufrufen. Der Bildschirminhalt ist danach aktualisiert.

C Schnittstellen zum System

An dieser Stelle werden die Schnittstellen des Monitoring Interface zu den anderen Komponenten des APRICOTS-Systems beschrieben. Die Beschreibung beschränkt sich nur auf den Teil der Schnittstellen, die auch tatsächlich für das Monitoring Interface von Bedeutung sind.

C.1 basedef.idl

Das IDL-File basedef.idl enthält allgemeingültige Definitionen, die sowohl vom Monitoring Interface, als auch von anderen Komponenten des APRICOTS-Systems benutzt werden.

```

module APRICOTS_Definitions {

    // Mögliche Zustände einer ConTract - Instanz

    /*
    CTS_INITIAL:          ConTract Instanz ist neu angelegt
    CTS_LINKED:          ConTract Instanz ist an eine Engine gelinked
    CTS_RUNNING_FORWARD: ConTract Instanz wird ausgeführt
    CTS_SUSPENDING_SOFT: ConTract-Ausführung wird suspendiert (soft)
    CTS_SUSPENDING_HARD: ConTract-Ausführung wird suspendiert (hard)
    CTS_SUSPENDED:       ConTract-Ausführung ist suspendiert
    CTS_TERMINATED:      ConTract-Ausführung ist abgeschlossen
    CTS_PREPARED_COMPENSATION: Instanz ist bereit für Kompensation
    CTS_COMPENSATING:    Kompensation wird durchgeführt
    CTS_SUSPENDING_SOFT_COMPENSATION: Kompensation wird suspendiert (soft)
    CTS_SUSPENDING_HARD_COMPENSATION: Kompensation wird suspendiert (hard)
    CTS_SUSPENDED_COMPENSATION: Kompensation ist suspendiert
    CTS_COMPENSATED:     Kompensation ist abgeschlossen
    CTS_ILLEGAL:         Unzulässiger Zustand
    */

    enum CTState
    {
        CTS_INITIAL,
        CTS_LINKED,
        CTS_RUNNING_FORWARD,
        CTS_SUSPENDING_SOFT,
        CTS_SUSPENDING_HARD,
        CTS_SUSPENDED,
        CTS_TERMINATED,
        CTS_PREPARED_COMPENSATION,
        CTS_COMPENSATING,
        CTS_SUSPENDING_SOFT_COMPENSATION,
        CTS_SUSPENDING_HARD_COMPENSATION,
        CTS_SUSPENDED_COMPENSATION,
        CTS_COMPENSATED,
        CTS_ILLEGAL
    };
}

```

```
/**/ Dieser Abschnitt beschreibt Exceptions, die nach einem Fehler auftreten können ***/
/*****/

// Exception, wenn der Aufruf einer Methode abgelehnt wird

exception Reject
{
    string reason;
};

// Exception, aufgrund eines Datenbankfehlers

exception DBError
{
    string detail;
};

// Exception, weil ein angesprochenes Objekt nicht gefunden wurde

exception NotFound
{
    string detail;
};

// Exception, weil die Operation, die ausgeführt werden soll, zum momentanen Zeitpunkt
// nicht erlaubt ist.

exception AccessDenied
{
    string detail;
};

/**/          Dieser Abschnitt definiert häufig verwendete Typen          ***/
/*****/

// Definiert einen String für ein Objekt

typedef string ObjectString;

// Definiert einen Namen in der Punktnotation des APRICOTS-Naming Service und eine
// Sequenz

typedef string NameString;
typedef sequence<APRICOTS_Definitions::NameString> NameStringSeq;

// Identifikation für Skript-Objekte

struct UniquelIdentification
{
    unsigned long id;
    unsigned long parallelIndex;
};
```

```
// Identifikation für eine ConTract-Instanz

struct ConTractInstanceID
{
    unsigned long cid;
    unsigned long cidInst;
};

typedef sequence<APRICOTS_Definitions::ConTractInstanceID> CiidSeq;

// Identifikation für Transaktionen

typedef UniqueIdentification TransactionID;

// Identifikation für Prädikate

typedef UniqueIdentification PredicateID;

// Identifikation für Steps

typedef UniqueIdentification StepObjectID;

// Identifikation für Transitionen

typedef UniqueIdentification TransitionID;

// Struktur mit Informationen über eine Step-Instanz

struct StepInstance
{
    StepObjectID    soid;
    StepInstanceVersion version;
};
```

