| | |
|---|---|
| **Prüfer:** | **Prof. Dr. A. Reuter** |
| **Betreuer:** | **Dipl.-Inf. Friedemann Schwenkreis** |
| | **Dipl-Inf. Helmut Wächter** |

| | |
|---|---|
| **begonnen am:** | **1. Juli 1992** |
| **beendet am:** | **31. Dezember 1992** |

| | |
|---|---|
| **CR-Klassifikation:** | **D.3.2, H.5.2** |

# Eine graphische Oberfläche für die interaktive Steuerung langlebiger, verteilter Abläufe

Marcus Eitler

Diplomarbeit Nr. 956

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20-22
D 7000 Stuttgart 80

# Kurzfassung

Die vorliegende Arbeit beschäftigt sich mit der graphischen Darstellung von langlebigen, verteilten Abläufen. Die Grundlage dafür liefert in diesem Zusammenhang das an der Universität Stuttgart am Institut für Parallele und Verteilte Höchstleistungsrechner entwickelte ConTract-Modell . Dieses Modell entspricht einem Konzept, das mit Unterstützung einer Datenbank, Abläufe mit besonders langer Laufzeit überwacht und zur Ausführung bringt. Die Abläufe werden dabei in Einzelschritte unterteilt, wobei diese auf verschiedenen Ausführeinheiten abgewickelt werden können.

Aufgabe dieser Arbeit war es, eine graphische Darstellung solcher Abläufe zu entwickeln, die es erlaubt, einen Ablauf visuell darzustellen und die Kontrolle darüber über eine Oberfläche zu steuern. Die Anordnung der Einzelschritte soll dabei die Abhängigkeiten dieser untereinander ersichtlich machen. Innerhalb der Arbeit sollten die Möglichkeiten der Darstellung erarbeitet und unter möglichst geringen Antwortzeiten realisisert werden. Es sollte eine geeignete Verbindung zum Gesamtsystem geschaffen werden, um die Steuerung des Ablaufs durch die Oberfläche zu sichern.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# KAPITEL 1 Einleitung

## 1.1 Einführung

Moderne Rechnersysteme sind heutzutage oft so aufgebaut, daß sie dem Betrachter auf den ersten Blick ein unüberschaubares Netz von mehr oder weniger zusammenhängenden Komponenten bieten. Diese Komponenten kennzeichnen sich dadurch, daß sie alle eine bestimmte Rolle im System einnehmen, sei es als Drucker, Netzwerkverwalter, lokaler Rechner oder als eine beliebig-integrierte Verarbeitungseinheit. Oftmals findet man auch mehrere Komponenten des gleichen Typs innerhalb eines Systems vor. Das allgemeinste Bestreben der Softwareentwicklung ist es daher, das Zusammenwirken dieser Komponenten so zu nützen, daß Abläufe in kürzester Zeit mit dem geringsten Aufwand erledigt werden können. Wichtig dabei ist nicht nur die Art und Weise mit der eine Aufgabe bewältigt wird, sondern auch die Sicherheit mit der die Bearbeitung zum Ziel führt. Insbesondere bei langen Verarbeitungen, deren Ausführzeiten eine kritische Grenze übersteigen, ist die Fehlertoleranz des Systems von großer Bedeutung.

Typische, standardisierte Anwendungen, die lange Laufzeiten haben können und über eine hohe Fehlertoleranz verfügen, sind Datenbankanwendungen. Solche Anwendungen unterliegen dem sogenannten Transaktionsparadigma, das unter Einhaltung des ACID-Prinzips[1], die Sicherheit der Abläufe garantiert. Ein erweitertes Transaktionsmodell, das über die Verarbeitung von Datenmengen hinaus geht, ist das ConTract-Modell [ReuWä90]. Dabei können beliebige Objekte in den Ablauf mit einbezogen werden (nicht nur Daten). Es findet dabei eine Unterteilung in einzelne Schritte statt, wobei sowohl alle Schritte wie auch der Gesamtablauf eine fehlertolerante Abwicklung durchlaufen sollen. Die Fehlertoleranz kennzeichnet sich in diesem Zusammenhang durch mögliche

1. Siehe Literatur über Datenbanken.

Komponsationsmaßnahmen im Fehlerfall, wodurch Änderungen durch vorher festgelegte Maßnahmen ausgeglichen werden.

Die genaue Struktur des gesamten Ablaufs ist nur durch eine graphische Anordnung der Einzelschritte genau zu erkennen. Es lassen sich daraus Zusammenhänge wie Parallelitäten, bedingte Abhängigkeiten und ähnliches leicht ablesen, was dem Anwender sonst nur durch ein hohes Maß an Kontextwissen gelingen würde. Wirklich sinnvoll wird eine so beschriebene, graphische Anordnung eines Prozesses erst, wenn diese Anordnung auch programmatisch eingesetzt werden kann. Dies ist der Fall, wenn die Möglichkeit besteht, den Kontrollfluß des Ablaufs aktiv zu beeinflussen, indem auf jeden Einzelschritt und auf den Gesamtablauf zugegriffen werden kann. In der Realisierung eines solchen Systems liegt der Schwerpunkt dieser Arbeit.

## 1.2 Aufgabenstellung

Im Rahmen des APRICOTS-Projekts am Institut für Parallele und Verteilte Höchstleistungsrechner der Fakultät Informatik an der Universität Stuttgart wurde ein prototypisches Laufzeitsystem zur interpretativen Abwicklung von Ablaufskripten entwickelt. Durch die Hinzunahme eines Ablaufmonitors soll die Benutzung des Systems erleichtert und veranschaulicht werden.

Der Aufbau gliedert sich in das Erstellen der graphischen Struktur und nachfolgend in der Visualisierung und Überwachung des Ablaufs durch die Funktionen des Monitors. Das Erstellen entspricht der Anordnung aller Einzelschritte nach einer vorgegebenen Abfolge. Die Darstellung des Ablaufs erfolgt über die Verbindung zu einer Kommunikationskomponente, die dem Monitor die jeweiligen Zustandsdaten der beteiligten Komponenten schrittweise übermittelt.

## 1.3 Gliederung der Arbeit

In Kapitel 2 wird das Konzept von APRICOTS vorgestellt. Das dritte Kapitel beschäftigt sich allgemein mit Anforderungen, die an eine Oberfläche gestellt werden. Das vierte Kapitel beschreibt die graphentheoretische Einordnung des Problems und versucht daraus Schlüsse zur Komplexität der Berechnung zu ziehen.

Das fünfte Kapitel zeigt die verwendeten Informationen auf, die zur Erstellung des Ablaufgraphen notwendig sind. Davon ausgehend werden die Verfahren zur Berechnung des Ablaufgraphen beschrieben. Im sechsten Kapitel wird der Bezug zur objektorientieren Einordnung des Gesamtsystems unter Verwendung entsprechender Entwicklungswerkzeuge hergestellt. Das siebte Kapitel befaßt sich mit Strategien zur Kommunikation und beschreibt die verwendeten Mechanismen. Im achten Kapitel wird die Arbeitsweise während eines Ablaufs beschrieben.

