Universität Stuttgart

Institut für Parallele und Verteilte Höchstleistungsrechner

Prüfer: Prof. Dr.-Ing. A. Reuter

Betreuer: Dipl.-Inform. F. Schwenkreis

Ein Programmierwerkzeug für APRICOTS

Jens Seifert

Studienarbeit Nr. 1422

CR-Klassifikation: D.1.7 D.3.2 D.3.3 H.2.3 H.2.8

Begonnen am: 7.11.1994

Beendet am: 6.5.1995

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

1.1 Einführung

Das Prinzip der Transaktionen hat sich im Bereich der Datenbank-Systeme bewährt, da es den Anwendungsprogrammierer von Aufgaben der Synchronisation und Kontrolle in parallelen Anwendungen entlastet. Jedoch gibt es viele Gebiete, in denen klassische Transaktionen nicht angebracht sind, um die reale Welt zu beschreiben. Das trifft vor allem auf langlebige (Minuten, Stunden, Tage, Wochen dauernd) und parallele Vorgänge in verteilten Systemen zu. Deshalb sind Erweiterungen am Transaktionsmodell erforderlich. Zwei Nachteile des klassischen Transaktionskonzeptes sind:

- Synchronisation: Die Atomaritätseigenschaft des klassischen Transaktionskonzeptes fordert, die Ergebnisse einer Transaktion erst sichtbar zu machen, nachdem diese erfolgreich abgeschlossen wurde. Um diese Forderung zu erfüllen, werden alle von einer Transaktion berührten Objekte für die gesamte Dauer der Transaktion gesperrt. Nebenläufige Transaktionen, die auf dieselben Objekte zugreifen, müssen solange warten, bis die sperrende Transaktion vollständig abgelaufen ist. Auf diese Weise wird die Serialisierung nebenläufiger, auf dieselben Objekte zugreifender Transaktionen erzwungen, wodurch der mögliche Parallelitätsgrad vieler Anwendungen enorm eingeschränkt wird.
- Fehlerbehandlung: Transaktionen werden vollständig zurückgesetzt, nachdem ein Systemfehler aufgetreten ist. Die Folge davon ist, daß alle bis zum Fehlerzeitpunkt berechneten Zwischenergebnisse verloren gehen. Die Transaktionen müssen neu gestartet werden, sobald es der Systemzustand erlaubt.

Diese Nachteile sind tragbar, wenn es sich um kurze Transaktionen im Sekundenbereich handelt. Bei langlebigen Transaktionen sind diese Eigenschaften nicht tragbar. Deshalb wurde das ConTract-Modell entwickelt. Die Idee beim ConTract-Modell ist es, die langlebigen Vorgänge in einzelne Teilschritte (Steps) zu unterteilen, die unter Transaktionsschutz nach dem ACID-Prinzip ablaufen. Dabei wird lediglich der Kontrollfluß (Skript) beschrieben, während die Kontroll- und Synchronisationsaufgaben dem System überlassen werden. Die wichtigsten Vorteile des ConTract-Modells sind:

- die Entkopplung und getrennte Beschreibung der Anwendungsprobleme und des Kontrollflusses bei der Implementierung langlebiger, paralleler Vorgänge
- die vorzeitige Freigabe der Sperren für schon bearbeitete Objekte, damit anderen parallel ablaufenden Operationen der Zugriff auf diese Objekte ermöglicht wird
- die Möglichkeit, die Abwicklung eines durch einen Fehler abgebrochenen Vorgangs fortzusetzen, ohne die bis zum Fehlerzeitpunkt berechneten Zwischenergebnisse zu verlieren (forward recovery)
- die Möglichkeit eine anwendungsorientierte Fehlerbehandlung anzugeben, falls ein Fehler bei der Bearbeitung eines Vorgangs aufgetreten ist

1.2 Aufgabenstellung

Im Projekt APRICOTS, dessen Aufgabe eine prototypische Implementierung eines ConTract verarbeitenden Systems ist, steht eine Sprache zur Verfügung, die es erlaubt den Kontrollfluß einer langandauernden Anwendung, unabhängig von den tatsächlich ausführenden Teilen (Steps) zu programmieren (Workflow). Da es wichtig ist solche Abläufe zu visualisieren, um während der Laufzeit das System überwachen und kontrollieren zu können, muß eine graphische Repräsentation der Abläufe an der Benutzerschnittstelle dargestellt werden. Um ein einheitliches Benutzungsmodell des Systems zu gewährleisten, bietet es sich somit an, bereits für den Definitionsprozeß eine graphische Programmierumgebung zur Verfügung zu stellen.

Die Aufgabe dieser Studienarbeit war es, ein Werkzeug zu realisieren, das die graphische Programmierung eines Kontrollflusses eines ConTracts ermöglicht. Die erzeugte Darstellung sollte auch als Grundlage für eine Komponente zur Visualisierung des Ablaufes dienen. Dabei mußte auf die Anbindung in das Gesamtsystem geachtet werden und berücksichtigt werden, daß mehrere Abläufe parallel verwaltet werden können.

1.3 Gliederung der Arbeit

In Kapitel 2 wird ein Überblick über das Projekt APRICOTS gegeben. Das dritte Kapitel zeigt den Entwicklungstand des Projekts in bezug auf diese Studienarbeit und die Ziele dieser Arbeit. Das vierte Kapitel beschreibt die Implementierungsansätze, die in dieser Arbeit Verwendung fanden. Das fünfte Kapitel beschreibt die Schnittstellen der Arbeit und somit auch die Integration der Arbeit in das Projekt. In Kapitel 6 folgt eine Zusammenfassung und ein Ausblick, was noch getan werden kann. Im Anhang befinden sich nützliche Hinweise für nachfolgende Arbeiten, die auf diese Studienarbeit aufbauen.

2 Überblick

2.1 Was ist ein ConTract?

ConTracts sind ein Modell für langlebige Aktionen auf Basis von ACID-Transaktionen. Für einen ConTract ist garantiert, daß die Ausführung in endlicher Zeit entweder erfolgreich abgeschlossen wird, der Originalzustand des ConTracts wiederhergestellt wird oder ein dazu logisch äquivalenter Zustand durch die Kompensierung der Aktionen rekonstruiert wird. Also ist garantiert, daß das ConTract Laufzeitsystem in fast allen Fällen eines Fehlers die Ausführung wiederaufnehmen und mit einem minimalen Verlust von Arbeit fortsetzen kann.

2.2 Wie wird ein ConTract definiert?

Die Definition eines ConTracts setzt sich aus zwei wesentlichen Teilen zusammen:

- Die Definition des Kontrollflusses und des Datenflusses: Dies wird durch die Programmierung eines Skriptes erreicht. Dazu ist es nötig die Steps anzuordnen, sowie den Kontrollfluß und die Abhängigkeiten untereinander, also den Datenfluß, festzulegen.
- Die Definition der Steps: Die Steps müssen kodiert werden. Ein Step besteht aus sequentiellem Code, der als ACID-Transaktion ausgeführt wird.

2.2.1 Das Skript

Ein Skript wird definiert, um den korrekten Ablauf eines Workflows zu garantieren. Für die Modellierung des Kontrollflusses stehen verschiedene Konstrukte zur Verfügung. Zum einen gibt es Konstrukte aus herkömmlichen Programmiersprachen wie bedingte Verzweigungen oder Schleifen. Zum andern existieren zusätzliche Konstrukte wie Parallelausführung oder Transaktion.

2.2.2 Die Steps

Ein Step ist ein elementarer Verarbeitungsschritt in einem ConTract. In der Codierung der Steps liegt die eigentliche Aufgabe der Anwendungsprogrammierung. Sie werden in einer beliebigen Programmiersprache programmiert. Sie bestehen aus rein sequentiellem Code, d.h. sie beschreiben nur sequentielle Vorgänge.

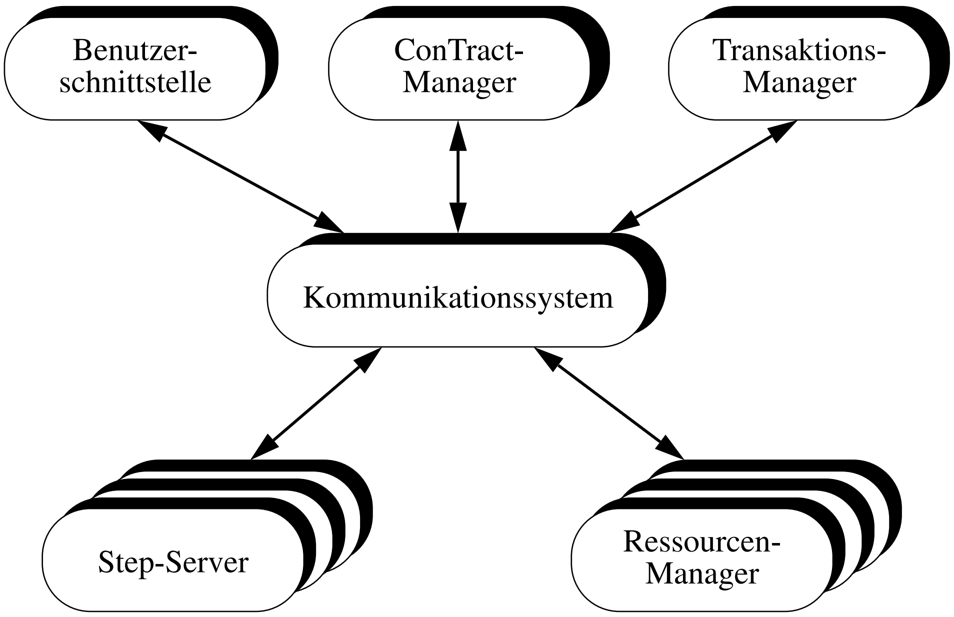
Jeder Step implementiert eine elementare Operation einer Anwendung wie z.B. eine Platzreservierung in einem Flugbuchungssystem oder eine Ab- oder Zubuchung in einem Überweisungsvorgang. Die Laufzeit eines Steps sollte möglichst gering sein. Damit wird erreicht, daß die zu bearbeitenden Daten nicht unnötig lange gesperrt bleiben und daß keine dazu parallel ablaufende Steps, die auf die gleichen Daten zugreifen, behindert werden. Außerdem sollten die von einem Step erfaßten Daten möglichst gering sein, damit im Fehlerfall nicht zu viele Daten verloren gehen.

Die Codierung der Steps ist unabhängig von der Spezifikation des Kontrollflusses der Anwendung. Die Programmierung der Steps und die Beschreibung des Ablaufes sind zwei getrennte Aufgaben. Für den Programmierer des Kontrollflusses ist nur die Schnittstelle eines Steps ausschlaggebend, die aus dem Namen, der Ressourcenmanagerklasse und Parametern besteht (siehe [Häu93]).

2.3 APRICOTS

APRICOTS steht für „**A PR**ototype **I**mplementation Of a **CO**n**T**ract **S**ystem“. Die Architektur von APRICOTS ist strikt modular. Die funktional zusammenhängenden Teile sind in Komponenten zusammengefaßt. Die Verknüpfung und Kommunikation findet über das Kommunikationssystem statt. Einen Überblick gibt folgende Abbildung:

ABBILDUNG 1. Die Architektur von APRICOTS



Die folgenden Abschnitte geben einen kurzen Einblick über die Aufgaben der einzelnen Komponenten.

2.3.1 Der ConTract-Manager (CM)

Ein ConTract-Manager hat die Aufgabe ein Skript auszuführen, das den Ablauf in einem ConTract beschreibt. Da sich die Skriptausführung über einen langen Zeitraum erstrecken kann, liegt der Schwerpunkt in der Zuverlässigkeit der Ausführung.

2.3.2 Der Transaktions-Manager (TM)

Diese Komponente ist für das Protokoll zuständig, um die transaktionale Semantik (z.B. 2-Phasen-Commit) zu gewährleisten.

2.3.3 Das Kommunikationssystem (CS)

Das Kommunikationssystem ist die zentrale Nachrichtenübermittlung-Schnittstelle. Damit können im gesamten System Remote Procedure Calls (RPCs) versendet werden.

2.3.4 Die Step-Server (SS)

Die Step-Server sind benutzerprogrammierte Komponenten. Ein Step-Server beinhaltet Code für Steps. Der Code ist beliebiger benutzerprogrammierter sequentieller Code. Die Ausführung des Step-Codes erfolgt asynchron zur Skript-Ausführung des ConTract-Managers.

2.3.5 Die Ressourcen-Manager (RM)

Ein Ressourcen-Manager verwaltet Objekte. Er bietet Funktionen zum Zugriff auf Objekte wie z.B. eine Datenbank an. Im Gegensatz zu den Step-Servern wird ein Ressourcen-Manager synchron aufgerufen. Der Aufruf wird durch einen Step-Server initiiert und nicht durch den ConTract-Manager.

2.3.6 Die Benutzerschnittstelle

Die Benutzerschnittstelle erlaubt es dem Benutzer, einen ConTract zu starten oder durch Kommandos zu manipulieren. Die Schnittstelle visualisiert das Skript in einer graphischen Form, so daß der aktuelle Zustand der Bearbeitung festgestellt werden kann.

3 Entwicklungsstand und Ziele

3.1 Skripteingabe

Die Skripteingabe für den Ablauf eines ConTract erfolgt bisher durch eine Programmiersprache, die durch einen Compiler in ein ablauffähiges Prädikat-Transitionsnetz überführt wird. Die Sprachdefinition war Inhalt einer Studienarbeit [Masm90]. Das Programm wird dabei mit einem Texteditor eingegeben und durch einen Kommandozeilen-gesteuerten Compiler in eine interne Darstellung überführt. Diese Darstellung kann vom ConTract-Manager interpretiert werden.

3.2 Die Visualisierung des Ablaufs

Der Ablauf eines ConTract wurde im APRICOTS-Projekt bisher durch ein Programm verrichtet, welches die vom Compiler erzeugte interne Darstellung einliest und graphisch aufbereitet. Die graphische Aufbereitung ist dabei sehr aufwendig. Dieses Programm wurde im Rahmen einer Diplomarbeit [Eitl92] erstellt.

3.3 Motivation

Zur Erläuterung der Motivation für die graphische Eingabe des Skriptes eines ConTracts werden im folgenden die Vor- und Nachteile diskutiert.

3.3.1 Vorteile einer graphischen Eingabe

- Die syntaktische Korrektheit der Eingabe kann gewährleistet werden. Dadurch ist vom Compiler keine Syntaxprüfung mehr vorzunehmen.
- Die semantische Korrektheit kann größtenteils vor dem Compilieren oder Erzeugen einer Instanz geschehen, indem Prüfungen direkt nach der Eingabe auf der Oberfläche geprüft werden.
- Dem Programmieren wird Gestaltungsspielraum genommen, zum Beispiel gibt es keine Möglichkeit überlange Zeilen zu machen. Das heißt, die Lesbarkeit für andere Programmierer wird durch eine graphische Darstellung erhöht. Dies ist ein Beitrag zum sogenannten „ego-less-programming“.
- Die graphische Eingabe des Skriptes hat den Vorteil, daß die Darstellung auch für die Visualisierung des Ablaufs verwendet werden kann.
- Die Akzeptanz des Benutzers ist höher, da er keine Syntax zu lernen braucht und die graphische Oberfläche eine strukturierte Darstellung vorschreibt, ohne daß er es merkt.
- Es können Werte für Eingabefelder vorgegeben sein, sogenannte Defaults. Damit kann schnell eine Version erstellt werden, ohne alles selbst eingeben zu müssen.

3.3.2 Nachteile einer graphischen Eingabe

- Die Erstellung einer Oberfläche, die Benutzerfehler abfängt, die Fehleingaben vermeiden hilft und dennoch benutzerfreundlich ist, ist relativ gesehen zum Aufwand eines Compilers für eine Texteingabe, der die Syntax und Semantik prüft, aufwendig. Das liegt vor allem an der Verfügbarkeit von Werkzeugen wie lex und yacc für die Verarbeitung von Texteingaben.
- Die Oberfläche muß für ein graphikfähige Umgebung geschrieben werden. Dies schränkt die Portabilität gegenüber einer reinen Texteingabe ein.

3.4 Ziele

Ein Ziel des Projekts APRICOTS ist es, dem Benutzer eine einheitliche Schnittstelle zur Verfügung zu stellen. Dazu gehört auch ein Werkzeug, das es erlaubt, einen ConTract einzugeben und auszuführen. Dieses Werkzeug wird im weiteren als **ConTract-Development-Tool** bezeichnet.

3.4.1 Das ConTract-Development-Tool (CDT)

Das ConTract-Development-Tool soll folgendes leisten:

- die Auswahl des zu bearbeitenden ConTracts
- das Erstellen und Editieren eines ConTracts
- das Erzeugen einer ausführbaren Instanz eines ConTracts
- die Visualisierung des Ablauf eines ConTracts

Die Aufgabe der Studienarbeit war es, ein Programmierwerkzeug zur graphische Erstellung des Skriptes eines ConTracts zu entwickeln. Diese Komponente wird mit **ConTract-Editor** bezeichnet. Die Möglichkeiten einer Visualisierung des Ablaufs sollten erarbeitet werden. Die Komponente für die Visualisierung wird mit **ConTract-Animator** bezeichnet. Die Erzeugung einer ausführbaren Instanz mit Hilfe eines Compilers ist nicht Teil dieser Studienarbeit, mußte aber berücksichtigt werden.

3.4.2 Der ConTract-Editor (CE)

Mit dem ConTract-Editor soll ein Programmierwerkzeug für die Erstellung des Skriptes eines ConTract erstellt werden. Damit wird ein Ersatz geschaffen für die bisherige textuelle Eingabe des Skriptes. Dabei ist darauf zu achten, daß der Mechanismus zur Eingabe des Skriptes so gestaltet ist, daß Fehler in der Syntax ausgeschlossen und Fehler in der Semantik vermieden werden. Das erzeugte Skript muß dem Compiler in einer Form zur Verfügung gestellt werden, aus der dieser eine ausführbare Instanz erstellen kann. Die ausführbare Instanz besteht in diesem Projekt aus einem Prädikat-Transitionsnetz, das vom einem ConTract-Manager interpretiert wird. Der wichtigste Vorteil einer graphischen Eingabe ist es aber, daß die Visualisierung des Ablaufs auf der gleichen Darstellung wie im Editor beruhen kann.

3.4.3 Der ConTract-Animator (CA)

Die Visualisierung des Ablaufes durch die Darstellung des Prädikat-Transistionsnetz wie es bisher der Fall war, hat den Nachteil, daß die Bindung zur Eingabe des Skriptes naturgemäß fehlt, da es sich ja um eine textuelle Repräsentation handelte. Dieser Nachteil soll durch die Integration des Editors und der Visualisierung des Ablaufs aufgehoben werden. Außerdem ist ein Benutzer des Systems wenig daran interessiert, welche Transitionen gerade „feuern“, sondern nur wann welche Steps warum ausgeführt werden. Deshalb soll durch diese Studienarbeit die Visualisierung des Ablaufes eines ConTracts mit der vom ConTract-Editor erstellten Skriptes erarbeitet werden.

4 Implementierungsansätze

4.1 Sprachumfang

Wenn ein Werkzeug zur Programmierung eines Skriptes erstellt werden soll, muß man sich Gedanken um den Sprachumfang machen. Die Sprache muß zur Definition des Kontrollflusses eines ConTract geeignet sein. Der Sprachumfang war in diesem Fall schon durch die vorangegangene Studienarbeit [Masm90] grob umrissen. Zum Vergleich der Sprachkonstrukte der alten und der neuen Variante sind im folgenden die Sprachkonstrukte aufgeführt. Um die beiden Varianten besser vergleichen zu können, sind die durch die Oberfläche eingebbaren Sprachkonstrukte in erweiterter Backus-Naur-Form (EBNF) angegeben. Damit kann zumindest die Syntax der beiden Sprachvarianten verglichen werden.

4.1.1 Vergleich der in beiden enthaltenen Konstrukte

4.1.1.1 ConTract

Der ConTract umfaßt bei beiden Varianten das Skript sowie die für den Datenfluß notwendigen Kontextvariablen. In der alten Variante sieht ein ConTract in EBNF wie folgt aus:

```
ConTract = 'ConTract' CONTRACT_ID
             'begin_context_decl'
                [ContextVariableDeclaration]
             'end_context_decl'
             'begin_script'
                Sequence
             'end_script'.
             [ 'begin_conflict_resolution'
                  [ConflictResolution]
               'end_conflict_resolution' ].
ContextVariableDeclaration =
   { ContextVariableName {',' ContextVariableName}
       ':' Type ';' }.
```

Die neue Variante deckt sich quasi komplett mit der alten Variante, nur daß die Konfliktauflösungen nun Bestandteil des Skriptes sind. In EBNF sieht ein ConTract nun wie folgt aus:

```
ConTract = 'CONTRACT' Name ',' Version
                [ContextVariableDeclaration]
             'BEGIN'
               Sequence
             'END' '.'.
ContextVariableDeclaration =
             'VAR' { ContextVariableName ':' Type ';' }.
```

Ein weiterer Unterschied besteht darin, daß für den ConTract keine Nummer mehr vergeben wird, sondern eine Identifizierung des ConTracts durch Name und Version angewandt wird.