C.2 monitorInterface.idl

Dieses IDL-File beschreibt die Schnittstelle des Monitoring Interface zum Monitor Agent.

```
#include "basedef.idl"
#include "objectthread.idl"
#include "ct_engine.idl"

interface APRICOTS_MonitorInterface:APRICOTS_ObjectThread {

/**
    Dieser Abschnitt definiert häufig verwendete Typen
    ****/
/*****/

// Für das Monitoring Interface wichtige Zustände eines ConTracts

enum CTStatus { RUNNING, TERMINATED, COMPENSATED, SUSPENDED,
                RECOVERING, RECOVERED, UNKNOWN };

/* Momentaner Status einer Transaktion
```

```

BEGIN:          Transaktion läuft
ABORTED:       Transaktion abgebrochen
COMMITTED:     Ausführung der Transaktion beendet
*/

enum TStatus { BEGIN, ABORTED, COMMITTED };

/* Momentaner Status eines Steps

  ACTIVATED:    Step ist aktiv
  FINISHED:    Ausführung beendet
*/

enum StepStatus { ACTIVATED, FINISHED };

// Struktur zur Weitergabe von Informationen über eine Transaktion

struct TInfo
{
    ::APRICOTS_Definitions::TransactionID TAID;
    TStatus status;
};

// Struktur zur Weitergabe von Informationen über einen Step an das Monitoring Interface

struct StepInfo
{
    ::APRICOTS_Definitions::StepObjectID StepID;
    StepStatus status;
};

// Struktur zur Weitergabe von Informationen über ein Prädikat an das Monitoring Interface

struct PredInfo
{
    ::APRICOTS_Definitions::ConTractInstanceID ciid;
    ::APRICOTS_Definitions::PredicateID pid;
    boolean result;
};

// Sequenz von Steps

typedef sequence<StepInfo> StepInfoSeq;

// Sequenz von Transaktionen

typedef sequence<TInfo> TInfoSeq;

// Sequenz von Prädikaten

typedef sequence<PredInfo> PredInfoSeq;

/***/                               Methoden                               ***/

```

```

/*****/

/*
  Aufruf von Monitoring Interface: Starte die Bearbeitung eines ConTracts
  Parameter: ciid : ConTract instance identification
*/

void start(in ::APRICOTS_Definitions::ConTractInstanceID ciid)
  raises (::APRICOTS_Definitions::Reject);

/*
  Aufruf von Monitoring Interface: Stoppe die Bearbeitung eines ConTracts
  Parameter: ciid : ConTract instance identification
*/

void stop(in ::APRICOTS_Definitions::ConTractInstanceID ciid)
  raises (::APRICOTS_Definitions::Reject);

/*
  Aufruf von Monitoring Interface: Setze die Bearbeitung eines ConTracts nach einem
  Stop fort.
  Parameter: ciid : ConTract instance identification
*/

void cont(in ::APRICOTS_Definitions::ConTractInstanceID ciid)
  raises (::APRICOTS_Definitions::Reject);

/*
  Aufruf von Monitoring Interface: Suspendiere einen ConTract
  Parameter: ciid : ConTract instance identification
  Parameter: mode : SuspendMode -> hard or soft
*/

void suspend(in ::APRICOTS_Definitions::ConTractInstanceID ciid, in
APRICOTS_ConTractEngine::SuspendMode mode)
  raises (::APRICOTS_Definitions::Reject);

/*
  Aufruf von Monitoring Interface: Setze die Bearbeitung eines ConTracts nach einem
  Suspend fort.
  Parameter: ciid : ConTract instance identification
*/

void resume(in ::APRICOTS_Definitions::ConTractInstanceID ciid)
  raises (::APRICOTS_Definitions::Reject);

/*
  Aufruf von Monitoring Interface: Beginne Kompensation
  Parameter: ciid : ConTract instance identification
*/

void compensate(in ::APRICOTS_Definitions::ConTractInstanceID ciid)
  raises (::APRICOTS_Definitions::Reject);

/*
  Anfrage vom Monitoring Interface: Übergebe alle Instanzen eines Templates

```

```

Parameter: cid      : ConTract identification
Parameter: ciidSeq  : sequence of ConTract instance identifications
raises DBError     : problem reading
raises NotFound    : wrong cid
raises AccessDenied : session not valid or invalid access rights
*/

void getInstances( in unsigned long cid, out ::APRICOTS_Definitions::CiidSeq instances)
    raises ( ::APRICOTS_Definitions::DBError,
            ::APRICOTS_Definitions::NotFound,
            ::APRICOTS_Definitions::AccessDenied);

/*
Anfrage vom Monitoring Interface: Erzeuge eine neue Instanz aus einem Template
Parameter: cid : ConTract identification
Parameter: ciid : ConTract instance identification
raises DBError   : problem writing
raises NotFound  : wrong cid
raises AccessDenied : session not valid or invalid access rights
*/

void createNewInstance(in unsigned long cid, in string name, out
::APRICOTS_Definitions::ConTractInstanceID ciid)
    raises ( ::APRICOTS_Definitions::DBError,
            ::APRICOTS_Definitions::NotFound,
            ::APRICOTS_Definitions::AccessDenied);

/*
Anfrage vom Monitoring Interface: Übergebe aktuellen Status einer Instanz
Parameter: ciid : ConTract instance identification
Parameter: TAs  : Sequence of Transactions
Parameter: Steps : Sequence of Steps
Parameter: Preds : Sequence of Predicates
Parameter: status : The actual Contract status
*/

void getState(in ::APRICOTS_Definitions::ConTractInstanceID ciid,
    in boolean actual,
    out TAInfoSeq TAs, out StepInfoSeq Steps,
    out PredInfoSeq Preds, out CTStatus status)
    raises (::APRICOTS_Definitions::Reject);
};