# KAPITEL 2 Allgemeine Betrachtungen zu APRICOTS und ConTracts

## 2.1 Einführung in das Gesamtsystem

Innerhalb des von der Deutschen Forschungsgesellschaft (DFG) initiierten Schwerpunktprogramms *Objektbanken für Experten,* wurde an der Universität Stuttgart das APRICOTS-Projekt (**A PR**ototype **I**mplementation of the **CO**n**T**ract **S**ystem) ins Leben gerufen [ReuWä90]. Gegenstand des Projektes ist die Entwicklung und Erprobung von langlebigen verteilten Abläufen, deren einzelne Ausführungseinheiten (*Steps*) unter Transaktionskontrolle ausgeführt werden . Diese Kontrolle beinhaltet beispielsweise:

- fehlertolerante Kontrollflußabwicklung für verkettete und verteilt ablaufende Transaktionen.
- robuste Kontextverarbeitung für zusammengehörige Verarbeitungsschritte.
- automatisches Wiederaufsetzen und Fortführen eines unterbrochenen Ablaufs.
- Synchronisation von Datenbankzugriffen über Transaktionsgrenzen hinweg.
- Verfahren zur Auflösung von Konflikten (Sperrenbehandlung).
- Zurücknahme (*Kompensation*) bereits freigegebener Änderungen bei Abbruch einer Anwendung.

Dabei sollen der Anwendungsebene die Behandlung von Systemfehlern, Betriebsmittelverwaltung, Synchronisation usw. durch eine fehlertolerante Betriebsumgebung verborgen werden. Undefinierte Zustände dürfen nicht auftreten. Bei einem Systemfehler muß die Anwendung nahtlos nach vorne fortgesetzt (*vorwärts-recovery*) oder zurückgesetzt (*kompensiert*) werden können.

Die prototypische Implementierung des ConTract Modells, APRICOTS, soll geeignete Konzepte zur Programmierung und zur Transaktionsverarbeitung auf verteilten Rechnern realisieren und diese in verständlicher Form darstellen. Dazu gehören einzelne Server, die Systemdienste anbieten (ConTract-Manager, Step-Computation-Server, Transaktions-Manager), ein Kommunikationssystem und geeignete Oberflächen zur Bedienung des Systems. Gegenstand dieser Arbeit ist es, eine graphische Oberfläche zu implementieren, die die ConTract-Überwachung und Steuerung erlaubt, das heißt den Ablauf sichtbar macht und Kommandos durch den Benutzer erlaubt (*suspend, resume, compensate* u.ä.). Darüberhinaus sollten Zusatzhilfen zu den einzelnen Bestandteilen, beispielsweise in Form von Informationsfenstern, erstellt werden, damit sich der Benutzer nach Bedarf zusätzlich Detailinformationen, zu jeder im System beteiligten Komponente, einholen kann.

## 2.2 Das ConTract-Modell

Das ConTract-Modell ist eine Erweiterung des klassischen Transaktionskonzeptes, um Transaktionsmechanismen für funktionell erweiterte Datenbanksysteme[1] zu entwickeln [ReuWäSchw92]. Beide beziehen ihre Notwendigkeit aus der Forderung nach der fehlertoleranten Verarbeitung datenintensiver Berechnungen. Die klassische Methode stützt sich dabei auf Arbeitseinheiten, die logisch atomar definiert sind und geschützt ablaufen; als Resultat entsteht eine wohldefinierte, konsistente Zustandsänderung im System, in den meisten Fällen eine Änderung in einer Datenbank. Diese Änderungen unterliegen dem ACID-Prinzip, dessen Einhaltung die Konsistenz der Daten gewährleistet. ConTracts hingegen, verstehen sich als transaktionsübergreifende Verarbeitungskonstrukte, bei denen nur der Ablauf definiert wird, die fehlertolerante Verarbeitung aber dem System überlassen wird. Dieses verbindet klassische Transaktionskontrollmechanismen mit Betriebsystem- und Netzwerkdiensten und erhält so ein größeres Abstraktionsniveau. Daraus ergibt sich eine über mehrere Ausführungseinheiten gelegte Kontrollflußbeschreibung, die es über den ihm zugrundeliegenden, objektorientierten Ansatz erlaubt, beliebig aufgebaute Objekte als Einheit auffassen und zu bearbeiten. Beispielsweise soll das System in der Lage sein, erworbene Sperren, die von einer Ausführeinheit gehalten werden, an eine andere Ausführeinheit *weiterzureichen,* wenn diese zur gleichen Kontrollsequenz gehören.

Die Forderung nach einem Modell wie dem ConTract-Modell, ist entstanden aufgrund weitgehender Vernetzung moderner Rechenanlagen in den letzten Jahren und die damit verbundene Verteilung der darauf laufenden Anwendungen. Mit den vorhandenen Betriebs- und Datenbanksystemen ist dabei die Forderung nach fehlertoleranter Abwicklung, insbesonderer langlebiger, verteilter Ausführungen nicht mehr gesichert, was durch das ConTract-Konzept gewährleistet wird. Dazu bedarf es einer mehrstufigen Programmierung, die erstens der Kontrollstruktur der Anwendungswelt entspricht, also dem logischen Aufbau und zweitens unabhängig von der momentanen Umgebung[2] den Ablauf des Systems sichert. Nach einem auftretenden Fehler soll, wenn möglich fortgesetzt, statt zurückgesetzt werden, eventuell auf einem anderen Rechner[3]. Taucht dabei ein nicht zu behebender Fehler auf, so soll in kontrollierter Weise der Ausgangszustand wieder hergestellt werden und zwar in Rückwärtsschritten, die in umgekehrten Folge ihres Auftretens die Steps einzeln zurücksetzt[4]. Dieses geschieht nur nach expliziter Anforderung durch

1. Arbeitstitel „Objektbanken“.
2. Verfügbarer Plattenplatz, Rechner, Netzdienste usw.
3. Migration.
4. Kompensieren.

den Benutzer . Diese mehrstufige Programmiermethode trennt Anwendungs- und Kontrollaspekte, mit anderen Worten: Ablauf von Einzelberechnung.

### 2.2.1 Die ConTract-Programmierung

Es werden in der Programmierung eine endliche Anzahl auszuführender kleinster Einheiten angegeben. Dieses geschieht in einem *Skript,* diese Einheiten werden *Steps* genannt.

Die zweigeteilte Anlage der Erstellung eines ConTracts:

1. Programmierung des *Skriptes*, Festlegen des Kontrollflußes, Anordnen der Steps und deren Abhängigkeiten untereinander.
2. Kodierung der sequentiellen Einzelverarbeitungen, der *Steps*.

Die Programmierung erfolgt modular und kann ohne weiteres von verschiedenen Programmierern erstellt werden.