4.1.1.2 Sequenz

Die Sequenz ist die sequentielle Abarbeitung von Steps und anderen Konstrukten. Die alte Variante würde in EBNF wie folgt aussehen:

```
Sequence  = {Statement ';'}.
Statement =  Step
           | IfBranch | SwitchBranch | ParallelBranch
           | ParforeachBranch
           | ExplicteSequence
           | Whileloop | ForLoop.
```

Die Sequenz der neuen Variante sieht ähnlich aus, sie unterscheidet nur in den vorhandenen Anweisungen.

```
Sequence  = HorLine { [Label] Statement HorLine }.
Statement =  StepButton
           | IfBranch | SwitchBranch | ParallelBranch
           | WhileLoop | RepeatLoop | ForLoop
           | Block | Transaction.
```

Label bezeichnet dabei ein mögliches Sprungziel nach einer Konfliktbehandlungen. HorLine steht für horizontale Linie und ist nicht anders als ein Trennsymbol für die Anweisungen.

4.1.1.3 Step

Bei diesem Konstrukt werden die Unterschiede etwas größer. Zuerst einmal die alte Variante in EBNF:

```
Step = 'STEP_NAME' '('ContextReferences')'
            [ EntryInvariant]
            [ ExitInvariant ].
ContextReference =
   { {'out'|'in'|'in_out'} ':'
     StepVarName ':' Name [ 'by' StepId ] ':' Type ';' }.
```

Die neue Version in EBNF:

```
StepButton =
   'STEP' Name',' RmClass '(' ListOfContextVariables ')'
           'ENTRY' 'INVARIANT' EntryInvariant
           'EXIT'  'INVARIANT' ExitInvariant
           'ON' 'ENTRY' 'INVARIANT' 'FAILURE'
              ConflictResolution 'GOTO' Label
           'ON' 'ABORT'
              ConflictResolution 'GOTO' Label
           ['COMPENSATION' 'STEP' Name',' RmClass
                        '(' ListOfContextVariables ')']
   'END'.
ListOfContextVariables = [{ Name ',' } Name ].
```

Zunächst einmal etwas allgemeines über die Steps. Wie man sieht ist ein Step wie ein Prozeduraufruf mit Parameterschnittstelle in einer herkömmlichen Programmiersprache gehalten. Durch die Invarianten wird im ConTract-Modell ein Mechanismus implementiert, um eine flexiblere Synchronisation als mit herkömmlichen Sperrverfahren zu erreichen. Jeder

Step besitzt eine Eintrittsinvariante, die vor der eigentlichen Ausführung geprüft wird. Damit wird die Ausführbarkeit des Steps vorab geprüft. Mit der Austrittsinvariante wird ausgedrückt, was für Sperren durch die Ausführung des Steps notwendig geworden sind.

In der neuen Variante umfaßt das Step-Konstrukt nicht nur den Aufruf. In ihr wird außerdem noch der Step angegeben, der im Falle einer Kompensation ausgeführt werden soll, und die Behandlung eines Eintrittsinvariantenfehlers oder eines Ausführungsfehlers. Das Konstrukt wird mit StepButton bezeichnet, weil dies ein Objekt in der graphischen Darstellung des Skriptes bezeichnet. Die Fehlerbehandlung in der alten Variante ist komplett vom normalen Skript getrennt.

In der alten Variante ist die Parameterschnittstelle etwas komplizierter aufgebaut. Zu jedem Parameter muß der Name des Parameters in Kombination mit der zu übergebenden Kontextvariablen angegeben werden. Außerdem muß noch ein Übergabemodus angegeben werden (in, out, in_out). In der neuen Variante ist dies nicht mehr so. Das hat folgende Gründe:

- Der Name des Parameters muß nicht mehr angegeben werden, da die Zuordnung der Kontextvariablen zum Parameter über seine Position gesteuert wird.
- Es muß kein Übergabemodus mehr angegeben werden. Wie die Parameter an den Step übergeben werden, wird nun sinnvollerweise durch die Step-Schnittstelle gesteuert, das heißt, daß jeder Step weiß, in welchem Modus er seine Parameter erwartet.

Die Angabe der Step-Identifikation(StepId) in der alten Variante hat den Sinn festzustellen, welcher Step die Kontextvariable mit einem Wert belegt. Das ist aber eigentlich Teil einer Datenfluß-Prüfung und ist deshalb in der neuen Variante ausgelassen.

In der neuen Variante ist die Ausführung eines kompletten Skriptes nach dem Auftreten eines Fehlers und das fast beliebige Verzweigen nach der Abarbeitung der Fehlerbehandlung möglich. Nach einem Fehler (Eintrittsinvariantenfehler oder Ausführungsfehler des Steps) darf nicht beliebig verzweigt werden. Die Einschränkungen für das Verzweigen werden bei der Transaktion näher erklärt.

Die Adressierung eines Steps besteht in der neuen Variante nicht nur aus dem Namen, sondern auch aus der Angabe der Ressourcenmangerklasse (siehe [Häu93]).

4.1.1.4 Einfache bedingte Verzweigung

Die einfache Verzweigung entspricht den if-Verzweigungen herkömmlicher Programmiersprachen. Durch ein spezifizierbares Prädikat wird festgelegt, welcher Pfad genommen wird. Die alte Variante in EBNF:

```
IfBranch = `if` `(` Predicate `)`
              Sequence
           [`else` Sequence ]
           `endif`.
```

Neue Variante in EBNF:

```
IfBranch = `IF` Predicate `THEN`
              Sequence
           `ELSE`
              Sequence
           `END`.
```

Die beiden Varianten sind quasi identisch, nur daß in der neuen Variante der ELSE-Zweig nicht optional, aber die Sequenz dennoch leer sein kann.

4.1.1.5 Mehrfache Verzweigung

Hier unterscheidet sich die Semantik der Varianten gewaltig. Bei der alten Variante gilt das Prinzip: genau einen Zweig aus vielen Ausführen. Die Syntax in EBNF:

```
SwitchBranch = `case` `(` ContextVariableName `)`
                  { Value `:` Sequence }
                  [ `otherwise` `:` Sequence ]
               `endcase`.
```

In der neuen Variante besitzt jeder Zweig ein vollständiges Prädikat. Dieses Prädikat steuert, ob der Zweig ausgeführt wird oder nicht. Treffen mehrere Prädikate zu, werden diese Zweige parallel ausgeführt. Es gibt auch einen Default-Zweig, der grundsätzlich ausgeführt wird. In EBNF:

```
SwitchBranch = `PARBEGIN`
                  Alternative { `;` Alternative } `;`
                  Sequence
               `PAREND`.
Alternative  = `IF` Predicate `THEN` Sequence `END`.
```

4.1.1.6 Parallele Verzweigung

Bei der parallelen Verzweigung werden alle Zweige parallel ausgeführt. Hierin unterscheiden sich die beiden Varianten überhaupt nicht. Deshalb hier nur einmal die Syntax:

```
ParallelBranch = `parbegin`
                    { Sequence }
                 `parend`.
```

4.1.1.7 While-Schleife

Entspricht den While-Schleifen in herkömmlichen Programmiersprachen. Die beiden Varianten unterscheiden sich nicht in der Semantik. Die Syntax ist sehr ähnlich. Hier jetzt die alte Variante:

```
WhileLoop = `while` `(` Predicate `)`
               Sequence
            `endwhile`.
```

4.1.1.8 For-Schleife

In der alten Variante sieht die For-Schleife wie folgt aus:

```
ForLoop = `for` `(` Expression `to` Expression `)`
             Sequence
          `endfor`.
```

D.h. ein Schleifenzähler zählt vom ersten Ausdruck bis zum zweiten Ausdruck hoch, indem der Schleifenzähler jedesmal um 1 inkrementiert wird. In der neuen Variante entspricht die For-Schleifen der For-Schleife von Modula-2. In EBNF ausgedrückt:

```
ForLoop = `FOR` Variable `:=` Expression
                `TO` Expression
                `BY` ConstExpression `DO`
             Sequence
          `END`.
```

4.1.1.9 Transaktion

Die Transaktion hat die Aufgabe, erst bei korrekter Bearbeitung der Steps, die zu einer Transaktion gehören, ein Commit auf die Steps auszuführen. Das Konzept der Transaktion wird erst in [Masm91] beschrieben. Dort wird eine Transaktion durch eine Liste von Step-Identifikationen definiert. In der jetzigen Variante hat sich dieses Bild etwas gewandelt. Eine Transaktion besteht jetzt aus einem beliebigen Teil des Skriptes für den gilt, daß die Konstrukte so verbunden sind, daß der Kontrollfluß nur einen Eingang und einen Ausgang besitzt. Alle Steps innerhalb dieser Konstrukte gehören zur Transaktion. Falls die Bearbeitung eines Steps fehlschlägt, werden alle Aktionen, die von Steps innerhalb der Transaktion ausgeführt wurden, kompensiert. Hinzu kommt noch, daß beim Fehlschlagen einer Transaktion eine Fehlerbehandlung durchgeführt werden kann (sowie beim Step), um dann an einem fast beliebigen Ort im Skript fortsetzen zu können. Einschränkungen für das Verzweigen nach einer fehlgeschlagenen Transaktion sind:

- Es darf nicht in eine Transaktion gesprungen werden.
- Es darf nicht in einen Zweig eines Parallelkonstrukts gesprungen werden, mit Ausnahme des Falles, daß die Transaktion Teil dieses Parallelzweiges ist.

Diese Einschränkungen sind notwendig, damit die Ausführungseinheit (ConTract-Manager) bei der Interpretierung des Prädikat-Transitionsnetzes nicht durcheinandergerät. Es könnte zum Blockieren der Ausführung des Netzes kommen.

In EBNF könnte man eine Transaktion wie folgt ausdrücken:

```
Transaction =
    `TRANSACTION` Name
       `ON` `ABORT` ConflictResolution `GOTO` Label
        Sequence
    `END`.
```

4.1.1.10 Konfliktbehebung

Die Konfliktbehebung in der alten Variante bezieht sich nur auf Eintrittsinvarianten-Prüfungen. D.h. für jede Eintrittsinvariante kann im Falle eines Fehlschlagens eine Konfliktbehebung in Form eines Skriptes angegeben werden. Nach der Abarbeitung wird versucht, erneut die Eintrittsinvariante auszuwerten.

In der neuen Variante kann die Konfliktbehebung

- durch das Fehlschlagen einer Eintrittsinvariante eines Steps,
- durch das Fehlschlagen einer Stepausführung oder
- durch einen Abbruch einer Transaktion

ausgelöst werden.

Die neue Variante in EBNF ausgedrückt:

```
ConflictResolution = `CONFLICTRESOLUTION`
                         [ Sequence ]
                     `END`.
```

4.1.2 Weggefallene Konstrukte

4.1.2.1 Parforeach-Anweisung

Sie ist speziell auf Datenbank-Anweisungen abgestimmt. Die Erweiterung der neuen Sprachvariante durch die Parforeach-Anweisung ist vorgesehen.

4.1.2.2 Explizite Sequenz

Die explizite Sequenz wurde durch den Block ersetzt.

4.1.3 Hinzugekommene Konstrukte

4.1.3.1 Repeat-Schleife

Entspricht den Repeat-Schleifen in herkömmlichen Programmiersprachen. In EBNF sieht sie wie folgt aus:

```
RepeatLoop= 'REPEAT'
              Sequence
            'UNTIL' Predicate.
```

4.1.3.2 Block

Dient zur Zusammenfassung von Anweisungen, die nur einen Eingang und einen Ausgang im Kontrollfluß besitzen. Außerdem kann ein Name vergeben werden. IN EBNF:

```
Block = 'BLOCK' Name
          Sequence
        'END'.
```

4.1.4 Prädikate und Invarianten

In der Studienarbeit [Masm90] wird sowohl für Prädikate als auch für Invarianten eine Syntax angegeben. Da die Konzepte zur Behandlung und Syntax der Prädikate und Invarianten parallel zu dieser Studienarbeit überarbeitet wurden, konnten diese nicht integriert werden.

4.2 Objektorientierter Entwurf

Dieser Abschnitt gibt Einblicke, warum hier ein objektorientierter Entwurf sinnvoll ist. In der Entscheidung, warum ein objektorientierter Entwurf benutzt wurde, spielten folgende Aspekte eine Rolle:

- Eine Programmiersprache umfaßt eine endliche Menge von vordefinierten gleichartigen Konstrukten, z.B. Schleifen. Ein Programm ist eine Folge von solchen Konstrukten. Das bedeutet, daß **viele Instanzen** von einem Konstrukt gebraucht werden. Dies führt zur Benutzung des Konzepts des abstrakten Datentyps (**ADT**). Objektorientierte Sprachen unterstützen dieses Konzept durch Klassen. Das heißt, jedem Konstrukt einer Programmiersprache wird eine Klasse zugeordnet. Dann können beliebig viele Instanzen dieser Klassen gebildet werde. Außerdem wird durch die Klassen das **Information-Hiding**, das im Konzept des ADTs beinhaltet ist, unterstützt, indem der Zugriff auf die Datenelemente einer Klasse nur mit den dafür vorgesehenen Methoden möglich ist.
- Da jedem Konstrukt der Skript-Programmiersprache ein graphischer Repräsentant (Objekt) zugeordnet ist, taucht die Frage auf, in wie weit sich diese Repräsentanten

ähneln. Dabei stellt man fest, daß zumindest einige Attribute der Repräsentanten, z.B. Position, Breite, Höhe, Hintergrundfarbe, Vordergrundfarbe für jeden Repräsentant vorhanden sein müssen. Dies sind die optimalen Voraussetzungen, um die **Vererbungsmechanismen** einer objektorientierten Sprache zu benutzen.

- Da von den Klassen auch weitere Klassen abgeleitet werden können, garantiert ein objektorientierter Entwurf die Wiederverwendbarkeit der Klassen für andere Anwendungszwecke.

4.3 Warum C++?

Im Grunde genommen wäre jede objektorientierte Programmiersprache geeignet gewesen, um diesen Entwurf zu codieren. Gründe für C++ als Implementierungssprache:

- Das APRICOTS-System ist bisher ausschließlich in C geschrieben. Da C++ eine Obermenge von C darstellt, ist eine Integration von C-Code relativ problemlos möglich.
- Für die Anbindung an die Datenbank stand keine objektorientierte Schnittstelle zur Verfügung. Deshalb mußte Embedded SQL in C benutzt werden.
- Die Mächtigkeit der Sprache C++ (z.B. Mehrfachvererbung, parametrisierte Klassen).
- Die Geschwindigkeit und Ressourcenbedarf des erzeugten Compilats. Sie ist besonders wichtig, um den Benutzer vor langen Wartezeiten beim interaktiven Betrieb zu bewahren. C++ hat in dieser Disziplin einen klaren Vorsprung vor Eiffel oder Smalltalk.
- Die Verfügbarkeit von Compilern für C++ ist sehr gut. Für andere Sprachen trifft dies im allgemeinen nicht zu.
- Die Anbindung an die X-Windows-Umgebung. Für X-Windows sind nur C-Schnittstellen definiert. Außerdem mußte auf die Einbindung des verwendeten Interface Composer von IBM zur Erstellung der graphischen Benutzeroberfläche geachtet werden.

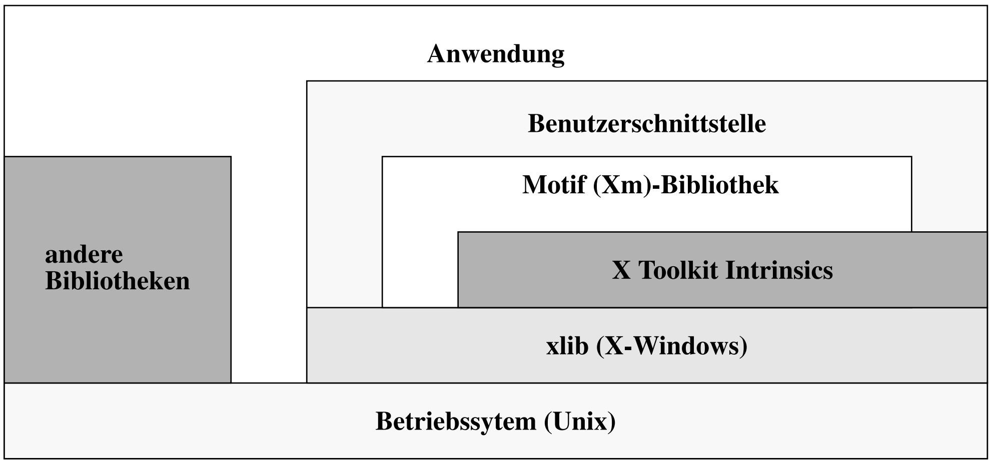
4.4 Die graphische Benutzeroberfläche

4.4.1 Warum gerade Motif?

Wenn eine graphische Benutzeroberfläche erstellt werden soll, sollte darauf geachtet werden, daß die Plattform auf der diese Oberfläche läuft stark verbreitet ist, um das Portieren zu erleichtern. Dies ist besonders wichtig, wenn die Programme in einer verteilten Umgebung auf verschiedenen Rechnern und Betriebssystemen laufen sollen, was bei diesem Projekt der Fall ist. Im Bereich der Unix-Workstations gibt es dafür X-Windows. X-Windows ist zur Zeit die wohl am verbreitetsten und am besten verfügbare Basis für die Programmierung einer graphischen Benutzeroberfläche(GUI). Da die Erstellung einer graphischen Benutzeroberfläche nur mit X-Windows sehr aufwendig und fehleranfällig ist, gibt es verschiedene Aufsätze zur Erweiterung der Funktionalität von X-Windows. Zum einen stellt X-Windows die sogenannten X Toolkit Intrinsics zur Verfügung, welche eine Bibliothek von GUI-Objekten zur Verfügung stellt. Diese Bibliothek ist primitiv und nicht sehr umfangreich. Deshalb wurde von der Open Software Foundation der Motif-Toolkit entwikkelt. Dieser Toolkit stellt einen kompletten Satz von graphischen Elementen zur Programmierung einer GUI zur Verfügung, deren „look and feel“ im Motif-Style-Guide und der

Motif-Application-Specification beschrieben ist. Dies garantiert dem Benutzer, daß sich alle Programme, deren graphische Benutzeroberfläche nach den Motif-Konventionen entworfen und implementiert wurden, in etwa gleich verhalten und in gleicher Art und Weise bedient werden können. Das Modell wie die Bibliotheken aufgebaut sind, zeigt folgende Abbildung:

ABBILDUNG 2. Die GUI-Bibliotheken



Da Motif auf sehr vielen Unix-Plattformen verfügbar ist, wurde Motif in dieser Arbeit verwendet. Außerdem gibt es für Motif zahlreiche Programme zur graphischen Eingabe der Benutzeroberfläche (Interface Composer).

4.4.2 Warum ein Interface Composer?

Die Programmierung einer Benutzeroberfläche von Hand ist aufwendig, weil sich der Programmierer mühsam um die Positionierung und Gestaltung der Oberflächen-Elemente kümmern muß. Um den Aufwand zur Programmierung der graphischen Benutzeroberfläche zu verringern, liegt es nahe, diese auch graphisch einzugeben. Da zu Beginn der Studienarbeit nur ein einziges brauchbares Werkzeug zur Erzeugung einer Motif-konformen Benutzeroberfläche, nämlich der AIXwindows Interface Composer Version 1.2 von IBM, zur Verfügung stand, wurde dieser verwendet. Das Programm war auf einem RS/6000-Rechner von IBM unter AIX installiert. Die graphische Programmierung der Benutzeroberfläche bringt auch Nachteile mit sich. Zum einen ist man an das verwendete Werkzeug gebunden, da sich jedes Werkzeug eine eigene interne Repräsentation der Benutzeroberfläche erstellt, die untereinander nicht kompatibel sind. Der verwendete Interface Composer war zudem nur auf dem oben genannten Rechner lauffähig. Des weiteren haben solche doch relativ komplexen Programme den Nachteil, daß sie auch Fehler haben. Im folgenden sind einige Probleme aufgeführt, die beim Umgang mit diesem Interface Composers aufgetreten sind:

- Ein nicht reproduzierbarer Fehler führte immer wieder zu Verlusten von Änderungen.
- Die C++-Ausgabe des Interface Composers war für mich unbrauchbar, da der Zugriff auf die Elemente eines Interfaces mit der Ableitungsregel protected versehen war. Eine Klasse davon ableiten war nicht möglich, da der Interface Composer einen Destroy-Callback[1] einhängt, der die von ihm erzeugte Klasse bei der Speicherfreigabe des Interface-Widgets den Destruktor dieser Klasse aufruft. Dabei wird natürlich nicht berück-

1. Ein Callback ist eine Funktion, die bei Eintritt eines X-Events ausgeführt wird.

sichtigt, daß möglicherweise eine Klasse abgeleitet wurde. Deshalb wurde die C-Ausgabe des Interface Composers gewählt.