```

C.3 storageagent.idl

Dieses IDL-File beschreibt die Schnittstelle des Monitoring Interface zum Monitor Agent.

```

#include "basedef.idl"
#include "ots.idl"

interface APRICOTS_StorageAgent
{

/**
        Dieser Abschnitt definiert häufig verwendete Typen
        ***/

```

```

/*****/

/*
  Struktur zur Übermittlung von Daten einer Transition

  Parameter: transitionID : identifiziert eindeutig eine Transition
  Parameter: talD         : Transaktion, innerhalb der der Step läuft
  Parameter: blockEndTransitionID : kind = TK_BLOCK_BEGIN
                                oder TK_PARFOREACH_BEGIN
                                oder TK_CT_BEGIN: blockEndTransitionID bezeichnet die Transition, die
                                das Ende des Blockes anzeigt.
  Parameter: expectedTokens : Benötigte Tokens, damit die Transition feuert
  Parameter: kind           : Typ der Transition
*/

struct Transition
{
  ::APRICOTS_Definitions::TransitionID transitionID;
  ::APRICOTS_Definitions::TransactionID talD;
  ::APRICOTS_Definitions::TransitionID blockEndTransitionID;
  unsigned long expectedTokens;
  TransitionKind kind;
};

// Sequenz von Transitionen

typedef sequence<Transition> TransitionSeq;

enum StepAddressType { SAT_NAME_STRING, SAT_CTXT_ELEMENT };

struct StepCtxtAddress
{
  string elementName;
  ::APRICOTS_Definitions::StepObjectID producer_soid;
};

union StepAddress switch (StepAddressType)
{
  case SAT_NAME_STRING : ::APRICOTS_Definitions::NameString nameString;
  case SAT_CTXT_ELEMENT: StepCtxtAddress stepCtxtAddress;
};

/*
  Struktur zur Übermittlung von Daten eines Steps

  Parameter: soid : identifiziert eindeutig den Step
  Parameter: stepName : der Name des Steps
  Parameter: stepAddr : Adresse des ORB Name Service in Punktnotation
  Parameter: compensation_soid : != NULL_STEPOBJECT, wenn Kompensationsstep
                                = NULL_STEPOBJECT, wenn kein "-"
  Parameter: talD : Transaktion, innerhalb der der Step läuft
*/

struct StepObject
{
  ::APRICOTS_Definitions::StepObjectID soid;
};

```

```

StepAddress          stepAddr;
string               stepName;
boolean              isTransactional;
::APRICOTS_Definitions::EntryInvariant entryInvariant;
::APRICOTS_Definitions::ExitInvariant exitInvariant;
StepObjectKind       kind;
::APRICOTS_Definitions::StepObjectID compensation_soid;
::APRICOTS_Definitions::TransactionID taID;
};

// Sequenz von Steps

typedef sequence<StepObject> StepObjectSeq;

// Typen von Prädikaten

enum PredicateKind
{
    PK_USERDEFINED,
    PK_TRUE,
    PK_RESULT
};

/*
    Struktur zur Übermittlung von Daten eines Prädikats

    Parameter: pid           : identifiziert eindeutig das Prädikat
    Parameter: kind          : Typ des Prädikats
    Parameter: truePath     : Nachfolger, wenn das Prädikat zu wahr evaluiert
    Parameter: falsePath    : Nachfolger, wenn das Prädikat zu falsch evaluiert
    Parameter: expression: kind = PK_USERDEFINED: vom Benutzer definiert
    Parameter: result       : kind = PK_RESULT: Result des Steps
*/

struct Predicate
{
    ::APRICOTS_Definitions::PredicateID pid;
    PredicateKind kind;
    ::APRICOTS_Definitions::TransitionID truePath;
    ::APRICOTS_Definitions::TransitionID falsePath;
    string expression;
    ::APRICOTS_Definitions::StepServerResult result;
};

// Sequenz von Prädikaten

typedef sequence<Predicate> PredicateSeq;

/*

    Struktur zur Übermittlung von Daten einer Transaktion

    Parameter: taID           : identifiziert eine Transaktion
    Parameter: taBeginTransitionID : Transition, die eine Transaktion einleitet