#### 2.2.1.1 Funktion und Aufgabe eines Skriptes

Zentrale Aufgabe des Skriptes ist die Beschreibung eines korrekten Ablaufs innerhalb einer transaktionsübergreifenden Verarbeitung. Der somit erstellte Kontrollfluß erlaubt die Ausführung verschiedener Konstrukte, wie beispielsweise:

- Sequenz
- Verzweigung
- Schleife
- Parallelverarbeitung

  und
- parforeach-Anweisungen; womit die parallele Verarbeitung einer qualifizierten Tupelmenge gemeint ist.

Als Notation wird eine einfach gehaltene, prozedurale Sprache verwendet, die einen Kontrollfluß entstehen läßt, der sich auf eine spezielle Petri-Netzvariante, ein Prädikat-Transistionsnetz, abbilden läßt. Dabei entsprechen:

- Die *Steps* des ConTracts den *Stellen* des Prädikat-Transitions-Netzes.
- Die *Events*[5] des ConTracts den *Transitionen*.
- Erscheinen und Auftreten im Netzgraphen geschehen immer abwechselnd, wobei jeder Step einen eindeutigen Vorgängerevent und einen oder mehrere Nachfolgerevents haben muß. Jeder Event hat einen oder mehrere Vorgängersteps und einen oder mehrere Nachfolgersteps; Ausnahmen sind Event 0, der keinen Vorgänger und Event 1, der keinen Nachfolger hat. Diese beiden umrahmen den ConTract.

Die Koordination innerhalb des Skriptes erfolgt mittels Kontrollkonstrukten, die sequentielle Verarbeitungen *(Steps)* umfassen. Solche Konstrukte sind PARBEGIN - PAREND, IF - ENDIF usw. Es entstehen dadurch neue übergeordnete Strukturen, die als Blöcke zu bezeichnen sind. Dadurch entsteht im ganzen betrachtet ein aus

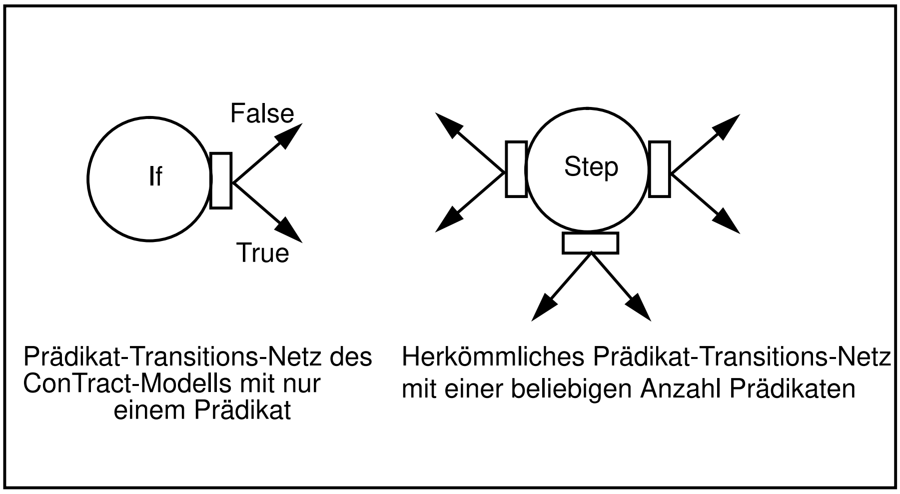
5. Erklärung im folgenden.

Blöcken zusammengesetztes Modell. Diese Blockstruktur ist besonders für den Aufbau des Ablaufgraphen von Bedeutung.

Unterscheidung

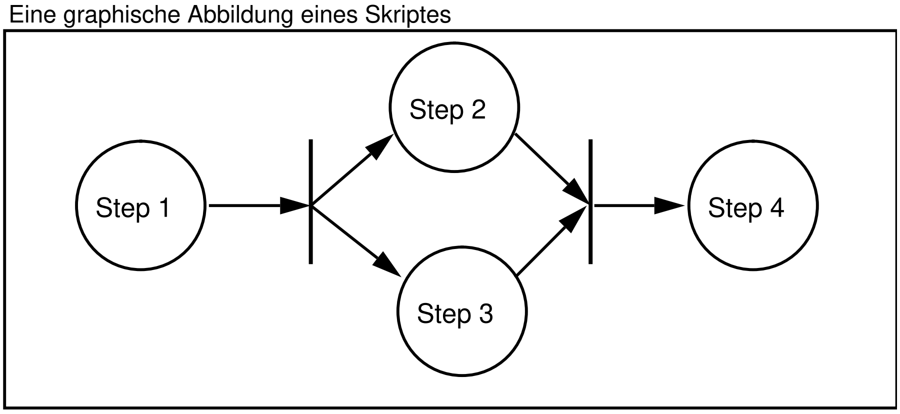
Der Unterschied zwischen herkömmlichen Petri-Netzen und Prädikat-Transitionsnetzen ist die mittels Abfrage bedingte Steuerung des Kontrollflusses. Das heißt, der Übergang zwischen Step und Event ist mit einer Bedingung verbunden.

**ABBILDUNG 1** Prädikat-Transitions-Übergänge




Eine korrekte ConTract-Abwicklung zur Laufzeit sichert der sogenannte ConTract-Manager, der das ConTract-Skript kennt und dieses interpretieren kann, siehe dazu [Wahl91]. Wichtig in diesem Zusammenhang ist die Tatsache, daß ein erstelltes Skript einen rein statischen Charakter hat. Ebenso wie die Erstellung der graphischen Oberfläche, die nach dem ersten Aufbau ihre Gestalt nicht mehr ändert. Der **Datenfluß** des ConTracts kann dagegen jedesmal einen anderen Verlauf nehmen, was sich auch optisch niederschlagen kann, aber nicht muß. Ein anderer Verlauf des **Kontrollflusses** wird dagegen immer sichtbar, siehe Kapitel 4 Seite 34.

**ABBILDUNG 2** Eine graphische Abbildung eines Skriptes



#### 2.2.1.2 Die Bedeutung der Steps

Steps sind die kleinsten Einheiten im Skript, und als solche unterliegen sie einem rein sequentiellen Ablauf, sie erscheinen wie Unterprogramme in prozeduralen Programmiersprachen. Man kann ein Skript auch als ein in seine sequentiellen Bestandteile, eben den Steps, aufgeteiltes Netz verstehen. Durch den ConTract-Manager erhält jeder Step eine Transaktionsnummer. Diese erlauben es, mehrere Ausführungen zu gruppieren und als weitere, übergeordnete *Ausführungen* zu definieren. Es entstehen dadurch neue Objekte, die den Unterobjekten[6] hierarchisch übergeordnet sind. Somit entsteht eine Beziehung unter den ausgewählten Steps, die bei Abbruch eines Steps, die vorher geschaffene, übergeordnete Ausführung ebenfalls abbricht. Um dieses Konzept erfolgreich zu unterstützen, müssen Ein- und Ausgabedaten der Steps stabil gespeichert werden. Dieses erfolgt im *Kontextsystem,* einer weiteren APRICOTS-Komponente [BuRai90]. In objektorientiertem Sinne können die Steps als Unterobjekte der Skripte angesehen werden.