- Um auf Elemente des generierten Interfaces zugreifen zu können, gibt es vordefinierte Makros. Die gehen allerdings davon aus, daß jedes Interface nur einmal instanziiert wird und greifen auf eine globale Variable zu. Um dies zu umgehen, wurden in den View-Klassen diese globale Variable überladen.
- Eine vom Interface Composer erzeugte Datei mußte durch einen fehlenden Funktionsprototyp ergänzt werden, da sonst der C-Compiler von Sun jedesmal mit einer Fehlermeldung die Übersetzung abgebrochen hat.
- Der Interface Composer hat einen integrierten C-Interpreter, der nicht abschaltbar war. Das führte dazu, daß innerhalb des Interface Composers kein C++-Code eingegeben werden konnte. Deshalb sind die Callbacks für die Interfaces nun Bestandteil der View-Klassen.

Trotz der aufgeführten Probleme und der langen Einarbeitungszeit, hat sich gezeigt, daß sich der Einsatz des Interface Composers zur Erstellung der GUI lohnt. Das liegt vor allem daran, daß viele unnötige Compilerläufe vermieden werden können, da die Änderungen an der GUI sofort sichtbar werden. Eine Zeitersparnis trat somit auch ein, da viele Compilerläufe, die zwischen 2 und 15 Minuten dauerten, vermieden werden konnten.

4.4.3 Die Darstellung des Skriptes

Die Darstellung des Skriptes sollte einige Bedingungen erfüllen:

- Um die Lesbarkeit des Skriptes für andere Programmierer und den späteren Benutzer der Ausführungskomponente zu gewährleisten, sollte das Skript eine **einheitliche Darstellung** besitzen. D.h. die Darstellung sollte gleich sein, unabhängig davon, wer das Skript programmiert hat.
- Der Kontrollfluß soll möglichst **kreuzungsfrei** dargestellt werden. Damit wird die Übersichtlichkeit erhöht.
- Die Sprachkonstrukte sollten sich in der Darstellung **nicht überlappen**, damit die Darstellung übersichtlich bleibt.
- Die Darstellung des Skriptes sollte jederzeit **syntaktisch korrekt** sein.

Würde man dem Benutzer erlauben, die Repräsentanten für die Sprachkonstrukte frei auf dem Bildschirm zu plazieren, müßte man darauf hoffen, daß der Benutzer bei der Erstellung des Kontrollflusses zwischen den Sprachkonstrukten keine Fehler macht und auch keine Verbindungen vergißt. Außerdem wäre nicht gewährleistet, daß der Kontrollfluß kreuzungsfrei ist und daß sich die Sprachkonstrukte in der Darstellung nicht überlappen. Deshalb wird das Skript blockorientiert dargestellt. Blockorientiert bedeutet dabei, daß der Kontrollfluß eines Sprachkonstrukts nur einen Eingang und einen Ausgang besitzt. In herkömmlichen Programmiersprachen wird dies z.B. durch die Einschachtlung von Anweisungen in „begin“ und „end“ erreicht. Jedes hier verwendete Sprachkonstrukt ist so aufgebaut, daß es nur einen Eingang und einen Ausgang aufweist. Des weiteren wird jedem Sprachkonstrukt ein Zeichenbereich zur Verfügung gestellt. Dieser Zeichenbereich ist der Einfachheit halber rechteckig. Der Repräsentant des Sprachkonstrukts ist für seinen Zeichenbereich zuständig. Dazu sind folgende Fähigkeiten nötig:

- Der Repräsentant muß sich zeichnen können.

- Er ist für die Positionierung der Sprachkonstrukte, die er umfaßt zuständig. Dies beinhaltet, daß sich die Zeichenbereiche der eingeschlossenen Sprachkonstrukte nicht überschneiden dürfen, da sonst die Kreuzungsfreiheit und die Überlappungsfreiheit nicht gewährleistet werden kann.
- Er muß über seinen Platzbedarf Bescheid wissen. Dies ist wichtig, damit das umgebende Sprachkonstrukt die Positionierung dieses Sprachkonstrukts korrekt durchführen kann.

Genau diese Eigenschaften werden durch die abstrakte Klasse GraphObj abgedeckt (siehe Klassenhierarchie des Modells). Ein Problem bei der Darstellung des Skriptes sind die Verzweigungen nach Konfliktbehandlungen. Diese entsprechen den Goto-Sprüngen in herkömmlichen Programmiersprachen. Deshalb kann hierfür keine Kreuzungsfreiheit garantiert werden, da sie aber eher die Ausnahme darstellen, können diese auch separat gehandhabt werden.

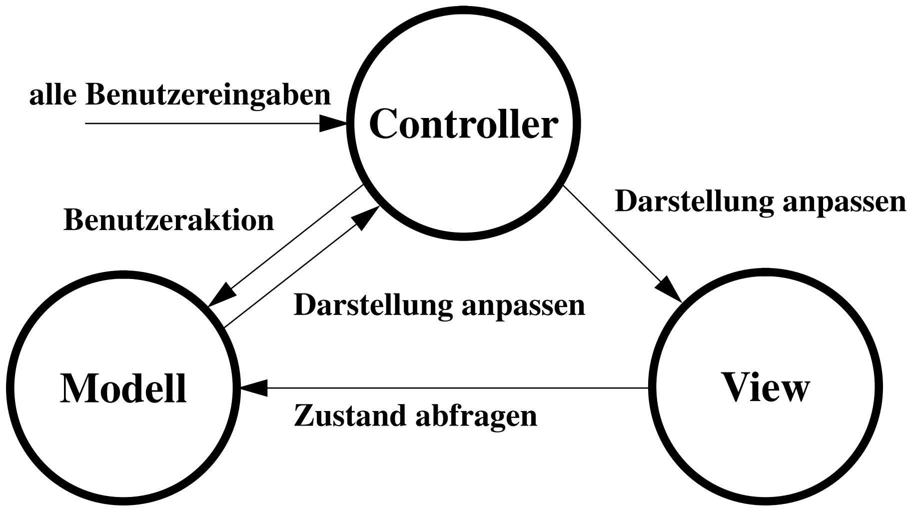
4.5 Das MVC-Konzept

Beim Entwurf und Anbindung der Oberfläche wurde das von Smalltalk bekannte *Modell-View-Controller*-Konzept (MVC) verwendet. Es dient der klaren Zuordnung der Zuständigkeiten für Aufgaben bei der interaktiven Kommunikation mit dem Benutzer. Dabei ist

- das *Modell* für die Daten im Programm,
- die *View* ist für das Aussehen der graphischen Benutzeroberfläche auf dem Bildschirm und
- der *Controller* für die Kommunikation mit dem Benutzer und zwischen *Modell* und *View*

zuständig. Ein wesentlicher Vorteil dieses Konzeptes ist, daß das *Modell* ohne Änderungen am *Controller* und der *View* ausgetauscht werden kann, sofern sich die Schnittstelle nicht ändert. Das gleiche gilt natürlich auch für die *View.* Auch wenn sich die Darstellung auf dem Bildschirm ändert, müssen weder *Controller* noch *Modell* angepaßt werden. D.h. heißt die Kopplung zwischen *Modell*, *View* und *Controller* ist gering. Das Prinzip des MVC-Konzepts ist in nachfolgender Abbildung nochmals zusammengefaßt:

ABBILDUNG 3. MVC-Konzept



Die Entscheidung zur Verwendung des MVC-Konzepts hat natürlich auch Auswirkungen beim Entwurf mit sich gebracht. Die Aufteilung in *Modell*, *View* und *Controller* sieht in dieser Arbeit wie folgt aus:

- *Modell*: Die Aufgabe des Modells übernimmt die Darstellung des Parsebaums des Programms, also des Skriptes für den Kontrollfluß.
- *View*: Diese Aufgabe wird von dem Code übernommen, der mit dem Interface Composer erstellt wurde. Bei dem erstellten Code handelt es sich nicht um Klassen mit Methoden, sondern um Code zur Erzeugung von Elementen der GUI. Die Methoden zur Manipulation des GUIs stellen die X- und die Motif-Bibliothek zur Verfügung.
- *Controller*: Diese Aufgabe wird von den Klassen übernommen, die von der Klasse View abgeleitet sind. Daß diese Klassen „Views" heißen, ist eine Inkonsistenz in der Namengebung, die daher rührt, daß die View-Klassen uneingeschränkten Zugriff auf die Darstellung (*View)* haben, die vom Interface Composer erstellt wird, und somit *Controller* und *View* quasi verschmelzen. Dies war nötig, um die Integration des vom Interface Composers generieten Codes zu gewährleisten. In den View-Klassen sind die erlaubten Benutzeraktionen enthalten, die zu Änderungen im Modell und der Darstellung (*View*) führen können. Deshalb können die View-Klassen als *Controller* nach dem MVC-Konzept betrachtet werden.

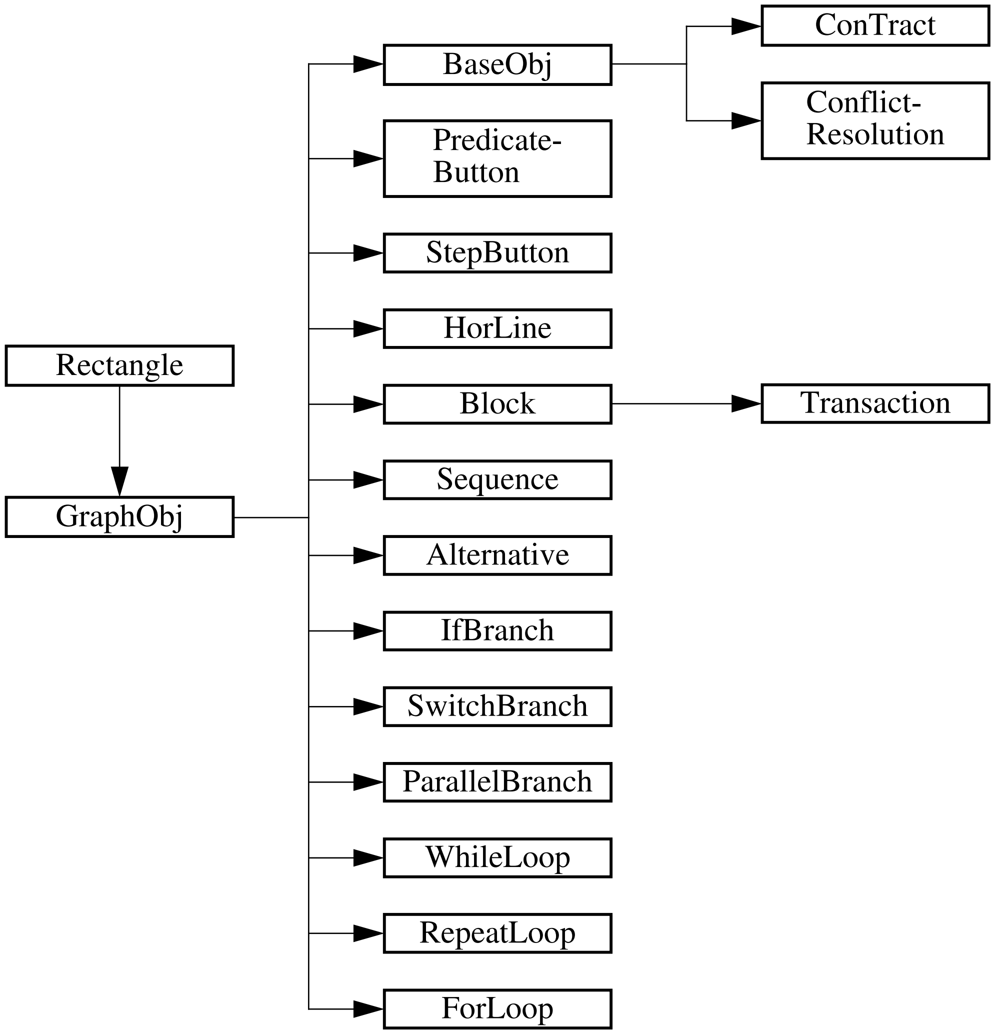
4.6 Die Klassenhierarchie des Modells

Zur Implementierung des *Modells* ist folgendes zu sagen. Das *Modell* sollte folgenden Eigenschaften aufweisen:

- Da die Ausgabe des Editors als Eingabe für den Compiler dient, ist es von Vorteil, wenn das Modell einen Parsebaum wiederspiegeln würde.
- Es sollte alle Daten enthalten, die für das Aussehen des Skriptes verantwortlich sind, damit eine Speicherung und Wiederherstellung des Skriptes problemlos möglich ist. Diese Eigenschaften sollten nach dem MVC-Konzept normalerweise in der *View* enthalten sein. Der Einfachheit halber wurden die Eigenschaften des Skriptes aber in das *Modell* übernommen.

Dem *Modell* liegt ein Klassenhierarchie zugrunde, die folgender Abbildung zu sehen ist. Dabei bedeute ein Pfeil: „ist Superklasse von".

ABBILDUNG 4. Klassenhierarchie des Modells

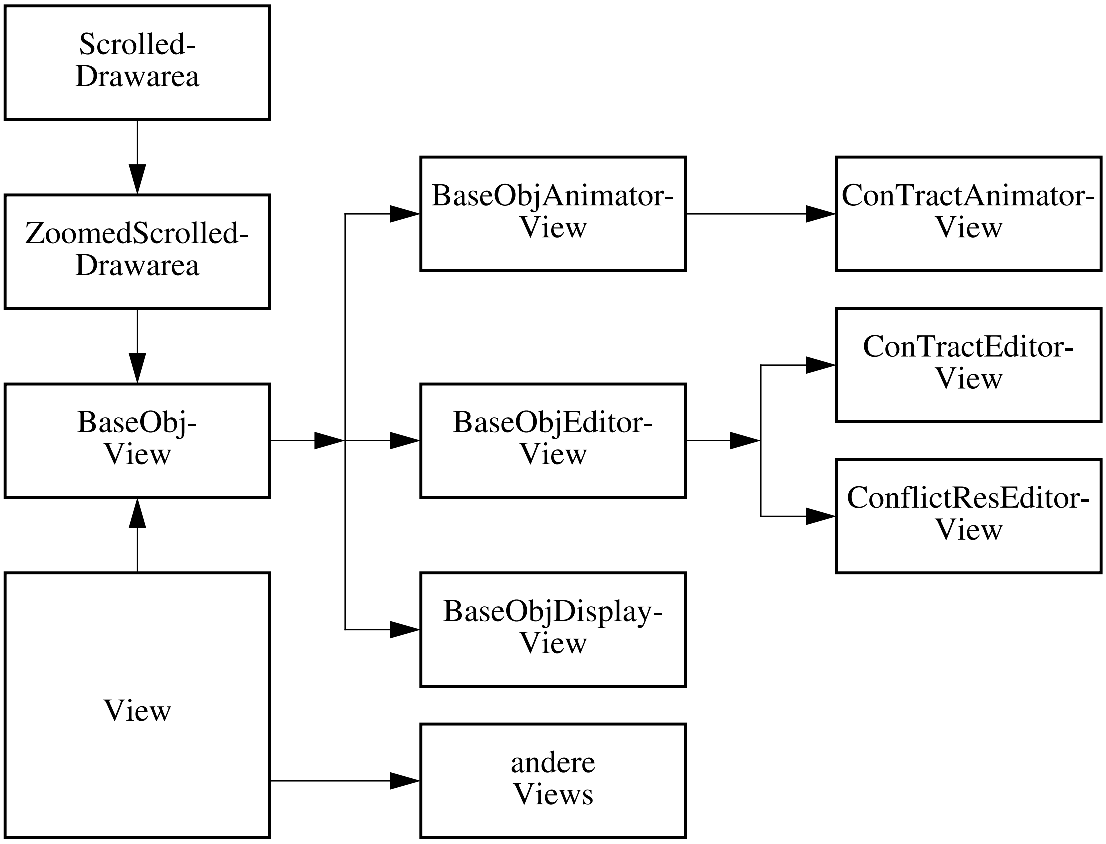


Die Klassenhierarchie orientiert sich dabei an den oben beschriebenen Kriterien zur Darstellung eines Skriptes. Den Kern dieser Hierarchie bildet dabei die Klasse GraphObj. Sie stellt die Basisklasse für alle graphischen Objekte dar, die in einem Skript angezeigt werden. Jede von ihr abgeleitete Klasse ist ein Repräsentant für ein Programmiersprachen-Konstrukt. Die Klasse GraphObj ist selbst Subklasse der Klasse Rectangle. Die Klasse Rectangle beschreibt die Eigenschaften und Methoden von Rechtecken, da es sich der Einfachheit halber um einen rechteckigen Zeichenbereich handeln soll. Die Subklasse BaseObj von Klasse GraphObj kommt eine besondere Aufgabe zu. Sie dient als Kommunikationsschnittstelle zum *Controller* (View-Klassen). Dies kommt zustande, da jedem ConTract und jeder ConflictResolution genau eine *View* zugeordnet ist. Alle Objekte, die unterhalb dieser Konstrukte im Parsebaum hängen, benutzen die gleiche *View* für ihre Ausgaben.

4.7 Die Klassenhierarchie der Controller

Die Klassenhierarchie der Controller ist etwas komplizierter. Eine graphische Übersicht wird in nachfolgender Abbildung gegeben. Die Pfeile bedeuten dabei: „ist Superklasse von".

ABBILDUNG 5. Klassenhierarchie der Controller



Die Klasse View dient als Basisklasse für alle *Controller*. Die Klasse ScrolledDrawarea beschreibt die Methoden und Eigenschaften eines Zeichenbereichs, der in einem verschiebbaren Fenster liegt. Für diesen Zeichenbereich sind relativ primitive Zeichenroutinen definiert, die auf Bildschirmkoordinaten beruhen. Die Klasse ScrolledDrawarea stellt eine Kapselung zu den primitiven Zeichenroutinen der X-Windows-Bibliothek dar. Um eine größenunabhängige Darstellung der Zeichnung zu bekommen, ist von der Klasse ScrolledDrawarea die Klasse ZoomedScrolledDrawarea abgeleitet. Diese Klasse umfaßt dann Methoden, die auf bildschirmunabhängigen Koordinaten arbeiten. Mit dieser Klasse kann bestimmt werden, wie groß die Darstellung der Zeichnung sein soll. Die nächste Stufe, die Klasse BaseObjView, ist eine völlig abstrakte Klasse, die zur Kommunikation mit dem *Modell* gedacht ist. Von der Klasse BaseObjView werden nun die Klassen abgeleitet, die eine Spezialisierung von der Basisklasse darstellen. Deren Funktion kann anhand der Namen relativ leicht nachvollzogen werden. Unter dem Begriff „andere Views“ sind z.B. alle Dialogboxen zusammengefaßt.

4.8 Die Datensicherung

4.8.1 Warum wird eine Datenbank verwendet?

Nachdem das Skript eingegeben wurde, muß es auch dauerhaft gesichert werden. Dazu stehen grundsätzlich zwei Möglichkeiten zur Verfügung:

- die Sicherung in einem Dateisystem

- die Sicherung in einem Datenbanksystem

Die Sicherung in einem Dateisystem hat den Nachteil, daß ein Dateisystem nicht so zuverlässig und fehlertolerant arbeitet wie Datenbanksysteme. Die Vorzüge der Verwendung eines Datenbanksystem sind:

- Die Zugriffe auf die Datenbank sind transaktionsgeschützt.
- Es steht eine standardisierte Zugriffssprache zur Verfügung (SQL).
- Die interne Darstellung ist für den Benutzer uninteressant, er arbeitet nur auf Relationenbasis.
- Zum Teil werden Datenbankabfragen vom System selbständig optimiert.
- Die heutigen Datenbanksysteme haben einen hohen Grad an Zuverlässigkeit erreicht.

Die Nachteile von Datenbanksystemen dürfen nicht verschwiegen werden. Diese wären z.B.:

- Die Zugriffszeiten sind im allgemeinen schlechter als bei reinen Zugriffen auf ein Dateisystem. Aber dies soll kein Grund sein, auf die Sicherheit von Datenbanksystemen zu verzichten.
- Ein Datenbanksystem ist nicht dazu geschaffen, viele Daten unterschiedlichen Typs zu verarbeiten. Die Datenbanksysteme sind darauf spezialisiert, viele Daten weniger Typen zu verarbeiten.
- Die heutzutage verfügbaren Datenbanksysteme sind im allgemeinen nicht in der Lage sogenannte *long-fields* zu verarbeiten. Dieser Nachteil hat keine Auswirkungen auf die Studienarbeit, da keine *long-fields* benötigt wurden.

Eine Alternative zu der Verwendung eines Datenbanksystem käme auch noch ein transaktionsgeschütztes Dateisystem in Frage, sie sind allerdings noch nicht so verbreitet wie Datenbanksysteme. Die Verwendung eines objektorientierten Datenbanksystems hätte die Speicherung des Skriptes etwas vereinfacht. Da objektorientierte Datenbanksysteme noch nicht standardisiert sind und nur wenig verbreitet sind, ist der Einsatz wenig sinnvoll.