```

```

    Parameter: taCommitTransitionID      : Transition für Commit
    Parameter: taAbortTransitionID      : Transition nach einem Rollback
*/

struct Transaction
{
    ::APRICOTS_Definitions::TransactionID taID;
    ::APRICOTS_Definitions::TransactionID taBeginTransitionID;
    ::APRICOTS_Definitions::TransactionID taCommitTransitionID;
    ::APRICOTS_Definitions::TransactionID taAbortTransitionID;
};

// Sequenz von Transaktionen

typedef sequence<Transaction> TransactionSeq;

// Struktur für eine Session

struct SessionId
{
    long restartCounter;
    long sessionCounter;
};

// Status eines Session

enum SessionState {STATE_UNKNOWN, STATE_UNUSED, STATE_IDLE,
                  STATE_ACTIVE};

/**
    Administrative Routinen
    ****
    *****/

/*
    Beginne eine Session mit dem Storage Agent

    return SessionId : SessionID
    raises ::APRICOTS_Definitions::DBError: severe problem
    raises ::APRICOTS_Definitions::AccessDenied: currently access cannot be granted
*/

SessionId beginSession ()
    raises ( ::APRICOTS_Definitions::DBError, ::APRICOTS_Definitions::AccessDenied);

/*
    Beende eine Session.
    Wenn die Session noch aktiv ist, wird ein Rollback durchgeführt.
    Parameter: session: Session, die beendet werden soll.
    raises ::APRICOTS_Definitions::DBError: severe problem
    raises ::APRICOTS_Definitions::NotFound: session does not denote a valid session
*/

void freeSession(in SessionId session)
    raises ( ::APRICOTS_Definitions::DBError, ::APRICOTS_Definitions::NotFound );

```

```

/*
  Commit Session

  Parameter: session: identifiziert die Session
  raises ::APRICOTS_Definitions::DBError: severe problem
  raises ::APRICOTS_Definitions::NotFound: session does not denote a valid session
*/

void commitSession(in SessionId session)
  raises ( ::APRICOTS_Definitions::DBError, ::APRICOTS_Definitions::NotFound );

/*
  Rollback Session.

  Parameter: session: identifiziert die Session
  raises ::APRICOTS_Definitions::DBError: severe problem
  raises ::APRICOTS_Definitions::NotFound: session does not denote a valid session
*/

void rollbackSession(in SessionId session)
  raises ( ::APRICOTS_Definitions::DBError, ::APRICOTS_Definitions::NotFound );

/*
  Übergibt den Status der Session

  Parameter: session: identifiziert die Session
  return SessionState: Status der Session
  raises ::APRICOTS_Definitions::DBError: severe problem
  raises ::APRICOTS_Definitions::NotFound: session does not denote a valid session
*/

SessionState getSessionState(in SessionId session)
  raises ( ::APRICOTS_Definitions::DBError, ::APRICOTS_Definitions::NotFound );

/***/
                Methoden zum Lesen eines PTN                ***/
/***/

/*
  Liest alle Transitionen

  Parameter: session          : identifiziert die Session
  Parameter: ciid            : identifiziert die ConTract-Instanz
  Parameter: transitionSeq   : Alle Transitionen einer ConTract-Instanz
  raises ::APRICOTS_Definitions::DBError   : Problem beim Lesen der Transitionen
  raises ::APRICOTS_Definitions::NotFound  : - ConTract-Instanz hat keine Transition.
                                           Dies ist ein Problem des PTN
                                           - falsche CIID
  raises ::APRICOTS_Definitions::AccessDenied : Session ungültig oder es existieren
                                           keine Zugriffsrechte
*/

void getAllTransitions ( in SessionId session,

```

```

        in ::APRICOTS_Definitions::ConTractInstanceID ciid,
        out TransitionSeq transitionSeq )
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

/*
Liest eine Transition nach einem Prädikat.

Parameter: session      : identifiziert die Session
Parameter: ciid         : identifiziert die ConTract-Instanz
Parameter: transitionID : identifiziert gelesene Transition
Parameter: transition    : identifiziert die ConTract-Instanz
raises ::APRICOTS_Definitions::DBError      : Transition kann nicht gelesen werden
raises ::APRICOTS_Definitions::NotFound     : - Transition nicht gefunden
                                           - falsche CIID
raises ::APRICOTS_Definitions::AccessDenied : Session ungültig oder es existieren
                                           keine Zugriffsrechte
*/

void getTransition ( in SessionId session,
                   in ::APRICOTS_Definitions::ConTractInstanceID ciid,
                   in ::APRICOTS_Definitions::TransitionID transitionID,
                   out Transition transition )
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

/*
Liest die erste Transition (Start-Transition) eines ConTracts.

Parameter: session      : identifiziert die Session
Parameter: ciid         : identifiziert die ConTract-Instanz
Parameter: transition    : Alle Transitionen einer ConTract-Instanz
raises ::APRICOTS_Definitions::DBError      : Transition kann nicht gelesen werden
raises ::APRICOTS_Definitions::NotFound     : - Transition nicht gefunden
                                           - falsche CIID

raises ::APRICOTS_Definitions::AccessDenied : Session ungültig oder es existieren
                                           keine Zugriffsrechte
*/

void getStartTransition ( in SessionId session,
                        in ::APRICOTS_Definitions::ConTractInstanceID ciid,
                        out Transition transition )
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

/*
Liest alle Steps nach einer Transition.

Parameter: session      : identifiziert die Session
Parameter: ciid         : identifiziert die ConTract-Instanz