Steuerung durch *event-trigger*-Mechanismus [DaHsuLa90]

Jeder Step wird nur ausgeführt werden, wenn der ihm assozierte Start-Event *getriggert* wurde. Ein erfolgreich bearbeiteter Step sendet ein *token* an alle die durch die Prädikate qualifizierten Termination-Events. Ein Event wird getriggert, wenn er die für ihn bestimmte Anzahl *tokens* erhalten hat[7]. Dieses entspricht verfahrensmäßig einem Prädikat-Transitionsnetz, wie es oben beschrieben wurde.

**TABELLE 1** Statische Daten eines ConTracts

| Objekte | Daten |
|---|---|
| ConTract | ConTract_Name<br>Skriptdaten |
| Steps | Step_ID<br>Start_Event<br>Termination_Event<br>Invariantenbehandlung |
| Events | Event_ID<br>expected_tokens |

#### 2.2.1.3 Die Bedeutung der Events

Jeder Übergang eines Steps, zu einem ihm nachfolgenden Step, ist gekennzeichnet durch das sogenannte *Triggern* eines Events. Events erscheinen innerhalb der Skriptprogrammierung nicht explizit - es gibt keine Konstrukte die Events darstellen. Stattdessen werden sie durch einen Übersetzer, der das Skript bearbeitet, erzeugt, mit einer Kennung markiert und als statische Daten in einer Datenbank abgelegt. Der Contract-Manager verwendet diese Daten bei einem Wiederanlaufen eines abgebrochenen ConTracts. Es werden bei einem Suspendieren eines ConTracts zusätzliche Kontextinformationen abgelegt, die den momentanen Zustand des ConTracts beschreiben. Diese werden beim Wiederaufbau des suspendierten ConTracts verwendet und sichern den Fortlauf aus dem richtigen Zustand heraus.

6. hier die bisherigen Ausführungen.
7. Entspricht den expected_tokens.

#### 2.2.1.4 Fehlertoleranz und Fehlerbehandlung

*Ein ConTract terminiert unter (System-) Garantie in endlicher Zeit und in einem korrekten Endzustand.* Siehe [ReuWä91]. Das bedeutet:

1. Ein ConTract befindet sich immer in einem wohldefinierten Zustand. Um diesen unter allen Umständen rekonstruieren zu können, werden Daten über den bisherigen Verlauf mitprotokolliert (logging).
2. Ein ConTract läuft entweder ganz durch oder alle bisherigen Änderungen werden logisch zurückgesetzt. Hier besteht Analogie zwischen dem herkömmlichen Transaktionskonzept, das eine atomare Ausführung einer Transaktion vorsieht.

Recovery

Darüberhinaus bestehen zusätzliche Verfahren[8], die einen einmal unterbrochenen Ablauf wieder aufnehmen. Ein Abbruch kann dabei sowohl durch einen Systemfehler hervorgerufen, wie durch den Benutzer ausgelöst werden[9]. Durch die Hinzunahme der mitprotokollierten Zustandsdaten kann ein Fortfahren des ConTracts an jeder Stelle im Skript erfolgen, was als *forward-recovery* bezeichnet wird. Im Gegensatz dazu, ist ein Fortfahren einer Transaktion nach einem Abbruch nicht möglich. Mit diesem Konzept soll verhindert werden, daß Prozesse mit langer Laufzeit bei Auftreten eines Fehlers nochmal ganz von vorne anlaufen müssen. Ein Schwachpunkt des Transaktionsmechanismus soll auf diese Weise behoben werden, da es auch bei Transaktionen nicht immer unbedingt notwendig sein müßte, ein ROLLBACK durchzuführen und daraufhin die ganze Aktion neu zu starten. Diese restriktive Regel wird im ConTract-Konzept aufgehoben, muß allerdings durch ein aufwendiges Mitprotokollieren bezahlt werden. Wobei auch hier darauf hingewiesen werden muß, daß die einzelnen Ausführungen durchaus noch dem herkömmlichem Transaktionsparadigma unterliegen können. Nur das in objektorientiertem Sinne hierarchisch höher gelegene *Objekt* ConTract wird auf diese neue Weise behandelt.

Compensate

Ein Sonderfall bei der Fehlerbehandlung tritt auf, wenn eine Ausführungseinheit unwiderruflich abbricht oder eine sogenannte *conflict-resolution* scheitert. Dies hat zur Folge, daß ein ConTract nicht zu Ende geführt wird, da ein Step nicht ausgeführt werden kann. Ein ROLLBACK, wie in herkömmlichen Datenbanksystemen, kann hier nicht ausgeführt werden, da keine dauerhaften Sperren auf den durch die Steps behandelten Datenobjekte, gehalten werden. Das heißt, die geänderten Datenbestände können nicht einfach mit den alten Beständen, wie sie durch das Datenbanksystem mitprotokolliert wurden, überschrieben werden, da die freigegebenen Änderungen der Steps zwischenzeitlich durch andere Transaktionenen geändert werden konnten. Es muß daher eine Umkehroperation anlaufen können, die alle bisherigen Änderungen logisch *umkehrt.* Für jeden Step muß ein *Kompensations*-Step existieren. Diese Kompensierung wird im Skript festgelegt und muß vom Benutzer angestoßen werden. Es werden dafür in [ReuWä91] errechnet mehrere Verfahren besprochen:

1. *Sequentielle Kompensation von Step-Gruppen.*

8. zusätzlich in bezug auf Transaktionen.
9. Auslösen eines *Suspend.*

Dabei werden die Startzeiten der einzelnen Steps mitprotkolliert und diese in umgekehrter Reihenfolge kompensiert. Wichtig ist dabei auch, daß zu Gruppen zusammengefaßte Steps zuerst alle kompensiert sein müssen, bevor die Steps einer andere Gruppe kompensiert werden können.

2. *Rückwärts-Interpretation des Skriptes.*

   Das Skript wird einfach in ungekehrter Reihenfolge bearbeitet.

3. *Rückwärts-Interpretation der Log-Daten.*

   Es wird der zeitliche Verlauf, wie er durch die Log-Daten festgehalten wurde, umgekehrt. Alle zeitlichen Abhängigkeiten werden in den Log-Dateien mitgeführt, somit alle Synchronisationsbedingungen eingehalten.

Grundsätzlich ist sowohl eine Kompensierung durch den ConTract-Manager selbst, wie auch durch einen separaten Compensation-Manager möglich; wobei sich am Konzept in beiden Fällen keine Änderung ergeben würde, lediglich die Effizienz des Vefahrens kann durch eine Trennung der beiden Komponenten verbessert werden.