4.8.2 Mehrbenutzerbetrieb

Der transaktionale Zugriff auf Daten hat einen entscheidenden Nachteil. Der Zugriff mehrerer Benutzer zur gleichen Zeit auf die gleiche Relation ist stark beschränkt, da das Datenbanksystem, die Relation, in der die zu bearbeitenden Daten liegen, für den Zugriff für die Dauer der Transaktion sperrt. Dies stellt vor allem dann ein Problem dar, wenn die Transaktionen langandauernd sind. Eine langandauernde Transaktion ist z.B. das Editieren des Skriptes eines ConTracts. Die Änderungen am Skript sollten nur dann sichtbar werden, falls die Änderungen zur Zufriedenheit des Benutzers ausgefallen sind. Dies würde den Mehrbenutzerbetrieb erheblich einschränken, denn dann könnte kein anderer Benutzer einen anderen ConTract bearbeiten. Deshalb wurde ein „Check-In-Check-Out“-Mechanismus verwendet. D.h. die Daten eines Skriptes werden zum Beginn der Bearbeitung aus der Datenbank in den Speicher geladen und nach der Durchführung von Änderungen wieder in die Datenbank zurückgeschrieben. Das bedeutet, daß nur für die Dauer der Schreib- bzw. Lesevorgänge Transaktionsschutz benötigt wird. Deshalb können auch mehrere Benutzer an ConTracts arbeiten. Diese Methode hat auch den Vorteil, daß die Änderungen am Skript nur im Speicher vonstatten gehen, was einen relativ hohen Geschwindigkeitsvorteil mit sich bringt. Diese vorgehensweise hat aber einen zwei entscheidende Nachteile:

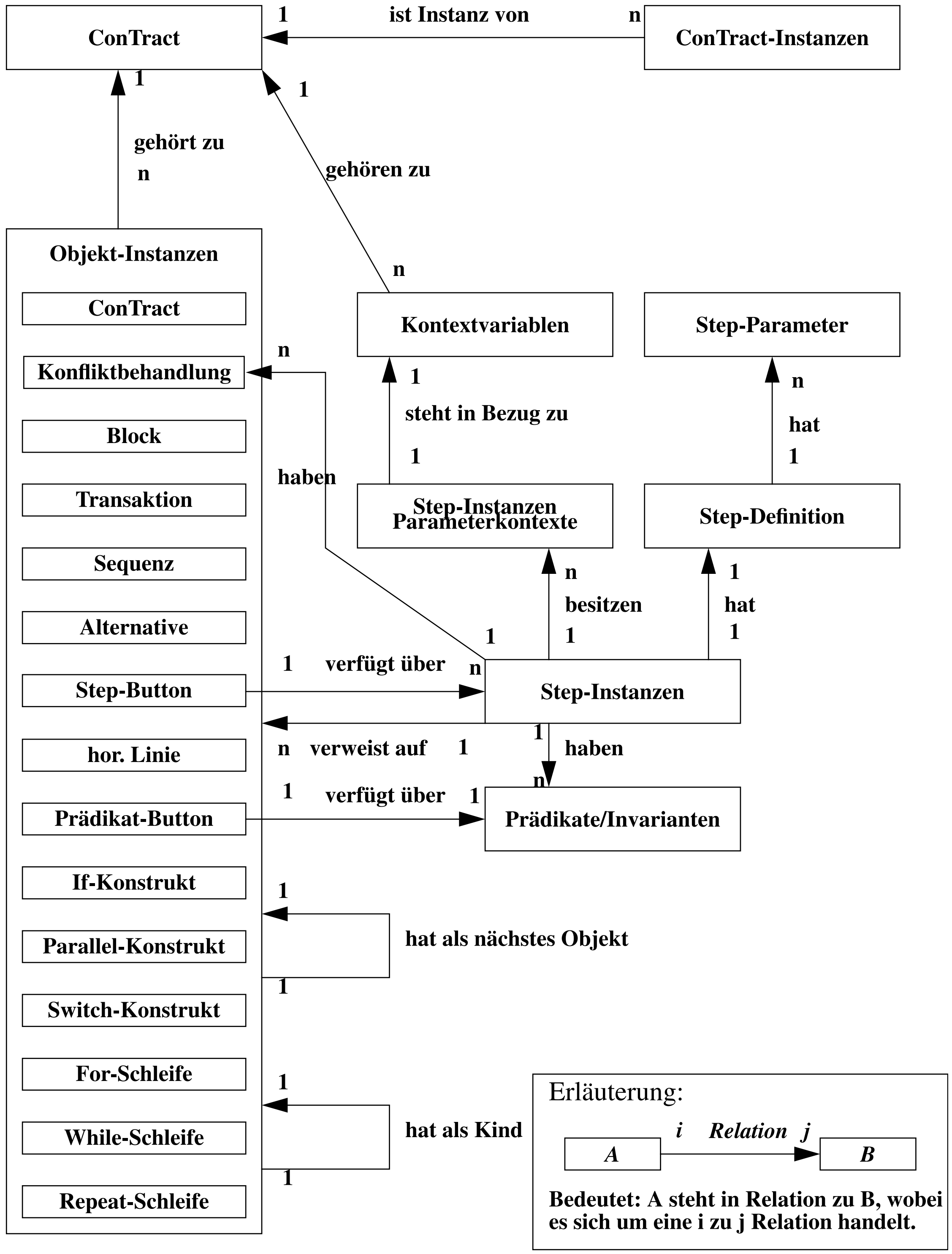
- Die automatischen Sperren des Datenbanksystems fallen weg. D.h. es könnten mehrere Benutzer die Daten des gleichen ConTracts verändern. Deshalb wurde ein logischer Sperrmechanismus eingeführt. Dieser funktioniert wie folgt: Wenn ein Benutzer die Bearbeitung eines ConTract wünscht, wird in der Datenbank vermerkt, daß dieser ConTract gerade in Bearbeitung ist und daß niemand anderes auf diesen ConTract zugreifen darf. Nach Beendigung der Arbeit des Benutzer wird dieser Vermerk wieder gelöscht.
- Falls der Editor oder der Rechner, auf dem der Editor läuft, während der Bearbeitung des ConTracts abstürzt, sind alle Änderungen verloren. Dieser untragbare Zustand kann abgefedert werden, wenn dem Benutzer erlaubt wird, die Änderungen von Zeit zu Zeit automatisch oder explizit abspeichern zu lassen. Da alle Änderungen des Benutzers erst nach Ablauf des Editiervorgangs sichtbar werden sollen, ist es nötig, beim Beginn des Editiervorgangs eine Kopie des ConTracts anzulegen. Auf dieser Kopie kann der Benutzer dann alle Änderungen abspeichern, ohne daß sich der Originalzustand des ConTracts ändert. Am Ende des Editiervorgangs wird dem Benutzer die Wahl gelassen, ob seine Änderungen eintreten sollen oder ob die Änderungen ignoriert, also weggeworfen werden sollen.

Zur Verbesserung des Mehrbenutzerbetriebs wurde eine einfache Versionenverwaltung eingeführt, die es erlaubt, mehrere Versionen eines ConTracts zu verwalten und zu bearbeiten.

4.8.3 Entity-Relationship-Datenmodellierung

Zur Datenmodellierung für ein Datenbanksystem bietet sich die Entity-Relationship-Modellierung an. Zur Darstellung werden Entity-Relationship-Diagramme eingesetzt, um die Relation zwischen den Datentypen darzustellen. Aufgrund dieser Darstellung versucht man Optimierungsmöglichkeiten zu finden. In der nächsten Abbildung ist das Datenmodell dieser Studienarbeit zu sehen. Die Attribute der Datentypen wurden aus Gründen der Übersichtlichkeit weggelassen. Die Attribute können aus Tabellen zu den Relationen im Anhang entnommen werden.

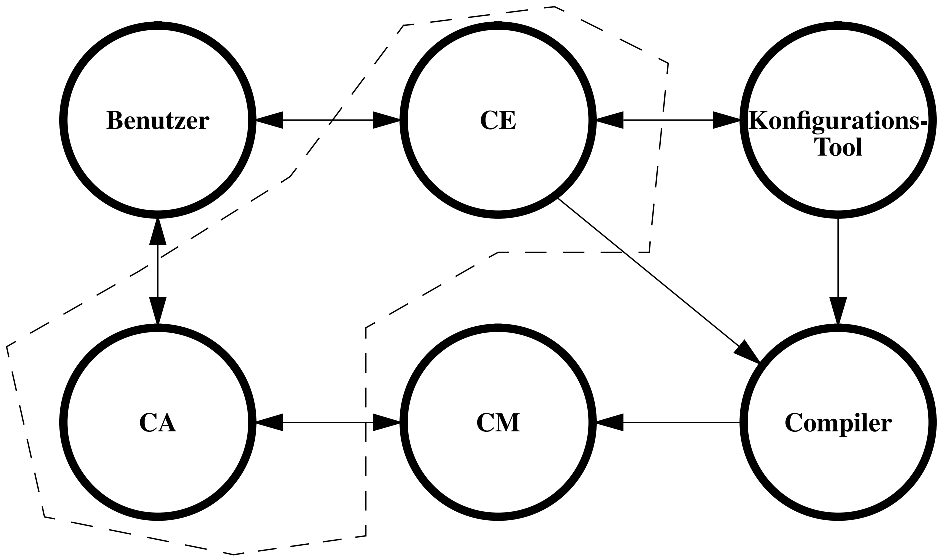
ABBILDUNG 6. Entity-Relationship-Diagramm



5 Schnittstellen

Nun muß noch geklärt werden wie die Schnittstellen zur Außenwelt aussehen. In der nachfolgenden Abbildung werden die vorhandenen Schnittstellen sichtbar. Der Kern des Systems bilden der ConTract-Editor und ConTract-Animator, die schon zum obengenannten ConTract-Development-Tool gehören. Sie sind gestrichelt umrandet. Immer dort, wo ein Pfeil von der gestrichelten Linie durchschnitten wird, ist eine Schnittstelle vorhanden.

ABBILDUNG 7. Schnittstellen



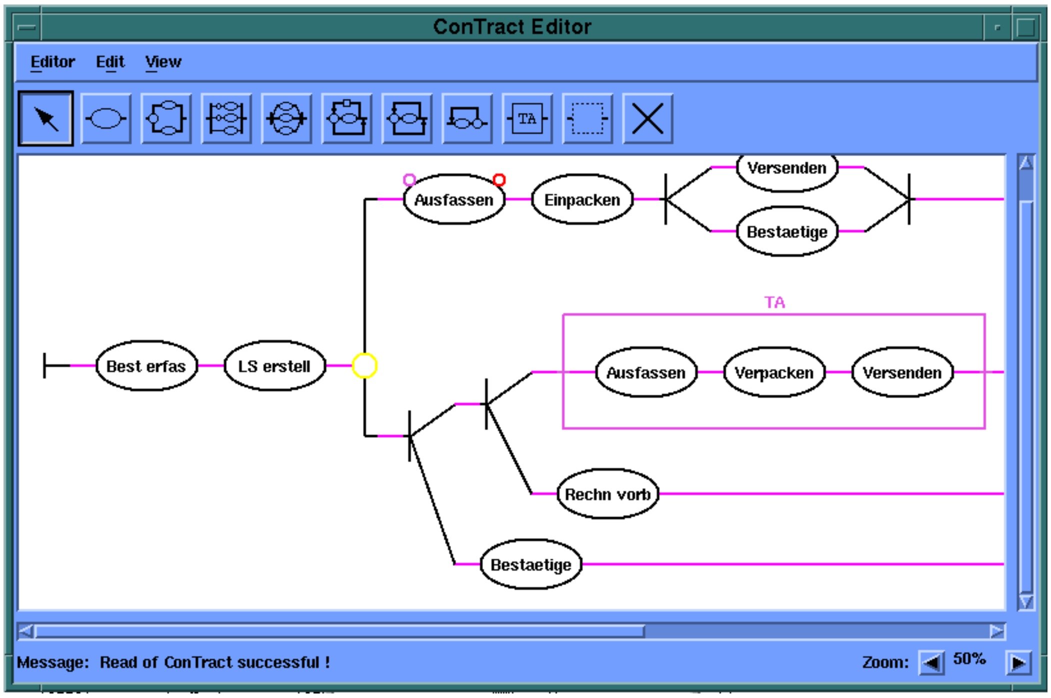
5.1 Die Schnittstelle zum Benutzer

Als Beispiel für eine Schnittstelle zum Benutzer wird hier der ConTract-Editor angeführt, da er als Hauptbestandteil der Studienarbeit aufgefaßt werden kann. Der ConTract-Editor zeigt sich dem Benutzer wie in der folgenden Abbildung. Es werden zwei Konzepte verwendet, um einen ConTract zu erstellen:

- Der Benutzer kann ein Sprachkonstrukt z.B. Step, For-Schleife an der Stelle einfügen lassen, an die er mit dem Mauszeiger klickt. Natürlich nicht beliebig, sonst würde die Blockstruktur zerstört. Dieser Ansatz dient dem Top-Down-Programmieren eines Skriptes. Das bedeutet, der Inhalt des eingefügten Konstruktes wird nachher ergänzt.
- Die zweite Möglichkeit beim Editieren besteht darin, Konstrukte zu selektieren, um diese dann zu komplexeren Konstrukten z.B. einer Transaktion zusammenzufassen. Diesen Ansatz könnte man als Bottom-Up-Programmieren bezeichnen. Die selektierten Objekte müssen dabei einige Eigenschaften erfüllen, damit ein vernünftiges Resultat entsteht. Diese Eigenschaften wären:
  - Die Konstrukte müssen in der gleichen Sequenz liegen.
  - Die Konstrukte liegen direkt aneinander, d.h. sie sind nur durch horizontale Linien getrennt.

Diese zwei Konzepte und die Möglichkeit Konstrukte zu löschen sind ausreichend, um ein Skript zu erstellen. Da sich die Implementierung sich an dem Motif-Style-Guide hält, kann man sich zu den meisten Konstrukten über die rechte Maustaste zusätzliche Attribute anzeigen lassen und sie gegebenfalls editieren.

ABBILDUNG 8. ConTract-Editor



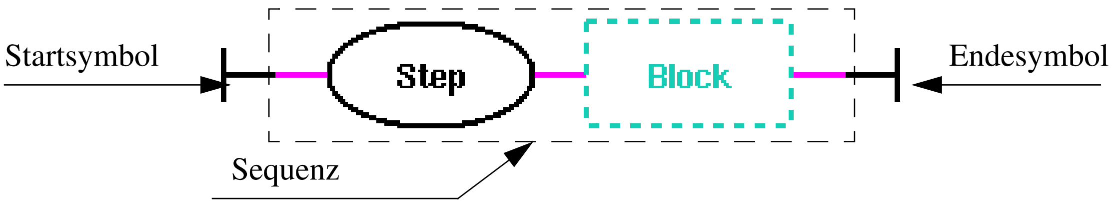
5.2 Die Schnittstelle zum Compiler

Die Schnittstelle zum Compiler bildet ein in der Datenbank abgelegter Parsebaum. Die Struktur des Parsebaum richtet sich genau nach oben beschriebenen Syntax. Zur Verdeutlichung sind im folgenden nochmals alle Konstrukte der Programmiersprache anhand ihrer Darstellung auf der Oberfläche aufgeführt. Zu jedem Konstrukt wird der zugehörige Parsebaum dargestellt. Als Namen der Konstrukte sind die Namen der Klassen angegeben, durch die sie repräsentiert werden (siehe Klassenhierarchie).

5.2.1 ConTract

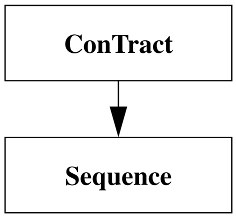
Ein ConTract wird wie in der nachfolgenden Abbildung zu sehen ist, auf dem Bildschirm visualisiert.

ABBILDUNG 9. ConTract



Da ein ConTract aus einer Sequenz von Anweisungen besteht, sieht der Parsebaum wie folgt aus:

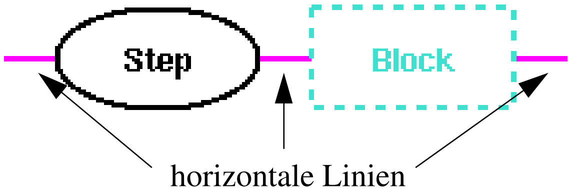
ABBILDUNG 10. ConTract als Parsebaum



5.2.2 Sequence

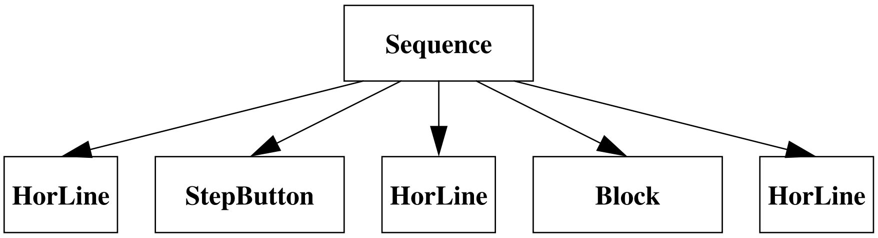
Die Darstellung der sequentiellen Ausführung von Konstrukten sieht auf der Oberfläche wie folgt aus:

ABBILDUNG 11. Sequenz



Eine Sequenz kann aus beliebig vielen Konstrukten zusammengesetzt sein. Die Konstrukte werden immer durch eine horizontale Linie getrennt. Das obige Beispiel sieht als Parsebaum wie folgt aus:

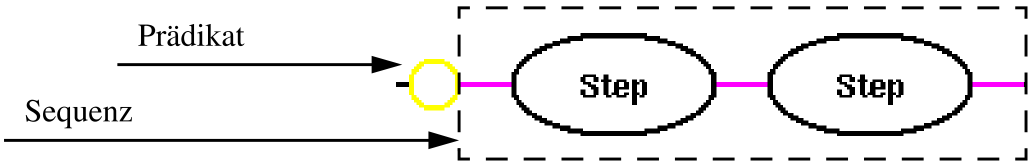
ABBILDUNG 12. Sequenz als Parsebaum



5.2.3 Alternative

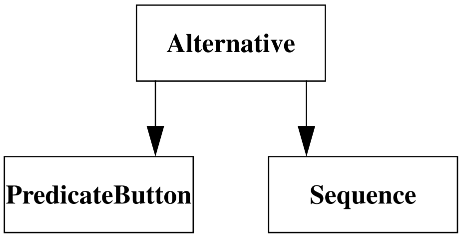
Die Alternative wird im Switch-Konstrukt verwendet. Es beschreibt eine Sequenz, die nur ausgeführt wird, falls dessen Prädikat nach True ausgewertet wird. Auf der Oberfläche sieht die Alternative wie folgt aus:

ABBILDUNG 13. Alternative



Die Parsebaum-Darstellung fällt auch nicht kompliziert aus, nämlich:

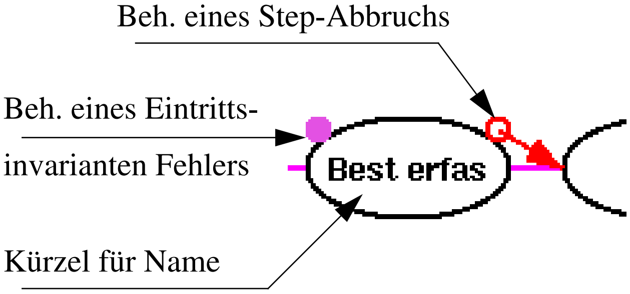
ABBILDUNG 14. Alternative als Parsebaum



5.2.4 StepButton

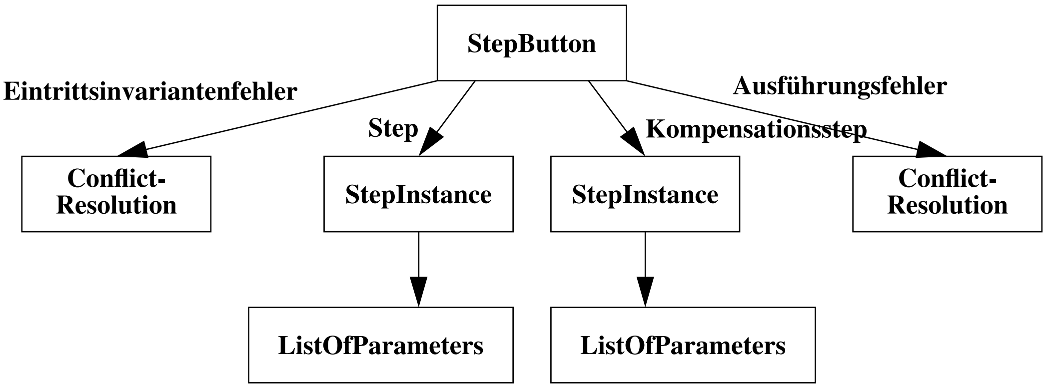
Die Darstellung eines StepButton in einem Skript sieht wie folgt aus.