```



```

void getPredicates ( in SessionId session,
                    in ::APRICOTS_Definitions::ConTractInstanceID ciid,
                    in ::APRICOTS_Definitions::StepObjectID   soid,
                    out PredicateSeq  predicateSeq )
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

/*
Liest alle Transaktionen mit einer bestimmten Elternttransaktion.

Parameter: session      : identifiziert die Session
Parameter: ciid        : identifiziert die ConTract-Instanz
Parameter: parent      : = NULL_TRANSACTION: gibt alle Top-Level Transaktionen
                        : !=NULL_TRANSACTION: gibt alle Transaktionen zu dieser
                        :                    : Elternttransaktion zurück

Parameter: transactionSeq: Sequenz von Transaktionen
raises ::APRICOTS_Definitions::DBError   : Problem beim Lesen der Transaktion
raises ::APRICOTS_Definitions::NotFound  : - Transaktion hat keine Subtransaktionen
                        - falsche CIID
raises ::APRICOTS_Definitions::AccessDenied : Session ungültig oder es existieren
                        keine Zugriffsrechte
*/

void getTransactions ( in SessionId session,
                      in ::APRICOTS_Definitions::ConTractInstanceID ciid,
                      in ::APRICOTS_Definitions::TransactionID  parent,
                      out TransactionSeq  transactionSeq )

raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

/*
Liest den Status einer ConTract Instanz.

Parameter: session      : identifiziert die Session
Parameter: ciid        : identifiziert die ConTract-Instanz
Parameter: managerString : der ConTract Manager, der für die Ausführung dieser
                        : ConTract Instanz verantwortlich ist
Parameter: logFile      : LogFile für die ConTract Instanz

Parameter: state        : Neuer Status der ConTract instance
Parameter: timestamp    : Zeitpunkt, zu dem die letzte persistente Speicherung
                        : erfolgte

raises ::APRICOTS_Definitions::DBError   : Problem beim Lesen des Status
raises ::APRICOTS_Definitions::NotFound  : falsche CIID
raises ::APRICOTS_Definitions::AccessDenied: Session ungültig oder es existieren
                        keine Zugriffsrechte
*/

```

```
void getConTractState ( in SessionId session,
                      in ::APRICOTS_Definitions::ConTractInstanceId ciid,
                      out ::APRICOTS_Definitions::ObjectString  managerString,
                      out LogFile                               logfile,
                      out ::APRICOTS_Definitions::CTState       state,
                      out ::APRICOTS_Definitions::Timestamp     timestamp )
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);

ProcessingStatus getProcessingStatus( in SessionId session,
                                     in ::APRICOTS_Definitions::NameString agentName,
                                     in string                               taName)
raises (::APRICOTS_Definitions::DBError,
       ::APRICOTS_Definitions::NotFound,
       ::APRICOTS_Definitions::AccessDenied);
```

D Wichtige Konstanten im ConTract-Skript

D.1 Typen von Transitionen

Konstante	Wert
TK_NORMAL	0
TK_TA_BEGIN	1
TK_TA_COMMIT	2
TK_TA_ABORT	3
TK_CT_BEGIN	4
TK_CT_END	5
TK_BLOCK_BEGIN	6
TK_BLOCK_END	7
TK_PARFOREACH_BEGIN	8
TK_PARFOREACH_END	9
TK_COMPENSATION_BEGIN	10
TK_COMPENSATION_END	11
TK_NULL	12
TK_FATAL	13

Tabelle 2: Typen von Transitionen

D.2 Typen von Steps

Konstante	Wert
SK_USERDEFINED_SYNC	0
SK_USERDEFINED_ASYNC	1
SK_FORLOOP_INIT	2
SK_FORLOOP_INCREMENT	3
SK_PARFOREACH_BEGIN	4
SK_DO_NOTHING	5