### 2.2.2 Übersicht über die Struktur von ConTracts

Im folgenden erscheint eine Tabelle, die statische und dynamische Daten eines ConTracts auflistet und die Schemadaten tabelliert [Masman92], [Wahl91].

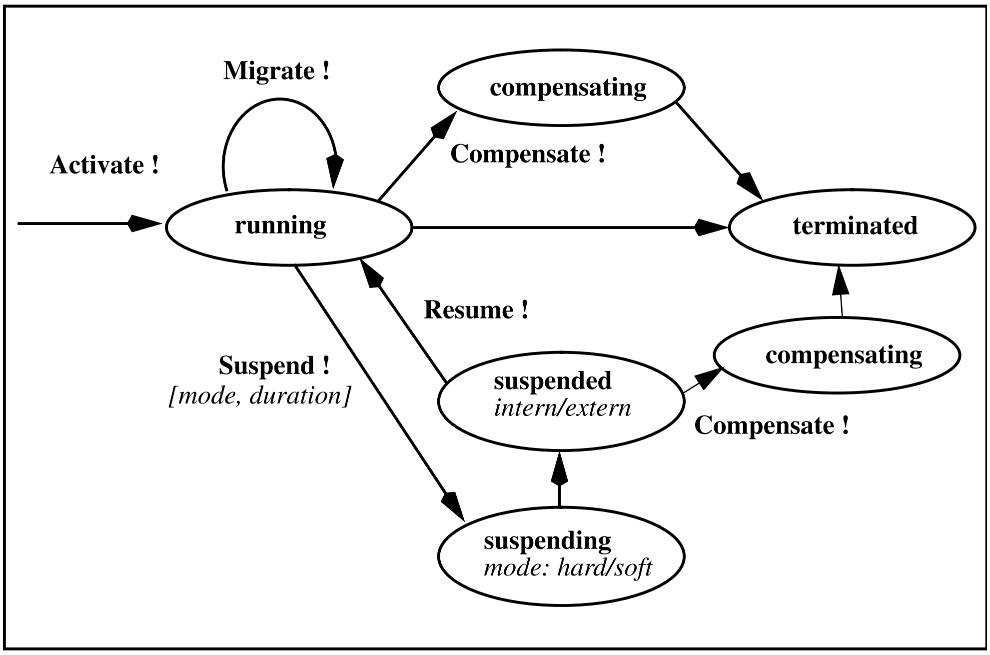
**TABELLE 2** Verarbeitungszustand eines ConTracts

| **Schemadaten** | Ablaufbeschreibung (Skript-Definition)<br>Programmierschritte (Step-Code)<br>Konfigurationsbeschreibung<br>(RM, SCS u.a. Server-Prozesse) |
|---|---|
| **Dynamischer Zustand**<br>a) Identifizierung<br>und Adressierung | ConTract-Id<br>Adresse des zuständigen CMs<br>Anwendungs-Id bzw. Benutzername<br>Terminaladresse |
| b) bisheriger Verlauf | aktueller ConTract-Zustand (running ...)<br>angefallene Kontextelemente<br>Synchronisationszustand (Invarianten etc.)<br>erzeugte Sicherungspunkte<br>Zustände aller Kontrollflußereignisse<br>(Markenzähler)<br>Anzahl abgeschlossener und momentan<br>aktiver Instanzen eines Step-Typs |
| c) aktueller Stand einer<br>Skriptausführung | Dynamische Daten der aktiven Step-Instanzen<br>- Versionszähler<br>- Anzahl paralleler Aktivierung<br>- Retry-Zähler<br>- Status des Bearbeitungsauftrags<br>- Nummer und Status der Step-TA<br>verzögerte Ereignisse und Steps (suspending) |

In der Datenbank werden außerdem weitere Daten über Steps und Events gespeichert, anhand derer die Reihenfolge des Erscheinens abzuleiten ist. Diese sind besonders für den Aufbau der Oberfläche relevant, da daraus wichtige Informationen über die Reihenfolge der graphischen Objekte[10] abzuleiten sind.

Den Ablauf und die dabei möglichen Zustände zeigt die nächste Abbildung. Der Benutzer kann die zulässigen Befehle aus dem jeweiligen Zustand heraus ausführen und erreicht so eine Zustandänderung, wobei hier deutlich wird, daß ein ConTract enden **muß**.

**ABBILDUNG 3** Zustandsdiagramm eines ConTracts




## 2.3 Der Aufbau von APRICOTS

Der ConTract verarbeitende Prototyp bietet seinem Benutzer unterschiedliche Dienste an, welche nach Zusammenfassen ihrer Funktionalität in Klassen eingeteilt sind und Resource Manager Klassen[11] genannt werden. Die dazu gehörenden Kontroll- und Synchronisationsmechanismen sollen dabei der Anwendungsebene völlig verborgen sein. Der Aufbau ist modular, sodaß ein Zusammenbinden mehrerer Klassen eine weitere modulare Einheit schafft. Man bezeichnet diese als Komponenten [Laube92].

### 2.3.1 Übersicht der Bestandteile

Das Gesamtsystem besteht aus einzelnen, miteinander kommunizierenden Kom-

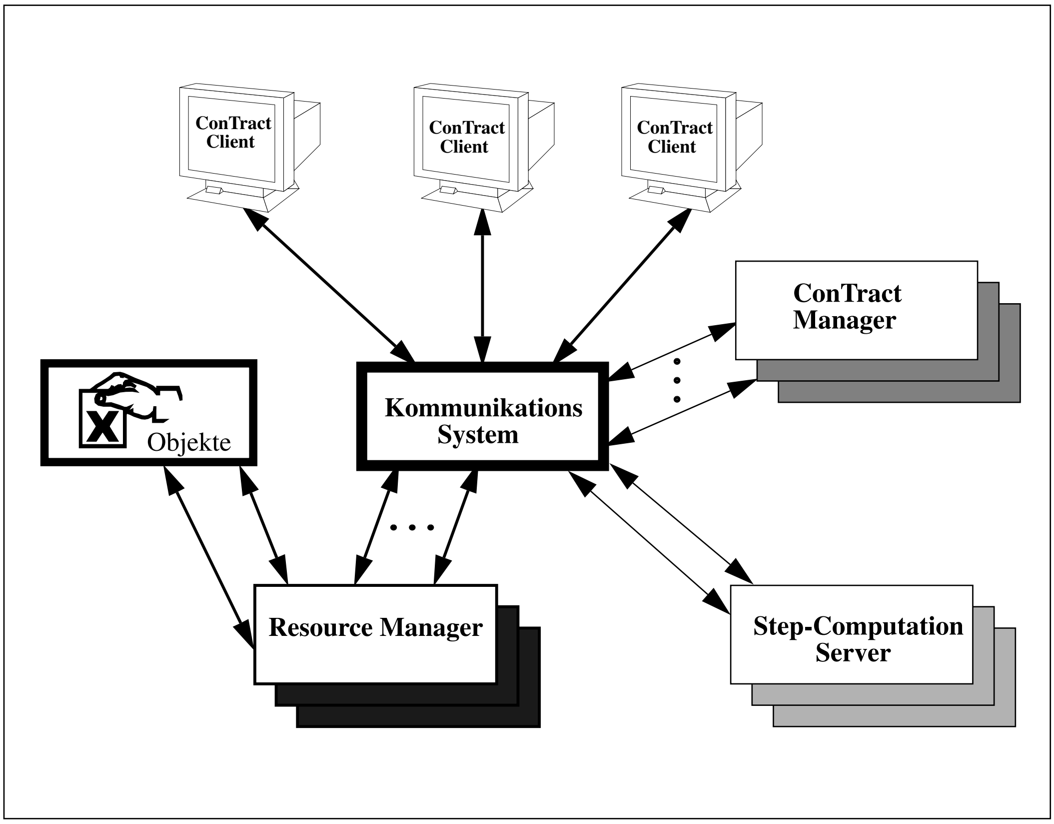
10. Als Objekte sind hier sowohl Steps, wie auch Events gemeint.