ABBILDUNG 15. StepButton



Im Parsebaum wird die interne Struktur sichtbar, die durch das StepButton-Konstrukt verdeckt wird. Jedem StepButton ist eine Step-Instanz für den normal auszuführenden Step und eine Step-Instanz für den Kompensations-Step zugeordnet. Die Step-Instanz umfaßt Attribute wie Name, Eintrittsinvariante, Liste von Parametern usw.. Außerdem können für einen Step zwei Konfliktbehandlungen angegeben werden: Eine Konfliktbehandlung für einen Eintrittsinvariantenfehler und eine für einen Ausführungsfehler des Steps. Der Parsebaum sieht wie folgt aus:

ABBILDUNG 16. StepButton als Parsebaum



5.2.5 HorLine

Die horizontale Linie findet man in jeder Sequenz. Sie dient der Trennung der Konstrukte, und zum Ausgleich von unterschiedlich langen Parallelzweigen im Parallel-Konstrukt und im Switch-Konstrukt. Die Linien sind vorhanden, damit der Benutzer weiß, wo er neue Konstrukte einfügen kann.

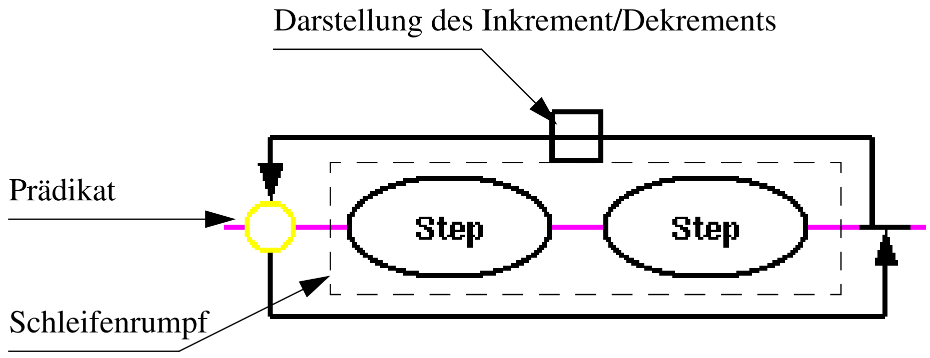
5.2.6 PredicateButton

Der PredicateButton taucht überall dort auf, wo ein Prädikat benötigt wird, also bei den Schleifen und den bedingten Verzweigungen. Ein Prädikat kann nur die Werte True oder False annehmen und beeinflußt damit den Kontrollfluß.

5.2.7 ForLoop

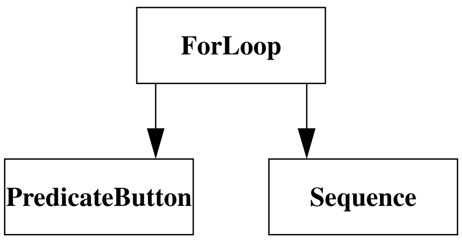
Die Darstellung einer For-Schleife sieht wie folgt aus:

ABBILDUNG 17. For-Schleife



Die For-Schleife sieht in der Parsebaum-Darstellung wie folgt aus:

ABBILDUNG 18. For-Schleife als Parsebaum



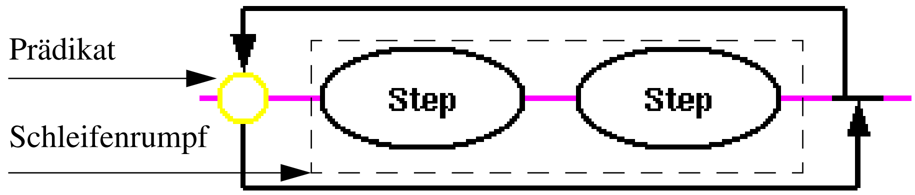
Der Element PredicateButton hat als Attribut das Prädikat der For-Schleife. Als Kind besitzt es eine Sequenz, die die Konstrukte des Schleifenrumpfes enthält.

Zur Ausführung: Zu Beginn der Schleife wird der Schleifenzähler, der Bestandteil des Prädikats ist, initialisiert. Danach prüft die Prädikatauswertung auf die im Prädikat enthaltene Schranke ab. Liefert die Prädikatauswertung True wird der Schleifenrumpf ausgeführt, andernfalls wird der Schleifenrumpf übersprungen und nach der Schleife im Kontrollfluß wieder aufgesetzt. Nach der Abarbeitung des Schleifenrumpfes wird der Schleifenzähler, der Bestandteil des Prädikats ist, inkrementiert bzw. dekrementiert. Daraufhin wird erneut das Prädikat ausgewertet.

5.2.8 WhileLoop

Die While-Schleife wird in dem Skript wie folgt repräsentiert. Die Parsebaum-Darstellung ist entsprechend der Darstellung der For-Schleife.

ABBILDUNG 19. While-Schleife

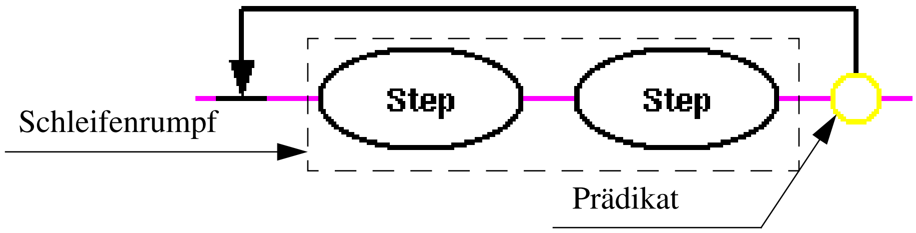


Zur Ausführungssemantik: Die Bedingung zur Schleifenwiederholung wird vor Abarbeitung der Schleife geprüft. Der Schleifenrumpf wird solange ausgeführt wie das Prädikat nach True ausgewertet wird.

5.2.9 RepeatLoop

Eine Repeat-Schleife wird im Skript wie folgt dargestellt. Die Parsebaum-Darstellung der Repeat-Schleife entspricht quasi die der For-Schleife.

ABBILDUNG 20. Repeat-Schleife

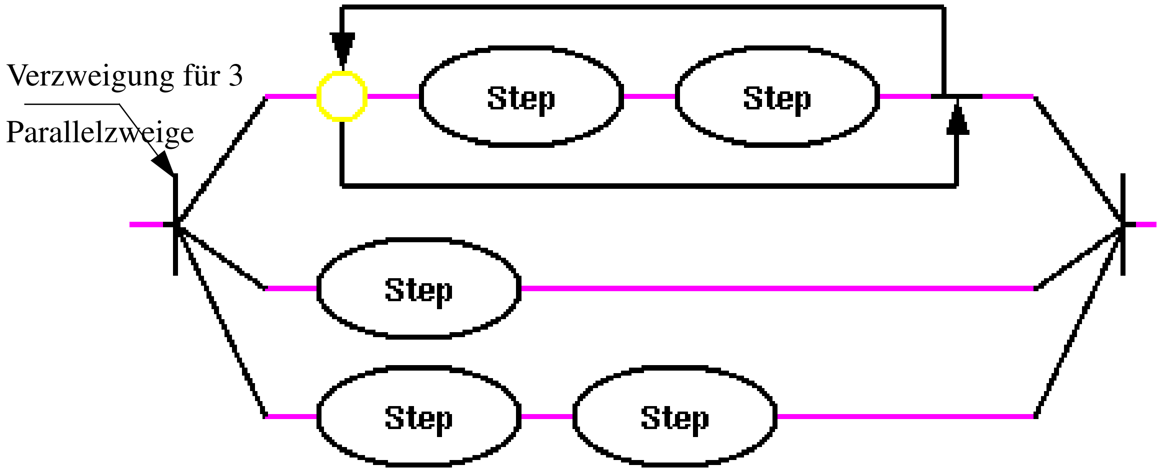


Zur Ausführungssemantik: Die Bedingung zur Schleifenwiederholung wird erst nach Abarbeitung der Schleife geprüft. Der Schleifenrumpf wird solange ausgeführt wie das Prädikat nach False ausgewertet wird.

5.2.10 ParallelBranch

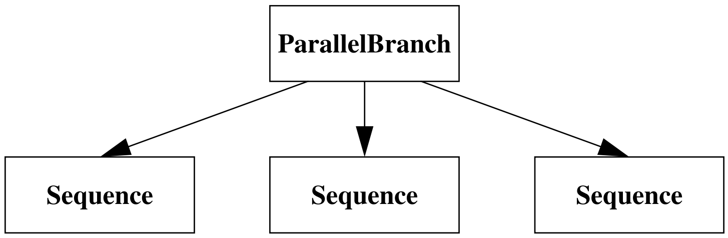
Die Darstellung eines Parallel-Konstrukts mit drei Parallelzweigen sieht auf der Oberfläche wie folgt aus:

ABBILDUNG 21. Parallel-Konstrukt



Jeder Parallelzweig entspricht einer Sequenz. Der Parsebaum für obiges Beispiels wie folgt aus:

ABBILDUNG 22. Parallel-Konstrukt als Parsebaum

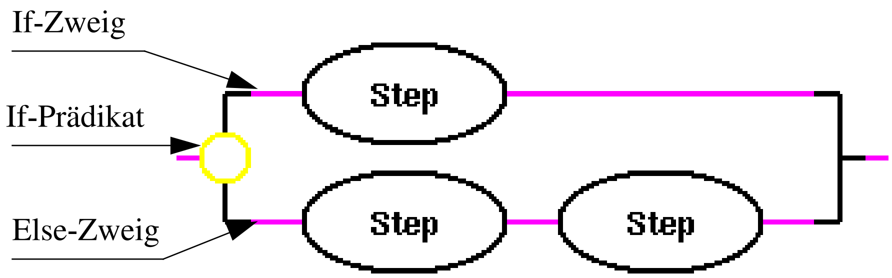


Zur Ausführungssemantik: Das Parallelkonstrukt kann beliebig viele Zweige haben, die alle parallel ausgeführt werden.

5.2.11 IfBranch

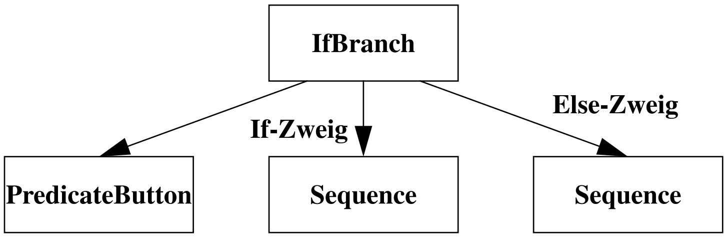
Das If-Konstrukt wird wie folgt dargestellt:

ABBILDUNG 23. If-Konstrukt



Der Parsebaum besteht aus folgenden Teilen:

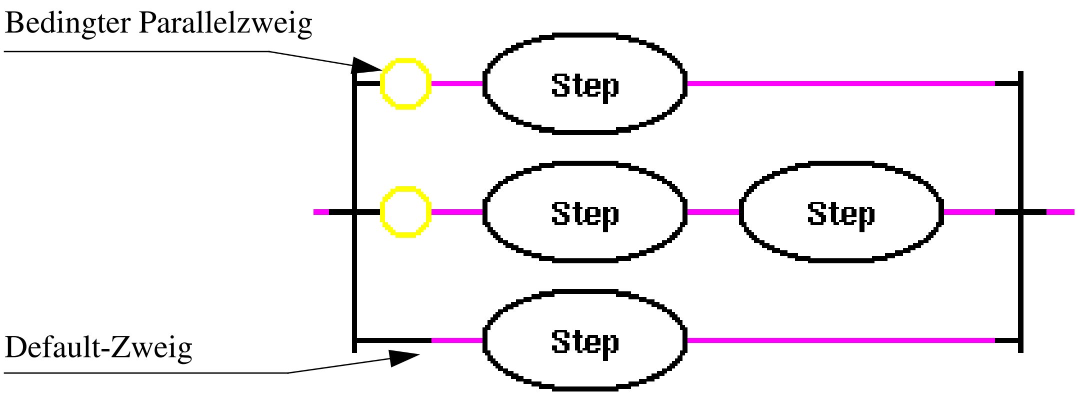
ABBILDUNG 24. If-Konstrukt als Parsebaum



5.2.12 SwitchBranch

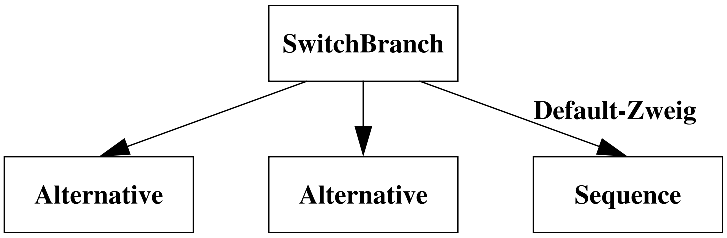
Die bedingte Mehrfachverzweigung sieht auf der Oberfläche wie folgt aus:

ABBILDUNG 25. Switch-Konstrukt



Das Switch-Konstrukt kann beliebig viele bedingte Zweige besitzen. Die bedingten Zweige sind Konstrukte vom Typ Alternative. Der Default-Zweig ist, da er unbedingt ist, eine Sequenz. Die Darstellung als Parsebaum des obigen Beispiels ist in nachfolgender Abbildung zu sehen:

ABBILDUNG 26. Switch-Konstrukt als Parsebaum

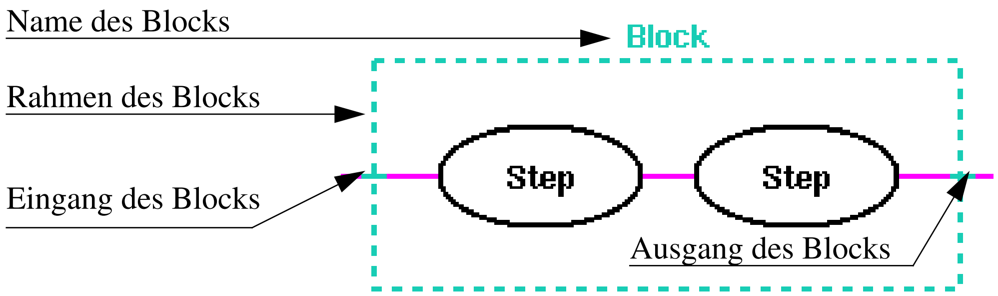


Zur Ausführungssemantik: Alle Zweige werden parallel angestartet. Das Prädikat steuert, ob die darauffolgende Sequenz von Anweisungen ausgeführt wird oder nicht. Die Anweisungen des Default-Zweigs werden in jedem Fall ausgeführt.

5.2.13 Block

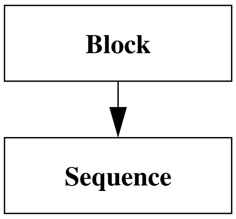
Der Block besitzt auf der Oberfläche eigentlich zwei Darstellungsvarianten. Zum einen kann er, wie in der nächsten Abbildung zu sehen ist, auftreten. Zum anderen kann der Block auch kollabiert dargestellt werden, d.h. es wird nur ein Repräsentant für den Block gezeichnet. Die im Block enthaltenen Konstrukte sind dann nicht sichtbar.

ABBILDUNG 27. Block



Die Parsebaum-Darstellung ist natürlich nicht von diesen Darstellunsvarianten abhängig und sieht wie folgt aus:

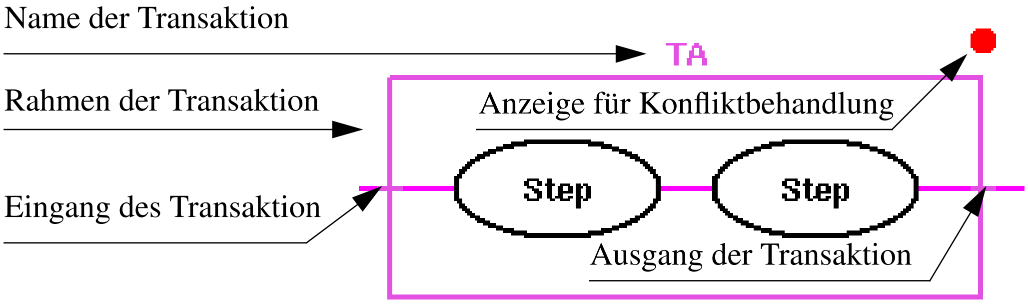
ABBILDUNG 28. Block als Parsebaum



5.2.14 Transaction

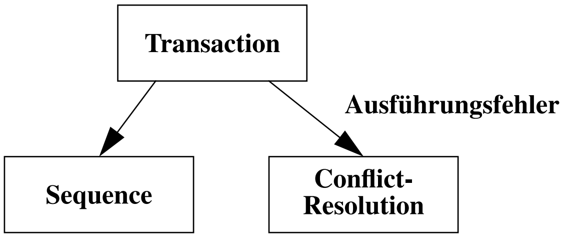
Auch die Transaktion besitzt wie der Block zwei Darstellungsvarianten. Hier auch wieder die expandierte Variante:

ABBILDUNG 29. Transaktion



Die interne Darstellung als Parsebaum besteht aus einer Sequenz und einer Konfliktbehandlung. Die Konfliktbehandlung ist optional und beschreibt, welche Aktionen nach einem Abbruch einer Transaktion ausgeführt werden sollen. Daraus resultiert folgende Darstellung des Parsebaums:

ABBILDUNG 30. Transaktion als Parsebaum

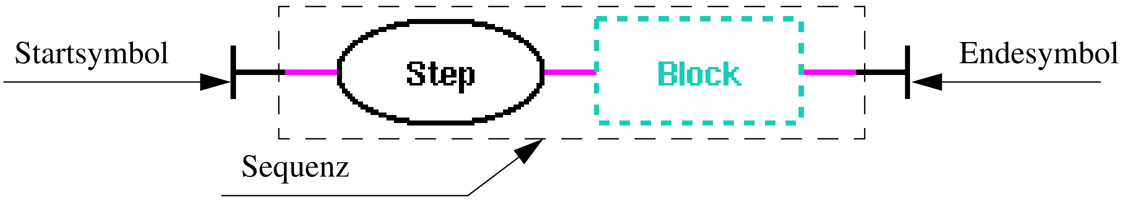


Zur Ausführungssemantik: Die in der Transaktion enthaltene Sequenz wird ganz normal ausgeführt, außer daß die Steps erst nach erfolgreicher Ausführung der gesamten Transaktion in den Commit-Zustand übergehen. Zur Realisierung der Transaktionssemantik wird das 2-Phasen-Commit-Protokoll eingesetzt. Tritt bei der Abarbeitung eines Steps ein Konflikt auf und kann dieser nicht innerhalb der Transaktion gelöst werden, werden allen Aktionen, die durch die Transaktion verrichtet wurden, wieder rückgängig gemacht (kompensiert). Daraufhin wird eine Konfliktbehandlung gestartet. Nach Abarbeitung der Konfliktbehandlung kann an einem fast beliebigen Punkt im Skript wieder aufgesetzt werden.

5.2.15 ConflictResolution

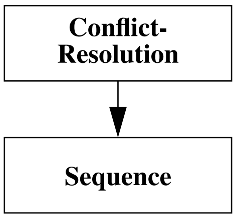
Die Konfliktbehandlung wie sie z.B. bei der Transaktion verwendet wird, umfaßt wieder ein komplettes Skript. Die Darstellung auf der Oberfläche sieht wie folgt aus:

ABBILDUNG 31. Konfliktbehandlung



Die Parsebaum-Darstellung ist ähnlich dem eines ConTracts, nämlich:

ABBILDUNG 32. Konfliktbehandlung als Parsebaum



5.3 Die Schnittstelle zum Kommunikationssystem

5.3.1 Welche Nachrichten müssen versendet werden?

Die ConTract-Animatoren sind für die für die Darstellung des Ablaufes zuständig. Damit der ConTract-Animator über den aktuellen Zustand des Ablaufes informiert ist, muß der ConTract-Manager Nachrichten an den ConTract-Animator versenden können. Auf der anderen Seite muß man vom ConTract-Animator die Kontrolle über den Ablauf haben. Deshalb muß der ConTract-Manager auch Nachrichten von der Oberfläche empfangen können. Diese Nachrichten sind, zum Beispiel daß der ConTract suspendiert oder daß die ConTract Bearbeitung abgebrochen werden soll. Zur Kommunikation zwischen den ConTract-Animatoren und den ConTract-Managern kommt natürlich nur das Kommunikationssystem in Frage.

5.3.1.1 Nachrichten vom ConTract-Manager zum ConTract-Animator

Damit der ConTract-Animator auch etwas mit der Nachricht anfangen kann, müssen dem ConTract-Animator folgende Informationen zur Verfügung stehen:

- die ConTract-Instanzen-Identifikation (CIID): damit kann der ConTract-Animator ermitteln, ob die Nachricht für ihn bestimmt ist.
- die Objekt-Identifikation (OID): jedem Konstrukt in der Anzeige ist intern eine eindeutige Nummer zugeordnet, damit es eindeutig adressiert werden kann.
- eine Aktion: sie legt fest, welche Aktion auf dem Objekt ausgeführt werden soll. Dies wäre zum Beispiel eine Zustandsänderung eines Steps, nachdem er erfolgreich abgearbeitet worden ist.

5.3.1.2 Nachrichten vom ConTract-Animator zum ConTract-Manager

Die Nachrichten sollten folgende Dinge enthalten:

- die ConTract-Instanzen-Identifikation (CIID): damit das Kommunikationssystem den richtigen ConTract-Manager ansprechen kann.
- eine Aktion: die vom ConTract-Manager auszuführende Anweisung z.B. suspendiere ConTract.