Tabelle 3: Typen von Steps

D.3 Typen von Prädikaten

Konstante	Wert
PK_USERDEFINED	0
PK_TRUE	1
PK_RESULT	2

Tabelle 4: Typen von Prädikaten

D.4 Status einer Transaktion

Konstante	Wert
ABORTED	0
COMMITTED	1
ACTIVE	2
PREPARED	3
UNKNOWN	4

Tabelle 5: Status einer Transaktion

E Das Makefile für das Monitoring Interface

Mit Hilfe des Makefile können die einzelnen Source-Module des Monitoring Interface kompiliert und zu lauffähigen Programmen zusammengebunden werden. Die Pade im Makefile können sich natürlich je nach System ändern und müssen daher bei Bedarf angepasst werden.

```
include $(APRICOTS_ROOT)/environment

MONITORDIR = $(APRICOTS_ROOT)/MonitoringInterface
JAVAOUTPUTDIR = $(APRICOTS_IDL_Java_stubs)
CLASSPATH=./opt/JavaWorks/JDK/lib/classes.zip:$(ORBIX_JAVA_PATH)/classes:
    $(APRICOTS_IDL_Java_stubs)/local/apricots/Java/classes:$(MONITORDIR)

JAVACOPT = -classpath $(CLASSPATH)
IDL2JAVASERVEROPT = -m interOp -I $(APRICOTS_IDL) -d$(REPOSITORY) -R -jO
    $(APRICOTS_IDL_Java_stubs)

all:    generate_java_stubs \
    $(JAVAOUTPUTDIR)/APRICOTS_Definitions/ConTractInstanceID.class \
    $(JAVAOUTPUTDIR)/APRICOTS_StorageAgent.class \
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorCallBack.class \
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorRegPoint.class \
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorInterface.class \
    $(MONITORDIR)/DrawObject.class \
    $(MONITORDIR)/Raster.class \
    $(MONITORDIR)/Element.class \
    $(MONITORDIR)/AboutBox.class \
    $(MONITORDIR)/CreateInstance.class \
    $(MONITORDIR)/PassBox.class \
    $(MONITORDIR)/QuitBox.class \
    $(MONITORDIR)/IO.class \
    $(MONITORDIR)/ShowAsyn.class \
    $(MONITORDIR)/ShowSyn.class \
    $(MONITORDIR)/Monitor.class

$(MONITORDIR)/DrawObject.class:$(MONITORDIR)/DrawObject.java
    echo Compiling DrawObject.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/DrawObject.java

$(MONITORDIR)/Raster.class:    $(MONITORDIR)/Raster.java
    echo Compiling Raster.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/Raster.java

$(MONITORDIR)/Element.class:    $(MONITORDIR)/Element.java
    echo Compiling Element.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/Element.java

$(MONITORDIR)/AboutBox.class:    $(MONITORDIR)/AboutBox.java
    echo Compiling AboutBox.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/AboutBox.java

$(MONITORDIR)/CreateInstance.class:    $(MONITORDIR)/CreateInstance.java
```

```
    echo Compiling CreateInstance.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/CreateInstance.java

$(MONITORDIR)/PassBox.class:  $(MONITORDIR)/PassBox.java
    echo Compiling PassBox.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/PassBox.java

$(MONITORDIR)/QuitBox.class:  $(MONITORDIR)/QuitBox.java
    echo Compiling QuitBox.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/QuitBox.java

$(MONITORDIR)/IO.class:  $(MONITORDIR)/IO.java
    echo Compiling IO.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/IO.java

$(MONITORDIR)/ShowAsyn.class:  $(MONITORDIR)/ShowAsyn.java
    echo Compiling ShowAsyn.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/ShowAsyn.java

$(MONITORDIR)/ShowSyn.class:  $(MONITORDIR)/ShowSyn.java
    echo Compiling ShowAsyn.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/ShowSyn.java

$(MONITORDIR)/Monitor.class:  $(MONITORDIR)/Monitor.java
    echo Compiling Monitor.java ...
    $(JAVAC) $(JAVACOPT) $(MONITORDIR)/Monitor.java

##### Create class-files #####

$(JAVAOUTPUTDIR)/APRICOTS_StorageAgent.class:
    $(JAVAOUTPUTDIR)/APRICOTS_StorageAgent.java
    $(JAVAC) $(JAVACOPT) $(JAVAOUTPUTDIR)/APRICOTS_StorageAgent.java

$(JAVAOUTPUTDIR)/APRICOTS_Definitions/ConTractInstanceID.class:
    $(JAVAOUTPUTDIR)/APRICOTS_Definitions/ConTractInstanceID.java
    $(JAVAC) $(JAVACOPT) $(JAVAOUTPUTDIR)/APRICOTS_Definitions/*.java

$(JAVAOUTPUTDIR)/APRICOTS_MonitorCallBack.class:
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorCallBack.java
    $(JAVAC) $(JAVACOPT) $(JAVAOUTPUTDIR)/APRICOTS_MonitorCallBack.java

$(JAVAOUTPUTDIR)/APRICOTS_MonitorRegPoint.class:
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorRegPoint.java
    $(JAVAC) $(JAVACOPT) $(JAVAOUTPUTDIR)/APRICOTS_MonitorRegPoint.java

$(JAVAOUTPUTDIR)/APRICOTS_MonitorInterface.class:
    $(JAVAOUTPUTDIR)/APRICOTS_MonitorInterface.java
    $(JAVAC) $(JAVACOPT) $(JAVAOUTPUTDIR)/APRICOTS_MonitorInterface.java

##### Java-files from IDL #####

generate_java_stubs:
    cd $(JAVAOUTPUTDIR); $(APRICOTS_MAKE)
```

F Die Arbeitsumgebung des Monitoring Interface

F.1 Starten des Programms

Das Monitoring Interface ist ein Java Programm und ist daher theoretisch plattform-unabhängig. Es kann sowohl als Stand-Alone-Anwendung als auch als Applet in einem WWW-Browser gestartet werden. Voraussetzung ist natürlich, daß von dem jeweiligen System Java-Programme unterstützt werden.

Start als Stand-Alone-Anwendung

```
java Monitor.java
```

Es ist zu beachten, daß vor dem Start die Pfade richtig gesetzt werden müssen. Insbesondere muß die Java-Umgebungsvariable `$CLASSPATH` richtig gesetzt werden, da sonst einige zur Ausführung benötigte Klassen nicht gefunden werden.