11. Objekte

ponenten; das Programmiermodell erlaubt dabei mehrstufige und wechselseitige Beziehungen und ist verglichen mit dem herkömmlichen Client/Server-Modell eine Erweiterung dessen, da in diesem eine eindeutige Zuordnung des Clients und des Servers gegeben sein muß.

Um abzusichern, daß die Abwicklung robust abgearbeitet werden kann, müssen sowohl Daten über lokale Zustände, wie auch solche über den Zustand der Gesamtausführung stabil verwaltet werden. Es reicht nicht, das herkömmliche Logging auf die einzelnen Ausführschritte anzuwenden, sondern weitere Daten, die den, uunter Umständen verteilten Ablauf zum augenblicklichen Zeitpunkt genau beschreiben, müssen global und dauerhaft gesichert werden. Es läßt sich so bei Ausfall des Systems genau nachvollziehen, welcher Knoten für einen Abbruch verantwortlich war und anhand geeigneter Verwaltungsmechanismen einen Weiterbetrieb im Sinne von *Forward-Recovery* ausführen [ReuWäSchw92].

**ABBILDUNG 4** Aufbau des APRICOTS-Gesamtsystems




### 2.3.2 Die Rolle des Kommunikationssystems

Das Kommunikationssystem hat innerhalb des Gesamtsystems die zentrale Rolle, es verknüpft unterschiedliche Komponenten miteinander, verwaltet Betriebsmittel und sichert bei Betriebsausfall eines Knotens den Fortlauf eines ConTracts. Siehe dazu [Laube92]. Darüberhinaus erlaubt es den Komponenten auf einem hohen Abstrahierungsniveau Dienste anzufordern[12] und diese einzubinden. Insbesondere

12. Es wird nur logisch angefordert.

hat es die Aufgabe, den ConTract-Clients den wahren Ablauf zu verbergen, diese müssen nicht *wissen* wo ihr Auftrag *erledigt* wird. Es **kann**, ähnlich dem RPC-Mechanismus, nur über den Befehl ein Dienst angefordert werden. Dies geschieht ohne festzulegen, auf welchem Rechner die Ausführung stattfinden soll. Der Unterschied besteht darin, daß mit einem RPC-Aufruf Prozeduren und in diesem Modell ganze Dienste gemeint sind. Es ist aber auch möglich, einen Dienst auf einem bestimmten Rechner zu schicken. Dabei können Dienste auf zwei verschiedene Arten angefordert werden:

1. Bei der synchronen Anforderung wird der Kontrollfluß solange angehalten, bis der Dienst beendet ist.
2. Ansynchrone Anforderung bedeutet, es kann (während der Dienst läuft) im Kontrollfluß weitergearbeitet werden, die Ergebnisse werden erst später verwendet. Der Zugriff auf diese erfolgt über einen callback-Dienst, welcher dem System bekannt gemacht wird.

### 2.3.3 Die Rolle des ConTractManagers

Der ConTract-Manager übernimmt die in Bereich 2.2.1 auf Seite 5 beschriebene Aufgabe der Skriptabarbeitung und somit das Ausführen eines oder mehrerer ConTracts innerhalb des Gesamtsystems.

Seine Hauptaufgabe besteht darin, die Steps auf Transaktionen abzubilden und die Bearbeitung transaktions-übergreifender Kontrollmechanismen, die im Skript festgelegt wurden und die nun zur Ausführung kommen sollen, abzusichern. Darüberhinaus muß der ConTract-Manager Konfliktsituationen beseitigen und notfalls eine oben beschriebene Kompensation ausführen.

### 2.3.4 Die Rolle der Step Computation Server

Step Computation Server implementieren vom ConTract Manager innerhalb eines Skriptablaufs aufgerufene Step-Programme. Es sind dies die eigentlichen ConTract-Programmschritte, deren Ablauf von diesem ausgelagert werden. Jeder Programmschritt läuft außerhalb des ConTracts als eigener Server. Zusätzliche Aufgaben sind Invariantenbehandlung, Kontextbearbeitung oder RM_Calls [Laube92][BuRai90][Wahl91].

### 2.3.5 Die Rolle des Objektverwalters*(Resource Manager)*

Der Objektverwalter liegt an der Schnittstelle zwischen den Step Computation Servern und den gemeinsam, nutzbaren Objekten. Es kann sich dabei um eine Datenbank handeln, aber auch ganz andere Komponenten können benutzt werden. Nach Möglichkeit sollten die Objektverwalter Funktionen bereitstellen, die das Recovery des ConTract-Managers unterstützten. Dies kann beispielsweise dadurch erreicht werden, wenn alle Komponenten eines Knoten ein gemeinsames Log verwenden [ReuWäSchw92]. Handelt es sich dabei um eine verteilte Datenbankanwendung, so kann dabei das herkömmliche Zweiphasen-Commit *aufgeweicht* werden.

### 2.3.6 Die ConTract Clients

ConTract Clients sind interaktive Anwendungen, die den Benutzer gegenüber dem System repräsentieren. Es sind dies beispielsweise graphische Oberflächen, welche zu erstellen die Aufgabe dieser Arbeit ist. Diese hat als Hauptaufgabe den

Ablauf eines ConTracts graphisch darzustellen und gegebenenfalls durch Kommandoeingabe zu beeinflussen (starten, suspendieren, Dienste migrieren usw.). Sie muß dazu anhand der Daten, die in der Datenbank über den ConTract gespeichert sind, die graphische Struktur aufbauen. Durch die Verbindung über das Kommunikationssystem, muß sie die Zustandsänderungen des ConTracts, die der ConTract Manager während des Ablaufs ausgibt, durch visuelle Veränderung des Graphen und dessen Knoten anzeigen. Zusätzlich sollen für den Benutzer weitere Informationen über die Knoten während der Laufzeit erhältlich sein. Dies entspricht einem beiderseitigen Client-Server-Modell, indem der ConTract Client sowohl Server wie auch Client ist. Gleiches gilt für das Kommunikationssystem.