5.3.2 Problem bei der Kommunikation

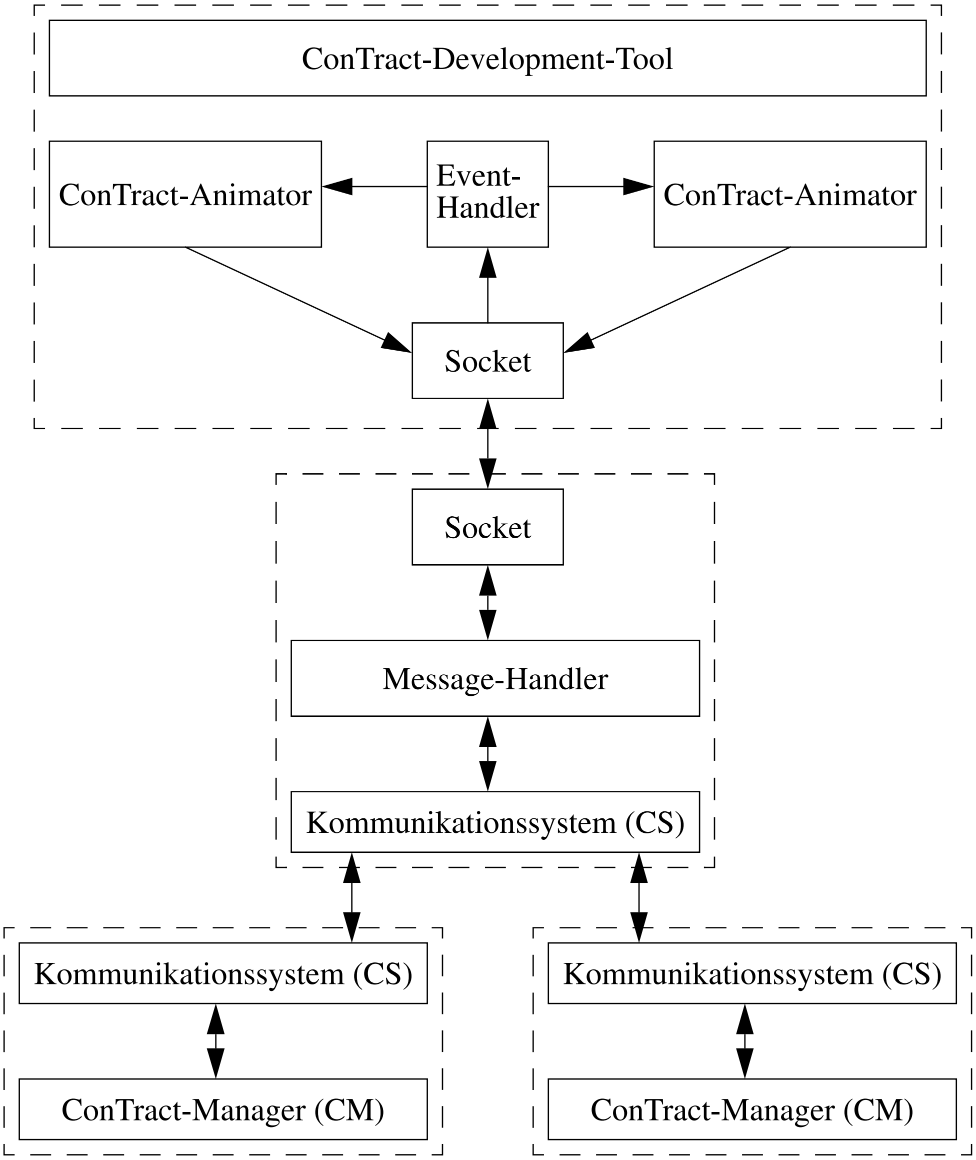
Das Problem bei der Kommunikation mit der Außenwelt besteht darin, daß man in einem ConTract-Development-Tool mehrere ConTract-Animatoren starten können soll. Das heißt, daß die Möglichkeit bestehen muß, von einem Arbeitsplatz aus, mehrere Abläufe überwachen zu können. Jeder der ConTract-Animatoren muß dabei auf Nachrichten von einem ConTract-Manager reagieren können. Das bedeutet, daß ein ConTract-Development-Tool mehrere Eingangskanäle hat. Um einen Busy-Wait zu verhindern, wurde folgende Lösung gewählt.

5.3.3 Lösung

Damit mit einem Programm (ConTract-Development-Tool) mehrere ConTract-Animatoren verwaltet werden können, läuft parallel ein Programm (Message-Handler) der asynchron zum ConTract-Development-Tool läuft und Nachrichten über das Kommunikationssystem empfängt und senden kann. Zur Kommunikation mit den ConTract-Animatoren wird ein

Socket eingesetzt. Die Verwendung eines Sockets hat den Vorteil, daß darauf mit X-Events gewartet werden kann. D.h. ein ConTract-Animator kann dem X-Server eine Funktion(Callback) angeben, die ausgeführt werden soll, wenn Daten im Socket angekommen sind. Zur Verdeutlichung folgende Abbildung:

ABBILDUNG 33. Einbindung der Ablaufkomponente



Der ConTract-Manager verschickt seine Nachrichten mit Hilfe des Kommunikationssystems an den Message-Handler. Dieser wandelt die Nachrichten um und legt die Nachricht im Socket ab. Jeder der ConTract-Animatoren hängt einen Callback ein, der auf Nachrichten, die am Socket ankommen, wartet. Im Callback prüft der ConTract-Animator, ob die Nachricht für ihn bestimmt ist und wertet gegebenfalls die Nachricht aus. Die andere Richtung funktioniert problemloser. Jeder ConTract-Animator kann seine Nachrichten an den Socket schicken. Der Message-Handler empfängt dann die Nachrichten und wandelt sie in ein vom Kommunikationssystem erwartetes Format um. Der Message-Handler sendet dann die Nachricht mit Hilfe des Kommunikationssystems an den ConTract-Manager. Der Message-Handler ist bisher noch nicht implementiert, sollte aber die oben beschriebene Funktionalität aufweisen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Die Arbeit hat gezeigt, daß es durchaus sinnvoll und durchführbar ist, das Skript eines ConTracts graphisch einzugeben. Bei den vielen Möglichkeiten, die man hat, um ein graphische Benutzeroberfläche zu implementieren, fällt es schwer, sich auf das wesentliche zu konzentrieren. Außerdem ist es schwierig die Akzeptanz der Benutzer vorauszusehen und die Erweiterbarkeit der Oberfläche nicht auszuschließen. Erst wenn viele und dem System fremde Benutzer mit der Oberfläche arbeiten, stellt sich heraus, wo Erweiterungen wünschenswert sind oder die Benutzerführung geändert werden sollte.

6.2 Erweiterungen

In den folgenden Abschnitten werden Verbesserungsmöglichkeiten und Erweiterungen der Sprache und der Oberflächen genannt. Die Anzahl der möglichen Erweiterungen zeigt, daß es noch viel zu tun gibt.

6.2.1 Erweiterung der Sprache

- Die Wiedereinführung der Parforeach-Anweisung.
- Die Einführung einer Verfeinerung mit Parameterschnittstelle brächte den Vorteil, daß man ähnlich wie beim aus dem Software-Engineering bekannten SADT[1]-Verfahren bei der Programmierung des Skriptes vorgehen könnte. Dadurch würde der Datenfluß übersichtlicher und die Möglichkeit zur Top-Down-Programmierung verbessert. Dies könnte in EBNF wie folgt formuliert werden:

  ```
  Procedure = 'PROCEDURE' '(' ParameterInterface ')'
                ContextVariableDeclaration
              'BEGIN'
                Sequence
              'END'.
  ```

  Das Parameterinterface würde dem Prozeduraufruf in einer herkömmlichen Programmiersprache entsprechen. Die Prozeduraufrufe dürfen dann allerdings nicht rekursiv erfolgen, da die Kontextvariablen persistent sein müssen. Die Kontextvariablen innerhalb der Prozedur können nur lokal, d.h. innerhalb der Verfeinerung, angesprochen werden, müssen aber während der gesamten Laufzeit des ConTracts vorhanden sein.
- Die Einführung benutzerdefinierter Typen nach dem ADT-Prinzip ist äußerst wichtig, da es nicht sinnvoll ist, zwischen den Steps nur einfache Datentypen wie Integer-Zahlen auszutauschen. Vielmehr wäre z.B. wichtig Dateireferenzen oder ähnliches übergeben zu können.

1. SADT: Structured Analysis and Design Techniques

6.2.2 Erweiterung des ConTract-Development-Tools

- Das bisher verfügbare Programm zeigt noch zu wenig Informationen über den ConTract an. Eine Anzeige wann und von wem ConTract angelegt bzw. gelockt wurde und das Datum der letzten Änderung sollten noch zusätzlich angezeigt werden.
- Die Vergabe von Benutzerrechten (read, write, execute) wie im Unix-Dateisystem wäre angebracht, da dies Vorteile im Mehrbenutzerbetrieb in bezug auf die Datensicherheit bringen würde.
- Die Einteilung der ConTracts in Zugriffsgruppen. Damit könnte jedem Benutzer einer Gruppe nur bestimmte Teile des Gesamtsystems sichtbar gemacht werden. Dies wäre für eine Gruppenarbeit unbedingt erforderlich.

6.2.3 Erweiterung des ConTract-Editors

- Die Integration des Konfigurationstools in den Editor. Die Bindung des Codes an die Steps ist ein wichtiger Bestandteil der ConTract-Programmierung.
- Wenn der Compiler implementiert wird, muß dem Compiler eine Möglichkeit verschafft werden, daß er seine Warnungen und Fehlermeldungen anzeigen kann. Da das Skript nicht als textuelle Eingabe vorliegt, können auch keine „normalen" Fehlermeldungen generiert werden. Z.B. müßte die Oberfläche das fehlerhafte Konstrukt hervorheben und die zugehörige Fehlermeldung anzeigen, um dem Benutzer die Möglichkeit zur Korrektur des Fehlers anbieten zu können.
- Ein mehrfaches selektieren durch ein „Gummiband" (ein durch den Mauszeiger aufgespannter Bereich auf der Oberfläche) würde das Selektieren von Blöcken erleichtern.
- Das Verschieben und Kopieren von Objekten innerhalb eines ConTract-Editors oder zwischen verschiedenen ConTract-Editoren nach dem Motif-Modell für Drag & Drop würde eine gewisse Arbeitserleichterung mit sich bringen.
- Die syntaktische Prüfung der Invarianten und Prädikate muß unbedingt noch Bestandteil der Oberfläche werden. Die Prädikate können zum Teil auch semantisch geprüft werden. Damit wird dem Compiler Arbeit abgenommen und der Programmierer erhält ein früheres Feedback.
- Die Einschränkungen von Verzweigen nach Konfliktbehandlung sollte schon bei der Eingabe überprüft werden.
- Die Darstellung des Datenflusses im Editor würde der Fehlervermeidung dienen.
- Das Prüfen des Datenflusses auf nicht initialisierte oder nicht benutzte Variablen oder Indeterminismus bei Zugriffen auf die gleiche Variable aus verschiedenen Parallelzweigen.
- Die Implementierung einer Undo-Operation ist sehr wichtig, um Falscheingaben rückgängig machen zu können.

6.2.4 Erweiterung des ConTract-Animators

- Auf die Darstellung von großen Parallel-Konstrukten muß besonders Rücksicht genommen werden. Der Benutzer verliert sonst möglicherweise die Übersicht über den aktuellen Zustand des Ablaufs, weil die Parallelzweige nicht auf einmal auf den Monitor passen.

- Ein Debug-Modus für den ConTract-Animator wäre sinnvoll, um die Prüfung der Skriptprogrammierung zu vereinfachen. Z.B. könnten Breakpoints eingeführt werden oder daß man sich die Werte der Kontextvariablen anzeigen und gegebenfalls auch ändern kann.
- Eine Anzeige, die darüber Auskunft gibt, auf welchem Rechner ein Step gerade ausgeführt wird oder wurde, wäre auch nützlich, um z.B. Rückschlüsse auf das Laufzeitverhalten machen zu können.
- Der Zustand der ConTract-Manager und der Step-Server, die im Netz zur Verfügung stehen, könnten angezeigt werden.
- Für den praktischen Gebrauch ist auch eine Simulation der Ausführung erforderlich. Diese Simulation könnte Daten (Laufzeitverhalten) ermitteln, die zur Optimierung der ConTract-Abarbeitung benutzt werden könnten. Damit wird automatisch auch der Geschäftsprozeß, der durch den ConTract modelliert wird, verbessert. Dies kann zu Kosteneinsparungen im betrieblichen Ablauf führen.

A Datenbankzugriff

Für das Speichern der Daten wird eine Datenbank eingesetzt. Eine Übersicht über die verwendeten Datenbankrelationen wird im nächsten Abschnitt gegeben, dabei wurde die Notation des Hilfsprogramms ingmenu der Ingres-Datenbank verwendet.

A.1 Übersicht über Datenbankrelationen

TABELLE 1. Relation ConTracts

Attribut	Datentyp	Schlüssel	Nullwert	Standardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
name	varchar(60)		yes	n/a	Name des ConTracts
version	i4		yes	n/a	Version des ConTracts
status	i1		yes	n/a	Status des ConTracts
bind_cid	i4		yes	n/a	Verweis auf zugehörigen ConTract

TABELLE 2. Relation Object_Instances

Attribut	Datentyp	Schlüssel	Nullwert	Standardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
oid	i4	2	no	no	Objekt-Identifikation
type	i1		yes	n/a	Datentyp des Objekts
next_obj	i4		yes	n/a	nächstes Objekt in Liste
first_obj	i4		yes	n/a	erstes Kindobjekt (für Liste)
x	i4		yes	n/a	x-Koordinate
y	i4		yes	n/a	y-Koordinate
w	i4		yes	n/a	Breite
h	i4		yes	n/a	Höhe
shortcut	varchar(10)		yes	n/a	Abkürzung
predicate_btn	i4		yes	n/a	Verweis auf Prädikat-Button
pid	i4		yes	n/a	Verweis auf Prädikat
collapsed	i1		yes	n/a	kollabiert/expandiert-Zustand
abort_conflict_res	i4		yes	n/a	Verweis auf Konfliktauflösung
on_abort_goto	i4		yes	n/a	Verweis auf Objekt, das bei einem Konflikt angesprungen wird
siid	i4		yes	n/a	Verweis auf Stepinstanz
compensation_siid	i4		yes	n/a	Verweis auf Stepinstanz des Compensationssteps

TABELLE 3. Relation Step_Instances

Attribut	Datentyp	Schlüssel	Nullwert	Standardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
siid	i4	2	no	no	Stepinstanzen-Identifikation

TABELLE 3. Relation Step_Instances

Attribut	Datentyp	Schlüs-sel	Null-wert	Stan-dardwert	Erläuterungen
sid	i4		yes	n/a	Verweis auf Stepdefinition
entry_inv	varchar(256)		yes	n/a	Eintrittsinvariante
exit_inv	varchar(256)		yes	n/a	Austrittsinvariante
entry_inv_conflict_res	i4		yes	n/a	Verweis auf Konfliktauflösung bei einem Eintrittsinvarianten-Fehler
on_entry_conflict_goto	i4		yes	n/a	Verweis auf Objekt, das nach abarbeiten der Konfliktauflösung angesprungen wird
abort_conflict_res	i4		yes	n/a	Verweis auf Konfliktauflösung
on_abort_goto	i4		yes	n/a	Verweis auf Objekt, das bei einem Konflikt angesprungen wird

TABELLE 4. Relation Step_Inst_Par_Context

Attribut	Datentyp	Schlüs-sel	Null-wert	Stan-dardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
siid	i4	2	no	no	Stepinstanzen-Identifikation
position	i2	3	no	no	Position des Parameters
ctxt_var	i4		yes	n/a	Verweis auf Contextvariable

TABELLE 5. Relation Context_Variables

Attribut	Datentyp	Schlüs-sel	Null-wert	Stan-dardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
ctxt_var	i4	2	no	no	Contextvariablen-Identifikation
name	varchar(60)	3	no	no	Name der Contextvariablen
type	varchar(60)		yes	n/a	Typ der Contextvariablen

TABELLE 6. Relation Step_Definitions

Attribut	Datentyp	Schlüs-sel	Null-wert	Stan-dardwert	Erläuterungern
sid	i4	1	no	no	Step-Identifikation
name	varchar(60)	2	no	no	Name des Steps
rm_class	varchar(60)	3	no	no	Klasse des Steps

TABELLE 7. Relation Step_Parameters

Attribut	Datentyp	Schlüs-sel	Null-wert	Standard-wert	Erläuterungen
sid	i4	1	no	no	Step-Identifikation
position	i2	2	no	no	Position des Parameters
type	var-char(60)		yes	n/a	Typ des Parameters
call_mode	i1		yes	n/a	Übergabemodus

TABELLE 8. Relation Predicates

Attribut	Datentyp	Schlüssel	Nullwert	Standardwert	Erläuterungen
cid	i4	1	no	no	ConTract-Identifikation
pid	i4	2	no	no	Prädikat-Identifikation
predicate	varchar(255)		yes	n/a	Inhalt des Prädikates als String
type	i1		yes	n/a	Typ des Prädikats

TABELLE 9. Relation ConTract_dynamic

Attribut	Datentyp	Schlüssel	Nullwert	Standardwert	Erläuterungen
ciid	i4	1	no	no	ConTract-Instanz-Identifikation
cid	i4		yes	n/a	ConTract-Identifikation
status	i4		yes	n/a	Status der ConTract-Instanz
compile_time	date		yes	n/a	Datum und Uhrzeit, zu der der ConTract übersetzt wurde
start_time	date		yes	n/a	Datum und Uhrzeit, zu der der ConTract gestartet wurde
suspend_time	date		yes	n/a	Datum und Uhrzeit, zu der der ConTract suspendiert wurde

A.2 Schnittstelle zur Datenbank

A.2.1 Allgemeines

Die Schnittstelle zur Datenbank ist weitgehend so gehalten, daß im Prinzip die Daten auch in einem Filesystem ablegt werden könnten, ohne daß sich die Schnittstelle ändert.

A.2.1.1 Dateien

dbinterface.h: enthält die Definitionen für den Datenbankzugriff

dbinterface.sc: Embedded SQL-Quelltext für den Datenbankzugriff

A.2.1.2 Datenstrukturen

Der Fehlercode wird in der Form eines Auszählungstyps (DBError) zurückgegeben.

```
typedef enum { DBE_NOERR,
               DBE_NOTFOUND,
               DBE_DENIED,
               DBE_ERR,
               DBE_NOTIMPL,
               DBE_CTEMPTY
             } DBError;
```

TABELLE 10. Datenbank-Fehlercode

Fehlercode	Erläuterung
DBE_NOERR	kein Fehler aufgetreten
DBE_NOTFOUND	gesuchtes Datum nicht gefunden

TABELLE 10. Datenbank-Fehlercode

Fehlercode	Erläuterung
DBE_DENIED	Zugriff verweigert (logisch, nicht von der Datenbank aus)
DBE_ERR	Fehler aufgetreten
DBE_NOTIMPL	Compilieren eines ConTracts fehlgeschlagen
DBE_CTEMPTY	ConTract war leer, als er dem Compiler übergeben wurde

A.2.1.3 Portierbarkeit

Die Datei dbinterface.sc ist auf die Embedded SQL Datenbanksprache von Ingres entworfen worden. Es sollte dennoch kein Problem sein es auf ein anderes Datenbanksystem zu portieren oder gar durch ein Filesystem zu implementieren, da keine speziellen Datenbankbefehle verwendet werden. Zu beachten ist, daß in der Definitionsdatei dbinterface.h der Datenbankname bei der Define-Anweisung für DATABASE_NAME angegeben werden muß.

A.2.1.4 Zugriff auf Datenbank

Methode - openDatabase

Definition:

```
DBError openDatabase ( void );
```

Beschreibung:

Öffnet den Zugang zur Datenbank. Der Datenbank-Name (DATABASE_NAME) ist in einer Define-Anweisung in der Definitionsdatei dbinterface.h zu finden. Das Gegenstück zu dieser Funktion ist closeDatabase.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten (z.B. Datenbank nicht vorhanden)

Methode - closeDatabase

Definition:

```
DBError closeDatabase ( void );
```

Beschreibung:

Schließt den Zugang zur Datenbank. Der Datenbankzugriff muß zuvor mit openDatabase eingeleitet worden sein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - commitWork

Definition:

```
DBError commitWork   ( void );
```

Beschreibung:

Macht einen Commit auf alle Aktionen, die in der Datenbanksession ausgeführt wurden.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - rollbackWork

Definition:

```
DBError rollbackWork ( void );
```

Beschreibung:

Macht alle Aktionen, die auf der Datenbank seit dem letzten commitWork ausgeführt wurden wieder rückgängig.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

A.2.2 Verwaltung von ConTracts

A.2.2.1 Datenstrukturen

```
typedef enum { CTS_ORIGINAL    = 0,
               CTS_COPY        = 1,
               CTS_LOCKED      = 2 } DBConTractStatus;

typedef struct DBCONTRACT
{
  long cid;
  char name[NAME_LENGTH];
  long version;
  DBConTractStatus status;
  long bind_cid;
} DBConTract;
```

Die Struktur DBConTract beschreibt die Eigenschaften eines ConTracts. Dazu gehört die eindeutige Identifikation cid. Die für den Benutzer sichtbare Darstellung setzt sich aus dem Namen name, der Version version und dem Status status zusammen. Das Element bind_cid ist nur zum internen Gebrauch gedacht. Der Status läßt sich wie im Aufzählungstyp DBConTractStatus deklariert aufschlüsseln.