Start als Applet

```
appletviewer Monitor.html bzw.
```

Öffnen der Datei Monitor.html in einem Java-fähigen WWW-Browser.

Auch hier ist zu beachten, daß die Pfade und Umgebungsvariablen vorher richtig gesetzt werden.

F.2 Bekannte Fehler

Das Monitoring Interface ist nur ein Teil des APRICOTS-Systems und daher auf andere Komponenten angewiesen. Falls diese nicht zur Verfügung stehen oder nicht richtig funktionieren, kann natürlich auch das Monitoring Interface nicht oder nicht richtig arbeiten.

Insbesondere benötigt das Monitoring Interface folgende Komponenten:

- Einen Monitor Agent
- Einen Storage Agent
- Den Naming Service von CORBA

Außerdem ist für den Betrieb des Monitoring Interface ein Minimum an Hauptspeicher erforderlich, da gewisse Datenstrukturen im RAM des Rechners aufgebaut werden müssen. Sollte dieser Hauptspeicher knapp werden oder nicht mehr zur Verfügung stehen, beendet sich das Monitoring Interface mit einer Fehlermeldung.

G Die Klassen und Dateien des Monitoring Interface

An dieser Stelle werden die Klassen und Dateien des Monitoring Interface kurz beschrieben. Für jede Klasse existiert sowohl eine Quell- als auch Klassendatei, die im Verzeichnis Monitoring Interface abgespeichert sind.

G.1 Beschreibung der Klassen

<i>AboutBox:</i>	Diese Klasse öffnet ein Dialogfenster mit Informationen zum Programm.
<i>CreateInstance:</i>	Diese Klasse öffnet ein Eingabefenster, in das der Name einer neu anzulegenden Instanz eingegeben werden kann.
<i>DrawObject:</i>	Diese Klasse wird zum Aufbau der internen Speicherstruktur für ein eingelesenes Prädikat-Transitionsnetz benötigt.
<i>Element:</i>	Diese Klasse speichert Daten des Prädikat-Transitionsnetzes.
<i>Monitor:</i>	Diese Klasse wird beim Start des Monitoring Interface aufgerufen und stellt das Hauptfenster auf dem Bildschirm dar.
<i>IO:</i>	In dieser Klasse werden alle Input- / Output-Methoden gekapselt. Sie bietet Methoden zum Zugriff auf den Monitor- und Storage Agenten usw.
<i>PassBox:</i>	Die Klasse PassBox implementiert eine rudimentäre Passwort Abfrage zu Beginn des Programms. Die eingegebenen Daten werden jedoch noch nicht ausgewertet.
<i>Raster:</i>	Diese Klasse ist für das Zeichnen des ConTract-Skripts auf dem Computer-Bildschirm verantwortlich.
<i>ShowAsyn:</i>	Diese Klasse ist relativ komplex. Sie ist für das Einlesen des Prädikat-Transitionsnetzes und für die Berechnung des sich daraus ergebenden Graphen zuständig. Nach dem Einlesen und der Berechnung wird ein zweites Fenster geöffnet, in dem der ConTract dargestellt wird. Neben Möglichkeiten zur Manipulation kann der ConTract natürlich auch mit Hilfe von Rollbalken in x- und y-Richtung gescrollt werden. Der Bearbeitungszustand des ConTracts wird nur als Snapshot dargestellt.

- ShowSyn:** Diese Klasse hat die gleiche Aufgabe wie die Klasse ShowAsyn, mit dem Unterschied, daß der Bearbeitungsstand eines ConTracts fortlaufend überwacht und angezeigt wird.
- QuitBox:** Diese Klasse implementiert eine Sicherheitsabfrage vor dem Verlassen des Programms, ob dies auch wirklich gewünscht wird.

G.2 Beschreibung der restlichen Dateien

- start:** Dies ist eine kleine UNIX-Skrip-Datei zum einfacheren Starten des Monitoring Interfaces.
- basedef.idl:** Dieses IDL-File enthält allgemein gültige Definitionen.
- monitorInterface.idl:** Die Schnittstelle zum Monitor Agent.
- storageagent.idl:** Die Schnittstelle zum Storage Agent.
- Makefile:** Das Makefile für das Monitoring Interface
- Monitor.html:** Das HTML-File zum Starten des Monitoring Interface über einen Java-fähigen WWW-Browser.