# KAPITEL 3 Anforderungen an eine Oberfläche

Eine Oberfläche ist, wie aus dem Namen hervorgeht, eine Anwendung, die an oberster Stelle eines Gesamtsystems den Zugang zu diesem bereitstellen soll. Dabei ist nur von Bedeutung, welche Funktionalitäten innerhalb des Gesamtsystems verfügbar sind, nicht wie sie realisiert werden. Nach dem ISO-7-Schichtenmodell[1] befindet sich eine Oberfläche auf Schichtebene sieben, also auf der Anwendungsebene. Die Interaktion, die zwischen Benutzer und Rechner stattfindet, ist entscheidend von der **Benutzerfreundlichkeit** der Schnittstelle zwischen diesen beiden *Endpunkten* abhängig. Deshalb kann die Benutzerfreundlichkeit einer Oberfläche, bei der qualitativen Bewertung, als Maßstab einen großen Einfluß ausüben. Im folgenden werden dafür Funktion und Konzepte der Steuerung einer Oberfläche beschrieben und gegenübergestellt.

## 3.1 Funktion einer Oberfläche

Der Benutzer, der mit der Oberfläche arbeitet, kann mit unterschiedlichen Gestaltungselementen der Anwendung konfrontiert werden. Diese sind für dessen subjektive Bewertung der Benutzerfreundlichkeit entscheidend. Die Akzeptanz einer Anwendung hängt wesentlich von der Akzeptanz dieser gestalterischen Elemente ab, denn das ganze interaktive Geschehen wird mit diesen kontrolliert. Weiter davon abhängig sind die Effektivität, mit der die Aufgabe bewältigt wird und die Unterstützung des Benutzers durch Hilfen, die ihm die Orientierung innerhalb der Oberfläche erleichtern. Dies kann unter Berücksichtigung der *Selbstbeschreibungsfähigkeit* der Elemente geschehen, indem beispielsweise Kommandos ähnlich ihrer realsprachlichen Verwendung benannt werden[2].

1. Siehe ISO-OSI Referenzmodell.
2. beispielsweise Verwenden von ***start*** zum Starten eines Ablaufs.

Für den Dialog wichtig sind die Ein-/Ausgabemedien. Diese müssen unter dem Aspekt der Eignung für die jeweilige Aufgabe ausgesucht werden. Ebenso und besonders bedeutend für die vorliegende Aufgabe ist die Gestaltung des Bildschirms in inhaltlichem und konzeptionellem Aufbau: was wird wo, wie angezeigt? Welche Möglichkeiten der Darstellung gibt es und wie werden sie für die jeweilige Aufgabe verwendet.

### 3.1.1 Programmsteuerung durch die Oberfläche

Jedes Konzept einer Programmsteuerung ist gebunden an Einschränkungen, die sich teils aus der Aufgabenstellung, teils aus der Umgebung (vorhandene Hardware usw.), in der ein Benutzer sich befindet, ergeben. Die Auswahl einer bestimmten Form der Programmsteuerung ist abhängig vom Kenntnisstand des Benutzers über das von ihm zu bedienende Programm. Übung und Erfahrung des Benutzers wirken sich auf sein Verhalten gegenüber der Oberfläche aus. Ziel ist es daher eine Umgebung zu schaffen, die sowohl aufgabeninhaltlich, wie auch benutzerspezifisch den Anforderungen weitgehend genügt [Oet88]. Zur besseren Strukturierung werden dafür die möglichen Befehle kontextabhängig in Klassen eingeteilt, wobei sich zwei Richtungen ergeben:

- **Basisbefehle** betreffen den gesamten zu bearbeitenden Bereich und sollten leicht zu handhaben sein.
- **Spezialbefehle** beziehen sich auf bestimmte Bereiche oder Abschnitte.

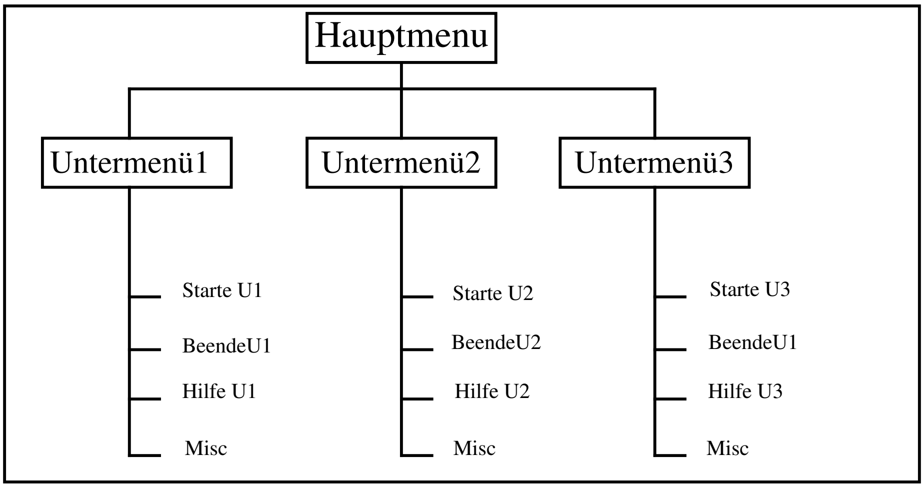
Basisbefehle besitzen während des gesamten Ablaufs die gleiche Bedeutung und sollten deshalb an jeder Stelle gleich zu bedienen sein. Beispielsweise sollte ein in Kapitel 1 beschriebener Suspend-Vorgang eines ConTracts, an jeder Stelle im Ablauf in gleicher Weise auszuführen sein. Spezialbefehle beziehen sich jeweils auf einen bestimmten Teil der Anwendung und sind ohne Kontextwissen meist nicht sinnvoll nutzbar. Bezogen auf die ConTract-Verarbeitung ist beispielsweise eine Detailbeschreibung der aktuellen Ausführeinheit, über den Aufruf eines Spezialbefehls, denkbar.

Oberflächen, die bestimmte Konzepte zur Programmsteuerung benutzen, verwenden diese in den meisten Fällen nicht in reiner Form. Vielmehr sind Mischformen am häufigsten implementiert, da diese am besten die gestellten Anforderungen durch mehr Flexibilität erfüllen können.

#### 3.1.1.1 Menüsteuerung

Bei der Menüsteuerung wird dem Benutzer eine meist fest vorgegebene Menge von Befehlen zu Verfügung gestellt. Diese Befehle sind in ihrer Funktionalität meist baumförmig angeordnet. Der Durchlauf durch die Verzweigungen gestaltet sich abhängig vom Bildschirm der verwendet wird. So wird bei einem Bildschirm, der nur Text ausgeben kann, nach jeder Auswahl, die nächste Bildschirmmaske aufgerufen. Bei pixelorientierten Bildschirmen, wie sie im Workstationbereich vorkommen, finden Menüleisten am oberen Ende des jeweiligen Fensters Verwendung. Bei Auswahl eines Menüpunktes kann dieser markiert werden, indem er optisch invers dargestellt wird.