A.2.2.2 Funktionen

Methode - openConTract

Definition:

```
DBError openConTract ( IN  char name[NAME_LENGTH],
                       IN  long version,
                       OUT DBConTract *dbConTract );
```

Beschreibung:

Öffnet den ConTract mit Name name und Version version zum Lesen und Schreiben. Dabei wird der locked-Status gesetzt. Es wird eine Arbeitskopie erstellt, auf der dann gearbeitet wird, bis eine closeConTract-Variante aufgerufen wird. Das Element bind_cid bekommt die Identifikation des Original-ConTracts zugewiesen. Die Arbeitskopie ist bis auf die ConTract-Identifikation identisch.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; es wird ein Commit durchgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback durchgeführt
DBE_DENIED	ConTract hat den locked-Status, d.h. es könnte sein, daß jemand anderes an dem ConTract arbeitet; es wird ein Rollback durchgeführt

Methode - openConTractRead

Definition:

```
DBError openConTractRead ( IN  char name[NAME_LENGTH],
                           IN  long version,
                            OUT DBConTract *dbConTract );
```

Beschreibung:

Öffnet den ConTract mit Name name und Version version zum Lesen. Dabei ändert sich der Status nicht. Das Gegenstück zu dieser Funktion ist closeConTractRead.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; es wird ein Commit durchgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback durchgeführt
DBE_DENIED	ConTract hat den locked-Status, d.h. es könnte sein, daß jemand anderes an dem ConTract arbeitet; es wird ein Rollback durchgeführt

Methode - openLockedConTract

Definition:

```
DBError openLockedConTract( IN char name[NAME_LENGTH],
                           IN  long version,
                            OUT DBConTract *dbConTract );
```

Beschreibung:

Öffnet einen ConTract, der den Status locked besitzt. Diese Funktion ist dann nötig, wenn das Programm oder der Rechner während der Bearbeitung eines ConTracts ausfällt. (Nur Anwenden, wenn unbedingt erforderlich!).

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt

Methode - newConTract

Definition:

```
DBError newConTract ( IN char name[NAME_LENGTH],
                      OUT DBConTract *dbConTract );
```

Beschreibung:

Legt einen neuen ConTract mit Name name an. Falls ein ConTract mit diesem Namen bereits existiert, wird eine neue Version dieses ConTracts erstellt.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; es wurde ein neuer ConTract angelegt; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt

Methode - closeConTract

Definition:

```
DBError closeConTract ( IN DBConTract *dbConTract );
```

Beschreibung:

Die Änderungen am ConTract werden mit dieser Funktion übernommen. Falls ein ConTract gerade mit newConTract angelegt wurde, wird dieser gespeichert. Andernfalls existiert eine Arbeitskopie. Die Arbeitskopie ersetzt den alten ConTract. Der lokked-Status wird gelöscht.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - closeConTractRead

Definition:

```
DBError closeConTractRead(IN DBConTract *dbConTract );
```

Beschreibung:

Schließt den Lesezugriff auf einen ConTract ab.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - closeConTractAsNewName

Definition:

```
DBError closeConTractAsNewName (
                          IN DBConTract *dbConTract,
                          IN char name[NAME_LENGTH] );
```

Beschreibung:

Die Änderungen am ConTract werden unter einem neuen Namen abgespeichert. Die Arbeitskopie erhält den neuen Namen. Der zu editierende ConTract wurde nicht geändert. Existiert bereits ein ConTract unter dem neuen Namen, wird der zu speichernde ConTract als neue Version abgespeichert.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - closeConTractAsNewVersion

Definition:

```
DBError closeConTractAsNewVersion ( IN DBConTract
*dbConTract );
```

Beschreibung:

Die Änderungen am ConTract werden unter einer neuen Version abgespeichert. Der zu editierende ConTract wurde nicht geändert.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - closeConTractDiscardChange

Definition:

```
DBError closeConTractDiscardChanges ( IN DBConTract
*conTract );
```

Beschreibung:

Die Änderungen am ConTract werden nicht gespeichert. Der zu editierende ConTract wurde nicht geändert. Die Arbeitskopie wird gelöscht.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - countConTracts

Definition:

```
DBError countConTracts ( OUT long *i );
```

Beschreibung:

Zählt die ConTracts in der Tabelle ConTracts.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter i enthält die Anzahl der ConTracts
DBE_ERR	Fehler aufgetreten

Methode - readFirstConTract

Definition:

```
DBError readFirstConTract ( OUT DBConTract *conTract );
```

Beschreibung:

Liest den ersten ConTract aus der Tabelle ConTracts.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter conTract enthält die Informationen über den ConTract
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Relation enthält keinen ConTract

Methode - readNextConTract

Definition:

```
DBError readNextConTract  ( INOUT DBConTract *conTract );
```

Beschreibung:

Liest den nächsten ConTract aus der Tabelle ConTracts.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter conTract enthält die Informationen über den ConTract
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	der letzte ConTract wurde bereits gelesen

Methode - readConTract

Definition:

```
DBError readConTract ( IN  char name[NAME_LENGTH],
                       IN  long version,
                       OUT DBConTract *conTract );
```

Beschreibung:

Liest Information über den ConTract mit Namen name und Version version ein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter conTract enthält die Informationen über den ConTract
DBE_ERR	Fehler aufgetreten

Methode - removeConTract

Definition:

```
DBError removeConTract ( IN DBConTract *dbConTract );
```

Beschreibung:

Löscht einen ConTract.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten
DBE_DENIED	ConTract hat den locked-Status

Methode - cmpConTracts

Definition:

```
DBError cmpConTracts ( IN DBConTract *c1, IN DBConTract
*c2 );
```

Beschreibung:

Vergleicht 2 Datenstrukturen vom Typ DBConTract.

Rückgabewerte:

DBE_NOERR	die Strukturen sind gleich
DBE_ERR	die Strukturen unterscheiden sich

Methode - deleteAllOfConTract

Definition:

```
DBError deleteAllOfConTract ( IN long cid );
```

Beschreibung:

Löscht alle Information, die in einem ConTract gespeichert sind. Mit dieser Funktion wird das Schreiben eingeleitet. Es wird kein Commit ausgeführt, d.h. die Daten sind noch nicht endgültig verloren.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - commitConTractEdits

Definition:

```
DBError commitConTractEdit ( IN long cid );
```

Beschreibung:

Führt einen Commit auf alle Änderungen in einem ConTract aus. Bis jetzt wird ein Commit auf die gesamte Datenbank durchgeführt. Führt zu Problemen bei konkurrierenden Schreib-/Lesezugriffen.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - rollbackConTractEdits

Definition:

```
DBError rollbackConTractEdits ( IN long cid );
```

Beschreibung:

Führt einen Rollback auf alle Änderungen in einem ConTract aus. Bis jetzt wird ein Rollback auf die gesamte Datenbank durchgeführt. Führt zu Problemen bei konkurrierenden Schreib-/Lesezugriffen.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

A.2.3 Zugriff auf ConTract-Instanzen

A.2.3.1 Datenstrukturen

```
typedef enum { CTIS_INITIAL   = 0,
               CTIS_RUNNING   = 1,
               CTIS_SUSPENDED = 2,
               CTIS_FINISHED  = 3
             } DBConTractInstStatus;

typedef struct
{
  long ciid;
  long cid;
  DBConTractInstStatus status;
  char compile_time[TIME_LENGTH];
  char start_time  [TIME_LENGTH];
  char suspend_time[TIME_LENGTH];
} DBConTractInst;
```

Die Struktur DBConTractInst beschreibt die Eigenschaften einer ConTract-Instanz.

A.2.3.2 Funktionen

Methode - countConTractInsts

Definition:

```
DBError countConTractInsts ( IN long cid,
                             OUT long *instances );
```

Beschreibung:

Zählt die ConTract-Instanzen in der Relation ConTract_Instances.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter instances enthält die Anzahl der ConTract-Instanzen
DBE_ERR	Fehler aufgetreten

Methode - readConTractInst

Definition:

```
DBError readConTractInst ( IN long cid,
                           IN long ciid,
                          OUT DBConTractInst *instance );
```

Beschreibung:

Liest die Informationen über eine ConTract-Instanz.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter instance enthält die Informationen
DBE_ERR	Fehler aufgetreten

Methode - readFirstConTractInst

Definition:

```
DBError readFirstConTractInst ( IN long cid,
                          OUT DBConTractInst *instance );
```

Beschreibung:

Liest erste ConTract-Instanz aus der Relation ConTract_Instances aus.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - readNextConTractInst

Definition:

```
DBError readNextConTractInst  ( IN long cid,
                                OUT DBConTractInst *instance );
```

Beschreibung:

Liest nächste ConTract-Instanz.

Rückgabewerte:

DBE_ERR	Fehler aufgetreten
DBE_NOERR	kein Fehler aufgetreten

A.2.4 Zugriff auf Steps

A.2.4.1 Datenstrukturen

```
typedef struct DBSTEP
{
  char name[NAME_LENGTH];
  char rm_class[NAME_LENGTH];
  long sid;
} DBStep;
```

Die Datenstruktur DBStep beschreibt die Eigenschaften einer Step-Definition. Der eindeutigen Identifizierung einer Step-Definition dient das Element sid. Die Aufschlüsselung ergibt den Name name und die Ressourcemanagerklasse rm_class.

A.2.4.2 Funktionen

Methode - readFirstStep

Definition:

```
DBError readFirstStep ( OUT DBStep *step );
```

Beschreibung:

Liest erste Step-Definition aus der Relation Step_Definitions aus.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter step enthält die Informationen über die Step-Definition
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Relation enthält überhaupt keine Step-Definition

Methode - readNextStep

Definition:

```
DBError readNextStep  ( INOUT DBStep *step );
```

Beschreibung:

Liest die nächste Step-Definition aus der Relation Step_Definitions aus. Zuvor muß ein readFirstStep erfolgt sein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter step enthält die Informationen über die Step-Definition
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	die letzte Step-Definition wurde bereits gelesen

Methode - readStep

Definition:

```
DBError readStep      ( IN  long sid,
                        OUT DBStep *step );
```

Beschreibung:

Liest die Informationen über einen Step mit Id sid.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter step enthält die Informationen über die Step-Definition
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Step-Definition nicht vorhanden

Methode - removeStep

Definition:

```
DBError removeStep    ( IN long sid );
```

Beschreibung:

Löscht die Step-Definition mit Id sid (Commit wird durchgeführt).

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; die Step-Definition ist gelöscht
DBE_ERR	Fehler aufgetreten

Methode - newStep

Definition:

```
DBError newStep       ( INOUT DBStep *step  );
```

Beschreibung:

Legt eine neue Step-Definition an. Es werden die Werte aus dem Parameter step benutzt bis auf die Step-Id, die wird von dieser Funktion gesetzt. (Commit wird durchgeführt)

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; das Element sid des Parameters step enthält nun die eindeutige Identifikation
DBE_ERR	Fehler aufgetreten

Methode - updateStep

Definition:

```
DBError updateStep    ( IN    DBStep *step );
```

Beschreibung:

Ändert den Eintrag einer Step-Definition ab. Es werden die Werte aus dem Parameter step benutzt. (Commit wird durchgeführt)

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - readParameter

Definition:

```
DBError readParameter ( IN  long sid,
                        IN  long pos,
                        OUT DBParameter *parameter);
```

Beschreibung:

Liest den Parameter an der Position pos einer Step-Definition.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter parameter enthält die Informationen über den Parameter; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt
DBE_NOTFOUND	kein Parameter an dieser Position vorhanden; es wird ein Rollback ausgeführt

Methode - removeParameter

Definition:

```
DBError removeParameter ( IN long sid,
                          IN long pos );
```

Beschreibung:

Löscht den Parameter einer Step-Definition mit der Position pos.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter ist gelöscht; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt
DBE_NOTFOUND	der angegebene Parameter konnte nicht gefunden werden; es wird ein Rollback ausgeführt

Methode - newParameter

Definition:

```
DBError newParameter ( IN long sid,
                       IN DBParameter *parameter );
```

Beschreibung:

Legt einen neuen Parameter an. Die Position wird dem Parameter parameter entnommen.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter wurde angelegt; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt

Methode - updateParameter

Definition:

```
DBError updateParameter ( IN long sid,
                          IN DBParameter *parameter );
```

Beschreibung:

Ändert die Daten eines Parameter einer Step-Definition ab.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter wurde abgeändert; es wird ein Commit ausgeführt
DBE_ERR	Fehler aufgetreten; es wird ein Rollback ausgeführt
DBE_NOTFOUND	der angegebene Parameter konnte nicht gefunden werden; es wird ein Rollback ausgeführt

A.2.5 Zugriff auf Objekte

A.2.5.1 Datenstrukturen

```
typedef enum { OBJ_CONTRACT, OBJ_CONFLICTRES,
       OBJ_STEPBUTTON, OBJ_HORLINE, OBJ_PREDICATEBTN,
       OBJ_IFBRANCH, OBJ_SWITCHBRANCH, OBJ_PARALLELBRANCH,
```

```
       OBJ_FORLOOP, OBJ_WHILELOOP, OBJ_REPEATLOOP,
       OBJ_BLOCK, OBJ_TRANSACTION,
       OBJ_SEQUENCE, OBJ_ALTERNATIVE
     } DBObjectType;

typedef struct DBOBJECT
{
  long next;
  long first;
  long x, y, w, h;
  long oid;
  DBObjectType type;
  union
    {
      struct
        {
          char shortcut[SHORTCUT_LENGTH];
          long siid;
          long compensation_siid;
        } stepbutton;
      struct
        {
          char type;
          char predicate [ INV_LENGTH ];
        } predicateBtn;
      struct
        {
          long predicateButton;
        } ifBranch;
      struct
        {
          long predicateButton;
        } forLoop;
      struct
        {
          long predicateButton;
        } whileLoop;
      struct
        {
          long predicateButton;
        } repeatLoop;
      struct
        {
          long collapsed;
          char shortcut[SHORTCUT_LENGTH];
        } block;
      struct
        {
          long collapsed;
          char shortcut[SHORTCUT_LENGTH];
```

```
          long abortConflictRes;
          long onAbortGoto;
        } transaction;
      struct
        {
          long predicateButton;
        } alternative;
    } contents;
} DBObject;
```

Die Struktur DBObject enthält die Eigenschaften eines graphischen Objekts. Die Felder der Union contents sind je nach Objekttyp type belegt. Da bei normalen Datenbanken keine Unions unterstützt werden, werden die Elemente der Union je nach Objekttyp auf Attribute der Relation Object_Instances abgebildet. Dabei wurde die Zuordnung wie in der nächsten Tabelle zu sehen ist durchgeführt.

TABELLE 11. Benutzung der opt. Teile der Relation Object_Instances

Objekt-Typ	**shortcut**	**predicate_btn**	**predicate_id**	**collapsed**	**conflict_res**	**on_conflict_goto**	**siid**	**compensation_siid**
OBJ_CONTRACT								
OBJ_CONFLICTRESOLUTION								
OBJ_STEPBUTTON	X						X	X
OBJ_HORLINE								
OBJ_PREDICATEBTN			X					
OBJ_IFBRANCH		X						
OBJ_SWITCHBRANCH								
OBJ_PARALLEL								
OBJ_FORLOOP		X						
OBJ_WHILELOOP		X						
OBJ_REPEATLOOP		X						
OBJ_BLOCK	X			X				
OBJ_TRANSACTION	X			X	X	X		
OBJ_SEQUENCE								
OBJ_ALTERNATIVE		X						

A.2.5.2 Funktionen

Methode - readObj

Definition:

```
DBError readObj ( IN  long cid,
                  IN  long oid,
                  OUT DBObject *obj );
```

Beschreibung:

Lese die Informationen eines Objekts mit Objekt-Id oid vom ConTract mit Id cid.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter obj enthält alle Informationen über das graphische Objekt
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Objekt konnte nicht gefunden werden

Methode - readFirstObj

Definition:

```
DBError readFirstObj ( IN  long cid,
                       OUT DBObject *obj );
```

Beschreibung:

Liest die Information über das graphische Objekt mit Objekttyp OBJ_CONTRACT. Mit dieser Funktion wird das Lesen des Objektbaums eingeleitet. D.h. es darf nur ein Objekt mit dem Typ OBJ_CONTRACT in einem ConTract vorhanden sein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter obj enthält die Informationen über das Objekt.
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Objekt konnte nicht gefunden werden

Methode - writeObj

Definition:

```
DBError writeObj ( IN    long cid,
                   INOUT DBObject *obj );
```

Beschreibung:

Schreibt die Informationen über ein graphisches Objekt. Von dieser Funktion wird eine eindeutige Objekt-Id vergeben.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; Daten wurden geschrieben; der Parameter obj enthält die eindeutige Objekt-Id
DBE_ERR	Fehler aufgetreten

Methode - updateObj

Definition:

```
DBError updateObj ( IN    long cid,
                    IN    long oid,
                    IN    DBObject *obj );
```

Beschreibung:

Ändert die Information des Objekts mit Objekt-Id oid.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; die Daten wurden geändert
DBE_ERR	Fehler aufgetreten

A.2.6 Zugriff auf Kontextvariablen

A.2.6.1 Datenstrukturen

```
typedef struct DBCONTEXTVARIABLE
{
  long cvid;
  char name[NAME_LENGTH];
  char type[NAME_LENGTH];
} DBContextVariable;
```

Die Struktur DBContextVariable beschreibt die Eigenschaften einer Kontextvariable. Mit der Kontextvariable-Id cvid ist die Variable eindeutig identifiziert. Der Name name ist der Name der Variable und type der Typ der Variablen.

A.2.6.2 Funktionen

Methode - readFirstContextVariable

Definition:

```
DBError readFirstContextVariable ( IN long cid,
                     OUT DBContextVariable *variable );
```

Beschreibung:

Liest die erste Kontextvariable eines ConTracts.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter variable enthält die Informationen über der ConTract
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	für diesen ConTract sind keine Kontextvariablen deklariert

Methode - readNextContextVariable

Definition:

```
DBError readNextContextVariable  ( IN long cid,
                   INOUT DBContextVariable *variable );
```

Beschreibung:

Lese nächste Kontextvariable eines ConTracts. Zuvor muß readFirstContextVariable aufgerufen worden sein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter variable enthält die Informationen über der ConTract
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	die letzte Kontextvariable wurde bereits gelesen

Methode - newContextVariable

Definition:

```
DBError newContextVariable       ( IN    long cid,
                    INOUT DBContextVariable *variable);
```

Beschreibung:

Lege neue Kontextvariable an. Dabei wird eine neue Kontextvariablen-Id vergeben.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; die Daten wurden geschrieben; der Parameter variable enthält die neue Kontextvariablen-Id
DBE_ERR	Fehler aufgetreten

A.2.7 Zugriff auf Step-Instanzen

A.2.7.1 Datenstrukturen

```
typedef struct DBSTEPINST
{
  long siid;
  long sid;
  char entryInv[INV_LENGTH];
  char exitInv [INV_LENGTH];

  long entryInvConflictRes;
  long onEntryInvConflictGoto;

  long abortConflictRes;
  long onAbortGoto;
} DBStepInst;
```

Die Struktur DBStepInst beschreibt die Eigenschaften einer Step-Instanz. Das Element siid ist eine eindeutige Identifikation für eine Step-Instanz innerhalb eines ConTracts. Das Element sid ist ein Verweis auf die Step-Definition von der die Step-Instanz eine Instanz bildet. Die Felder entryInv und exitInv geben die Eintritts- und die Austrittsinvariante an. Das Feld entryInvConflictRes ist ein Verweis auf ein graphisches Objekt vom Typ OBJ_CONFLICTRES und das Feld onEntryInvConflictGoto ist ein Verweis auf ein beliebiges graphisches Objekt, das im Falle des Fehlschlagens der Eintrittsinvariante angesprungen werden soll, nachdem die Konfliktbehandlung durchlaufen wurde. Das Feld abortConflictRes ist ein Verweis auf ein graphisches Objekt vom Typ OBJ_CONFLICTRES und das Feld onAbortGoto ist ein Verweis auf ein beliebiges graphisches Objekt, das im Falle des Fehlschlagens der Stepausführung angesprungen werden soll, nachdem die Konfliktbehandlung durchlaufen wurde.