H Literatur

- [Balz88] H.Balzert: *Software-Ergonomie und Software Engineering*. De Gruyter Verlag, 1988
- [Beims95] H.D.Beims: *Praktisches Software Engineering - Vorgehen - Methoden - Werkzeuge*. Carl Hanser Verlag München, 1995
- [BuRai90] H.Burkert, B.Raichle: *Ein Daten- und Verarbeitungsmodell für den Kontext*. Doppelstudienarbeit, Fakultät Informatik der Universität Stuttgart, Mai 1990
- [DaEde96] B.Davignon, G.Edelmann: *Java - Oder: Wie steuere ich meine Kaffeemaschine ?* tewi - Verlag, München, 1996
- [Eitl92] Markus Eitler: *Eine graphische Oberfläche für die interaktive Steuerung langlebiger, verteilter Abläufe*. Diplomarbeit Nr. 956, Fakultät Informatik der Universität Stuttgart, Juli 1992
- [Flan96] D.Flanagan: *Java in a Nutshell*. O'Reilly & Associates, Februar 1996
- [GrRe93] J.Gray, A.Reuter: *Transactional Processing Systems - Concepts and Techniques*. Morgan Kaufman Publishers, 1993
- [Har74] F.Harary: *Graphentheorie*. R.Oldenbourg Verlag, München, 1974
- [Jung87] D.Jungnickel: *Graphen, Netzwerke und Algorithmen*. B.I. Wissenschaftsverlag, 1987
- [LanSt85] O.Lange, G.Stegemann: *Datenstrukturen und Speichertechniken*. Vieweg Verlag, Braunschweig, 1985
- [IONA95] *The Orbix Architecture*. IONA Technologies Ltd., November 1995
- [IONA96] *Orbix - Naming Service*. IONA Technologies Ltd., March 1996
- [Kühn96] Ralf Kühnel: *Die Java-Fibel - Programmierung interaktiver Homepages für das World Wide Web*. Addison-Wesley-Verlag, Bonn, 1996
- [OMG93] *The Common Object Request Broker: Architecture and Spezifikation*. OMG Document Number 93.xx.yy; Revision 1.2; Draft; 29 December 1993
- [OMG95] *The Common Object Request Broker: Architecture and Spezifikation*. OMG Document Number 95.xx.yy; Revision 2.0; Juli 1995

-
- [Orbix95a] *orbix 2 - programming guide*. IONA Technologies Ltd., Release 2.0 November 1995
- [Orbix95b] *orbix 2 - reference guide*. IONA Technologies Ltd., Release 2.0 November 1995
- [Orbix96] *Orbix for Java - White Paper*. IONA Technologies Ltd., February 1996
- [Petri62] C.A.Petri: *Kommunikation mit Automaten*, Dissertation, Universität Bonn, 1962
- [ReSwe95] A.Reuter, F.Schwenkreis: *ConTracts - A Low-Level Mechanism for Building General-Purpose Workflow Management-Systems*, Bulletin of the Technical Committee on Data Engineering Vol. 18 No.1, IEEE Computer Society, 1995
- [ReSweWä92] A.Reuter, F.Schwenkreis, H.Wächter: *Zuverlässige Abwicklung großer verteilter Anwendungen mit ConTracts - Architektur eines Prototypen*. Informatik Aktuell, Springer Verlag, 1992
- [ReuWä90] A.Reuter, H.Wächter: *Grundkonzepte und Realisierung des ConTract-Modells*. Informatik Spektrum 5: 202 - 212, 1990
- [Sachs72] H.Sachs: *Einführung in die Theorie der endlichen Graphen, Teil II*. Mathematisch Naturwissenschaftliche Bibliothek, Leipzig, 1972
- [Schw93] F.Schwenkreis: *APRICOTS - A Prototype Implementation of a ConTract System - Management of the Control Flow and the Communication System*. 12th Symposium on Reliable Distributed Systems, Princeton, Oktober 1993
- [Schw94] F.Schwenkreis: *A Formal Approach to Synchronize Long-Lived Computations*. Proc. of the 5th Australian Conference on Information Systems, Melbourne, 1994
- [Schw95] F.Schwenkreis: *APRICOTS - a workflow programming environment*. Extended Abstract 6th Intern Workshop on High Performance Transaction Systems (HPTS), 1995
- [Sedge93] Robert Sedgewick: *Algorithmen in C*, Addison-Wesley-Verlag, 1993
- [Seif95] Jens Seifert: *Ein Programmierwerkzeug für APRICOTS*. Studienarbeit 1422, Fakultät Informatik der Universität Stuttgart, Mai 1995
- [WaBo89a] Klaus Wagner, Rainer Bodendiek: *Graphentheorie I - Anwendungen auf Topologie, Gruppentheorie und Verbandstheorie*, B.I. Wissenschaftsverlag, Mannheim 1989

-
- [WaBo89b] Klaus Wagner, Rainer Bodendiek: *Graphentheorie II - Weitere Methoden, Masse Graphen, Planarität und minimale Graphen*, B.I. Wissenschaftsverlag, Mannheim 1989
- [WaBo92] Klaus Wagner, Rainer Bodendiek: *Graphentheorie III - Zahlen, Gruppen, Einbettungen von Graphen und Geschichte der Graphentheorie*, B.I. Wissenschaftsverlag, Mannheim 1992
- [WaNä87] H.Walther, G.Nägler: *Graphen, Algorithmen, Programme*. Springer Verlag, Wien, 1987
- [WäRe92] A.Reuter, H.Wächter: *The ConTract Model*. A.K.Elmagarmid (ed.): *Advanced Transaction Models for New Applications*, Morgan Kaufman Publishers, 1992