A.2.7.2 Funktionen

Methode - readStepInst

Definition:

```
DBError readStepInst ( IN  long cid,
                       IN  long siid,
                       OUT DBStepInst *stepInst );
```

Beschreibung:

Liest eine Step-Instanz mit Id siid eines ConTracts ein.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; die Informationen stehen im Parameter stepInst
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	die Step-Instanz konnte nicht gefunden werden

Methode - writeStepInst

Definition:

```
DBError writeStepInst ( IN    long cid,
                        INOUT DBStepInst *stepInst );
```

Beschreibung:

Schreibt die Daten einer Step-Instanz. Die Funktion vergibt eine eindeutig Identifikation.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; die Daten wurden geschrieben; der Parameter stepInst enthält die neu vergebene Identifikation
DBE_ERR	Fehler aufgetreten

Methode - updateStepInst

Definition:

```
DBError updateStepInst ( IN long cid,
                         IN long siid,
                         IN DBStepInst *stepInst );
```

Beschreibung:

Ändert die Daten der Step-Instanz mit Id siid vom ConTract mit Id cid.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

Methode - readParameterContext

Definition:

```
DBError readParameterContext ( IN  long cid,
                               IN  long siid,
                               IN  long pos,
                              OUT DBParCtxt *parCtxt );
```

Beschreibung:

Liest die Informationen, welche Kontextvariable einem Parameter einer Step-Instanz übergeben wird.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten; der Parameter parCtxt enthält die Informationen
DBE_ERR	Fehler aufgetreten
DBE_NOTFOUND	Kontext zum Parameter existiert nicht

Methode - writeParameterContext

Definition:

```
DBError writeParameterContext ( IN long cid,
                                IN long siid,
                                IN DBParCtxt *parCtxt );
```

Beschreibung:

Schreibt die Informationen, welche Kontextvariable einem Parameter einer Step-Instanz übergeben wird.

Rückgabewerte:

DBE_NOERR	kein Fehler aufgetreten
DBE_ERR	Fehler aufgetreten

B Beschreibung wichtiger Klassen

B.1 Die Klasse Rectangle

B.1.1 Instanzenvariablen

TABELLE 12. Instanzenvariablen von Klasse Rectangle

Ablei-tungsregel	Typ	Name	Erläuterung
protected	Coord	_x	x-Koordinate des Rechtecks
protected	Coord	_y	y-Koordinate des Rechtecks
protected	Coord	_w	Breite des Rechtecks
protected	Coord	_h	Höhe des Rechtecks

B.1.2 Konstruktoren/Destruktoren

Definition:

```
public: Rectangle ( Coord x=0, Coord y=0,
                    Coord w=0, Coord h=0 );
```

Beschreibung:
Setzt die Werte für x- und y-Koordinate, sowie Breite und Höhe des Rechtecks. Defaultwert jeweils 0.

Definition:

```
public: ~Rectangle ( void );
```

Beschreibung:
Standarddestruktor.

B.1.3 Methoden

Methode - getX

Definition:

```
Coord     getX ( void ) const;
```

Beschreibung:
Liefert die x-Koordinate des Rechtecks.

Rückgabewert:
Die x-Koordinate des Rechtecks.

Methode - getY

Definition:

```
Coord     getY ( void ) const;
```

Beschreibung:
Liefert die y-Koordinate des Rechtecks.

Rückgabewert:
Die y-Koordinate des Rechtecks.

Methode - getW

Definition:
```
Coord     getW ( void ) const;
```
Beschreibung:
Liefert die Breite des Rechtecks.

Rückgabewert:
Die Breite des Rechtecks.

Methode - getH

Definition:
```
Coord     getH ( void ) const;
```
Beschreibung:
Liefert die Höhe des Rechtecks.

Rückgabewert:
Die Höhe des Rechtecks.

Methode - setX

Definition:
```
void      setX ( Coord x );
```
Beschreibung:
Setzt die x-Koordinate des Rechtecks.

Methode - setY

Definition:
```
void      setY ( Coord y );
```
Beschreibung:
Setzt die y-Koordinate des Rechtecks.

Methode - setW

Definition:
```
void      setW ( Coord w );
```
Beschreibung:
Setzt die Breite des Rechtecks.

Methode - setH

Definition:

```
void      setH ( Coord h );
```

Beschreibung:

Setzt die Höhe des Rechtecks.

Methode - isInMe

Definition:

```
BoolType isInMe     ( Coord x, Coord y ) const;
BoolType isInMe     ( const Rectangle& rec ) const;
```

Beschreibung:

Die erste Variante prüft, ob die Koordinaten (x, y) im Rechteck liegen. Die zweite Variante prüft, ob das Rechteck rec komplett im Rechteck liegt.

Rückgabewerte:

BOOLTRUE	Koordinaten bzw. Rechteck liegen innerhalb des Rechtecks
BOOLFALSE	Koordinaten bzw. Rechteck liegen nicht innerhalb des Rechtecks

Methode - intersects

Definition:

```
BoolType intersects ( const Rectangle& rec ) const;
```

Beschreibung:

Prüft, ob das Rechteck rec mit dem Rechteck überlappt.

Rückgabewerte:

BOOLTRUE	Es existiert eine Überlappung
BOOLFALSE	Es existiert keine Überlappung

Methode - section

Definition:

```
Rectangle section ( const Rectangle& rec ) const;
```

Beschreibung:

Prüft, ob das Rechteck rec mit dem Rechteck überlappt und liefert dann die Überlappung.

Rückgabewerte:

Rectangle(0,0,0,0)	Es existiert keine Überlappung.
sonst	Bereich, in der sich die Rechtecke überlappen.

B.1.4 Dateien

rectangle.h: Defintionsdatei enthält die Klassendefinition und inline-Methoden.

rectangle.cc: Quelltextdatei enthält die Methoden der Klasse.

B.2 Die Klasse GraphObj

Diese Klassen dient als Basisklasse für alle graphischen Objekte in einem Skript.

B.2.1 Instanzenvariablen

TABELLE 13. Instanzenvariablen der Klasse GraphObj

Ablei-tungsregel	Typ	Name	Erläuterung
private	BoolType	_selected	zeigt an, ob das Objekt selektiert ist
private	int	_referenceC ounter	zeigt an, wieviele Referenzen auf dieses Objekt verweisen
private	long	_oid	eindeutige Objekt-Identifikation
protected	GraphObj *	_parent	Verweis auf das Elternobjekt
protected	static DBObject	_obj	kann zum Schreiben und Lesen in bzw. aus der Datenbank verwendet werden
protected	Color	_fgColor	Vordergrundfarbe
protected	Color	_bgColor	Hintergrundfarbe

B.2.2 Konstruktoren/Destruktoren

Definition:

```
protected: GraphObj ( void );
```

Beschreibung:

Standardkonstruktor.

Definition:

```
protected: GraphObj ( const GraphObj& graphObj );
```

Beschreibung:

Copy-Konstruktor.

Definition:

```
protected: GraphObj (BaseObj *baseObj, GraphObj *parent,
                     Coord x, Coord y, Coord w, Coord h);
```

Beschreibung:

Für den Normalfall. Es wird ein eindeutige Objekt-Id vergeben.

Definition:

```
protected: GraphObj (BaseObj *baseObj,GraphObj *parent );
```

Beschreibung:

Für das Datenbank-Einlesen. Es wird keine eindeutige Objekt-Id vergeben, da die aus der Datenbank gelesen wird.

Definition:

```
public: virtual ~GraphObj ( void );
```

Beschreibung:

Standarddestruktor.

B.2.3 Methoden

Methode - copy

Definition:

```
public: virtual GraphObj *copy ( void )=0;
```

Beschreibung:

Legt eine Kopie des graphischen Objekts an. Ist noch nicht überall korrekt implementiert und ist noch nicht getestet.

Rückgabewert:

Zeiger auf Kopie.

Methode - draw

Definition:

```
public: virtual void draw ( BaseObjView& view )=0;
```

Beschreibung:

Objekt zeigt zeichnet sich selbst und dessen Kinder in die BaseObjView view.

Methode - recomputeSize

Definition:

```
public: virtual void recomputeSize ( BoolType redraw )=0;
```

Beschreibung:

Damit berechnet das Objekt seine eigene Größe neu und paßt die Position der Kinder gegebenenfalls an. Falls sich dabei die Größe des Objekts geändert hat, wird ein Neuberechnung der Elternobjekts ausgelöst, solange bis keine Neuberechnung mehr nötig ist. Falls der Parameter redraw den Wert BOOLTRUE besitzt, werden alle Objekte unterhalb des zuletzt neu berechneten Objekts neu gezeichnet.

Methode - search

Definition:

```
public: virtual GraphObj *search (Coord x, Coord y)=0;
```

Beschreibung:

Sucht das Objekt, das unter den Koordinaten (x, y) liegt.

Rückgabewerte:

NULL	Es liegt kein Objekt an diesen Koordinaten
sonst	Zeiger auf das gefundene Objekt

Methode - write

Definition:

```
public: long write ( long cid, long next )=0;
```

Beschreibung:

Ist für das Schreiben des Objekts in die Datenbank verantwortlich. Der Parameter cid ist die Identifikation des ConTracts für den die Daten bestimmt sind. Der Parameter next wird benutzt, falls sich das Objekt in einer Liste befindet. Dann hat next den Wert der Objekt-Id des folgenden Objekts in der Liste. Wenn das Objekt nicht Teil einer Liste ist, muß next auf 0 gesetzt sein.

Rückgabewerte:

0	Es ist ein Fehler beim Schreiben in die Datenbank aufgetreten; es wird kein Rollback durchgeführt
sonst	Objekt-Id des geschriebenen Objekts; es wird kein Commit durchgeführt

Methode - read

Definition:

```
public: virtual BoolType read ( BaseObj *baseObj,
                                GraphObj *parent,
                                long cid,
                                const DBObject& obj)=0;
```

Beschreibung:

Ist für das Lesen eines Objekts aus der Struktur obj und für das Lesen der Kinder des Objekts aus der Datenbank verantwortlich. Der Parameter baseObj gibt das Basisobjekt an, zu dem das Objekt gehören soll und der Parameter parent gibt das Elternobjekt an. Der Parameter cid gibt die ConTract-Identifikation des ConTracts an, aus dem das Objekt gelesen werden soll.

Rückgabewerte:

BOOLTRUE	kein Fehler aufgetreten
BOOLFALSE	Fehler aufgetreten

Methode - searchOid

Definition:

```
public: virtual GraphObj *searchOid ( long oid )=0;
```

Beschreibung:

Dient der Suche nach einem graphischen Objekt mit Objekt-Id oid.

Rückgabewerte:

NULL	Objekt nicht gefunden
sonst	Zeiger auf graphisches Objekt

Methode - drawExcepitons

Definition:

```
public: virtual void drawExceptions
                     ( BaseObjView& view )=0;
```

Beschreibung:

Ist für das Zeichnen der Ausnahmen (Pfeile) zuständig.

Methode - writeExceptions

Definition:

```
public: virtual BoolType writeExceptions ( long cid )=0;
```

Beschreibung:

Ist für das Schreiben der Ausnahmen verantwortlich. Dies muß in einem separaten Durchlauf erfolgen, da die Objekt-Ids sich während des Schreibens der Objekte in die Datenbank noch ändern.

Rückgabewerte:

BOOLTRUE	kein Fehler aufgetreten; es wird kein Commit durchgeführt
BOOLFALSE	Fehler aufgetreten; es wird kein Rollback ausgeführt

Methode - readExceptions

Definition:

```
public: virtual BoolType readExceptions ( long cid )=0;
```

Beschreibung:

Ist für das Lesen der Ausnahmen aus der Datenbank verantwortlich.

Rückgabewerte:

BOOLTRUE	kein Fehler aufgetreten
BOOLFALSE	Fehler aufgetreten

Methode - remove

Definition:

```
public: virtual void remove ( void );
```

Beschreibung:

Damit entfernt sich das Objekt selbst aus dem ConTract.

Methode - moveAbsolute

Definition:

```
public: BoolType moveAbsolute ( Coord x, Coord y );
```

Beschreibung:

Verschiebt das Objekt, samt Kinder an die absoluten Koordinaten (x, y).

Rückgabewerte:

BOOLTRUE	Verschieben war notwendig
BOOLFALSE	Verschieben war nicht notwendig, das Objekt befand sich bereits an der gewünschten Position

Methode - getReferenceCounter

Definition:
```
public: int       getReferenceCounter ( void ) const;
```
Beschreibung:
Gibt die Anzahl der Referenzen, die auf das Objekt verweisen zurück.

Rückgabewerte:

0	Es existiert kein Referenz auf dieses Objekt
>0	Anzahl der Referenzen auf dieses Objekt

Methode - incReferenceCounter

Definition:
```
public: void incReferenceCounter ( void )
```
Beschreibung:
Setzt den Referenzen-Zähler um eins hoch.

Methode - decReferenceCounter

Definition:
```
public: void decReferenceCounter ( void )
```
Beschreibung:
Setzt den Referenzen-Zähler um eins herunter.

Methode - setOid

Definition:
```
public: long setOid ( long oid );
```
Beschreibung:
Setzt die Objekt-Id auf den Wert des Parameters oid.

Rückgabewert:
Gibt die gesetze Objekt-Id zurück.

Methode - getOid

Definition:
```
public: long getOid ( void ) const;
```
Beschreibung:
Gibt die Objekt-Id des Objekts zurück.

Rückgabewert:
Die Objekt-Id des Objekts.

Methode - setParent

Definition:

```
public:  void setParent (GraphObj *parent);
```

Beschreibung:

Damit wird das Eltern-Objekt auf den Wert des Parameters parent gesetzt.

Methode - getParent

Definition:

```
public: GraphObj *getParent ( void ) const;
```

Beschreibung:

Damit erhält man einen Zeiger auf das Elternobjekt.

Rückgabewerte:

NULL	Das Objekt ist vom Typ GO_CONTRACT.
sonst	Zeiger auf Elternobjekt.

Methode - getBaseObj

Definition:

```
public: BaseObj *getBaseObj ( void ) const;
```

Beschreibung:

Damit erhält man einen Zeiger auf das Basisobjekt (ConTract oder ConflictResolution).

Rückgabewert:

Zeiger auf Basisobjekt.

Methode - swapColor

Definition:

```
public: inline void swapColor ( void );
```

Beschreibung:

Damit wir die Vordergrundfarbe mit der Hintergrundfarbe vertauscht.

Methode - getFgColor

Definition:

```
public: inline Color getFgColor ( void ) const;
```

Beschreibung:

Liefert die Vordergrundfarbe.

Rückgabewert:

Die Vordergrundfarbe.

Methode - getBgColor

Definition:

```
public: inline Color getBgColor ( void ) const;
```

Beschreibung:
Liefert die Hintergrundfarbe.

Rückgabewert:
Die Hintergrundfarbe.

Methode - setSelected

Definition:

```
public: void changeSelected (void);
```

Beschreibung:
Ändert den selektiert Status. Selektierte Objekte werden deselektiert und umgekehrt.

Methode - isSelected

Definition:

```
public: BoolType isSelected     (void) const;
```

Beschreibung:
Damit kann abgefragt werden, ob das Objekt selektiert ist oder nicht.

Rückgabewerte:

BOOLTRUE	Objekt ist selektiert
BOOLFALSE	Objekt ist nicht selektiert

Methode - setDimension

Definition:

```
public: BoolType setDimension ( Coord  x, Coord  y,
                                Coord  w, Coord  h );
```

Beschreibung:
Setzt die x- und y-Koordinate, sowie Breite und Höhe des Objekts.

Rückgabewerte:

BOOLTRUE	Objektausmaße oder Objektposition haben sich geändert
BOOLFALSE	Objektausmaße und Objektposition haben sich nicht geändert

Methode - getDimension

Definition:

```
public: void getDimension ( Coord& x, Coord& y,
                            Coord& w, Coord& h ) const;
```

Beschreibung:
Liefert die x- und y-Koordinate, sowie Breite und Höhe des Objekts.

Methode - redraw

Definition:

```
public: void redraw ( void );
```

Beschreibung:
Fordert die View auf, ein Neuzeichnen des Objekts anzuregen.

Methode - getConTract

Definition:

```
public: ConTract *getConTract ( void ) const;
```

Beschreibung:
Gibt einen Zeiger auf den ConTract zurück, zu dem das Objekt gehört.

Rückgabewert:
Zeiger auf ConTract.

B.2.4 Dateien

graphobj.h: Definitionsdatei enthält die Klassendefinition und inline-Methoden

graphobj.cc: Quelltextdatei enthält die Methoden der Klasse

C Das Programm cdt

Das Programm cdt (ConTract-Development-Tool) ist das programmiertechnische Resultat der Studienarbeit. Das Programm ist für den Start der ConTract-Editoren, des ConTract-Compilers und der ConTract-Animatoren zuständig.

C.1 Voraussetzungen für den Start des Programms

Das Programm cdt darf nur auf Rechnern gestartet werden, die eine Ingres Datenbank mit Namen apricots zur Verfügung stellt. Andernfalls bricht das Programm mit einer Fehlermeldung ab. Außerdem muß relativ zum Pfad des ausführbaren Programms cdt in ../bitmaps die Bitmaps, die für das Programm notwendig sind, stehen. Andernfalls werden alle Buttons mit Bitmaps nicht ordnungsgemäß dargestellt. Es sollten auch gewisse Fonts vorhanden sein. Im Programm werden Fonts vom Typ -adobe-helvetica-bold-r-normal-*-*-*-*-*-*-*-*-* und -adobe-courier-bold-r-normal-*-*-*-*-*-*-*-*-* verwendet.

C.2 Testumgebungen

Das Programm wurde auf zwei Plattformen getestet. Zum einen auf einer Silicon Graphics Indy und zum anderen auf ein Sun Sparcstation 10. In den nachfolgenden Abschnitten sind die Testumgebungen tabellarisch aufgeführt.

C.2.1 Sun

TABELLE 14. Testumgebung für SUN-Workstation

	Beschreibung
Rechner	Sun SPARCsystem 10
Betriebssystem	Solaris 2.3
Windowmanager	mwm
Libraries für Oberfläche	X11 Release 5 Library, X Toolkit Intrinsics (beide aus X11 Release 5) Motif Library der Version 1.2
Datenbank	Ingres Version 6.4
C++-Compiler	SPARCompiler C++ Version 4.0
C-Compiler	SPARCompiler C Version 2.01

C.2.2 Silicon Graphics

TABELLE 15. Testumgebung für Silicon Graphics-Workstation

	Beschreibung
Rechner	Silicon Graphics Indy
Betriebssystem	Irix 5.2
Windowmanager	4Dwm
Libraries für Oberfläche	X11 Release 5 Library, X Toolkit Intrinsics (beide aus X11 Release 5) Motif Library der Version 1.2.3

TABELLE 15. Testumgebung für Silicon Graphics-Workstation

	Beschreibung
Datenbank	Ingres Version 6.4
C++-Compiler	Silicon Graphics C++-Compiler Version 3.2.1
C-Compiler	Silicon Graphics C-Compiler Version 3.18

C.3 Bugs

- Existieren selektierte Objekte in kollabiertem Block bzw. Transaktion kann es zu Darstellungsfehlern kommen.
- Ausnahmen in oder aus einem kollabierten Block bzw. kollabierten Transaktion zeigen unter Umständen an die Stelle des Skript, an dem sich der Ansprungpunkt befand, als der Block bzw. die Transaktion noch nicht kollabiert waren.

D Literatur

[Masm90] Ioannis Masmanidis: Syntax und Compiler einer Sprache zur Bearbeitung paralleler und verteilter Abläufe. Studienarbeit Nr. 897, Fakultät für Informatik, Universität Stuttgart, 1990.

[Masm91] Ioannis Masmanidis: ConTract-Programmierung in Apricots - Sprachbeschreibung und Programmieranleitung. Fakultät für Informatik, Universität Stuttgart, 1992.

[Eitl92] Marcus Eitler: Eine graphische Oberfläche für die interaktive Steuerung langlebiger verteilter Abläufe, Diplomarbeit 956, Fakultät für Informatik, Universität Stuttgart, 1992.

[Schw93] Friedemann Schwenkreis: Management of the Control Flow and the Communication System. Fakultät für Informatik, Universität Stuttgart, 1993.

[Häu93] Michael Häuptle: Erweiterung eines Kommunikationsmechanismus für den rechnerübergreifenden Betrieb. Studienarbeit 1231, Fakultät für Informatik, Universität Stuttgart, 1993.

[TaPrg94] Martina Keller, Klaus Röhrle, Ralf Zimmermann, Michael Lindner, Jens Seifert: Dokumentation zum Fachpraktikum Transaktionales Programmieren. Fakultät für Informatik, Universität Stuttgart, 1994.

[MoRef93] Motif 1.2 Programmer‘s Reference Manual. SunSoft, 1993.

[MoPrg94] Dan Heller und Paula M. Ferguson: The Definitive Guides to the X Window System, Volume Six A, Motif Programming Manual. O‘Reilly & Associates, Inc., 1994.

[Det93] B. Detlaff: Skript zum Kompaktkurs X-Window System. Fakultät für Informatik, Universität Stuttgart, 1993.

[Str94] Bjarne Stroustrup: Die Programmiersprache C++, 2.Auflage. Addison-Wesley, 1994.

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

(Jens Seifert)