**ABBILDUNG 5** Beispiel eines Menübaums




Zur Aktivierung einer Verzweigung wird im Bildschirmmenü meist eine Zahl eingegeben, die als Stellvertreterfunktion für einen Auswahlpunkt fungiert. Bei graphischen Bildschirmen wird innerhalb von Menüleisten die Auswahl mittels eines Zeigers, meistens mit dem der Maus, getroffen. Programmsteuerung mittels Menü ist, falls nicht sehr viele Verzweigungen auftauchen, sehr übersichtlich und durch die leichte Bedienbarkeit nur wenig fehleranfällig. Der Aufbau eines Menüs sollte nach Häufigkeit und Wichtigkeit der Funktionen erfolgen.

#### 3.1.1.2 Funktionstastensteuerung

Mit der Funktionstastensteuerung ist die Steuerungs des Programmablaufs mittels den zusätzlichen Tasten einer Rechnertastatur gemeint. Diese zusätzlichen Tasten werden mit Funktionen belegt, die nach Betätigen der Taste ausgeführt werden. Die Aktivierung und Navigation im Programmablauf unterliegt dem sequentiellen Betätigen dieser gesonderten Tasten. Die verwendete Anzahl der Sondertasten sollte klein sein, da sich das mnemotechnische[3] Verhalten des Benutzers nur bei einer kleinen Zahl positiv auswirkt. Programmsteuerung durch Belegung von Funktionstasten wird oft, insbesondere bei Bildschirmen die Graphik unterstützen, mit der menügesteuerten Technik kombiniert. Innerhalb der Menüstruktur werden Auswahlpunkte zusätzlich mit einer Funktionstaste belegt.

**ABBILDUNG 6** Menü der graphischen Oberfläche mit Funktionstastenbelegung

| ConTract | Step | Conflict |
|---|---|---|
| Start | R6 | |
| Compensate | L4 | |
| Suspend | | |
| Resume | Meta-R | |
| Migrate | R4 | |
| Take Checkpoint | Meta-T | |

3. Mnemotechnik: Einprägen von Stoff durch besondere Lernhilfen.

So soll beispielsweise ein Startvorgang immer mit **einer** speziellen Taste initiiert werden können, ebenso ein Beenden. Die Funktionstasten sollten über die gesamte Dauer und wenn sie Basisfunktionen repräsentieren, über alle Bereiche ihre Bedeutung beibehalten.

#### 3.1.1.3 Steuerung durch Kommandos

Mit Kommandosteuerung wird eine Steuerung des Systems durch Eingabe von Befehlen, die auf einer speziellen Sprache beruhen, bezeichnet. Oft können Kommandos mit ihrem Anfangsbuchstaben ersetzt werden und durch Kombination mehrerer Befehle lassen sich Makrobefehle auslösen. Dies ergibt große Freiheitsgrade für den Benutzer, verlangt von ihm aber auch eine fundiertes Wissen über seine Umgebung. Kommandosprachliche Steuerung unterstützt nicht die an eine moderne Oberfläche gestellte Forderung nach Erkennung der Struktur, anhand hierarchischer Kommandosequenzen. Sie eignet sich zur Steuerung von Betriebssystemabläufen oder für natürlichsprachliche Anfragesysteme wie SQL-Datenbanksysteme.

### 3.1.2 Dialogformen

Der Dialog zwischen Mensch und Maschine ist vom jeweiligen Informationsaustausch zwischen den beiden Endpunkten abhängig. Dabei ist jede Programmsteuerung an eine bestimmte Dialogform gebunden, es werden hauptsächlich drei Formen unterschieden, die aber nicht näher beschrieben werden:

- Vom Benutzer initiierter Dialog
- Vom Rechner initiierter Dialog
- Beidseitiger Dialog

In modernen Systemen ist vor allem die dritte Art des Dialogs von Bedeutung, bei der sich Rechner und Benutzer in der aktiven Rolle stets abwechseln. Das bedeutet, daß auch der Rechner in bestimmten Situationen die Entscheidung fällt. Grundlage dafür sind Programmiermodelle, die in bestimmten Situationen unter Berücksichtigung bestimmter Auswahlkriterien ganz spezielle Entscheidungen zulassen. Beispielsweise werden bei der ConTract-Abarbeitung bedingte Verzweigungen mittels Prädikatabfrage entschieden, ohne Eingreifen des Benutzers. Der Hauptentscheidungsträger bleibt aber der Benutzer, der den Gesamtablauf jederzeit mit Basisbefehlen beeinflussen kann. Der Rechner ist nur Werkzeug des Benutzers und trägt die Hauptlast der Berechnungen. Der Dialog gestaltet sich daher oft so, daß vom Rechner Vorschläge kommen, die der Benutzer bestätigt oder verwirft.

## 3.2 Graphische Oberflächen

In zunehmendem Maße werden Oberflächen durch graphische Hilfen unterstützt, die es dem Entwickler erlauben, graphische Symbole zu definieren und zu benützen. Diese üben innerhalb eines Systems meist eine Repräsentantenfunktion aus, indem sie bei der Auswahl durch einen Zeiger einen bestimmten Befehl ausführen.

Der Benutzer braucht dafür keine genauen Kontextkenntnisse und muß sich nicht um die korrekte Syntax des Befehls kümmern. Ein Beispiel aus der APRICOTS-Oberfläche ist die Auswahl eines bestimmten Steps mit der Maus. Nach dem Anklicken eines Steps erscheinen Detailinformationen, die der Benutzer sonst nur über eine Datenbankabfrage auf SQL-Systemebene erhalten könnte. Nachteilig wirkt sich eine solche Oberflächentechnik insofern aus, daß der Freiheitsgrad, unter dem die Bearbeitung erfolgt, sehr niedrig ist. Es können die Befehle nur so ausgeführt werden, wie sie vom Entwickler implementiert wurden. Wird vom Benutzer eine flexible Abarbeitung, in der er Befehle variieren kann, gewünscht, so muß er auf einer anderen Ebene zusätzlich weiterarbeiten. Beispielsweise kann innerhalb der graphischen Oberfläche Detailinformation über die einzelnen Steps abgefragt werden, indem der Benutzer einen mit der Maus ausgewählten Step[4] **anklickt**. Werden aber Informationen benötigt, die sich auf mehrere Steps beziehen, kann das nicht mehr innerhalb der Oberfläche geschehen. Der Benutzer muß sich in einem solchen Fall über eine interaktive Datenbankabfrage die gewünschte Information holen. Theoretisch möglich wäre zwar ein Auswählen mehrerer graphischer Objekte, über die dann gemeinsam eine Abfrage bearbeitet wird. Die Realisierung solcher komplexeren Strukturen wäre aber deutlich schwieriger zu implementieren und in Bezug auf Fehleranfälligkeit kritisch zu bewerten.

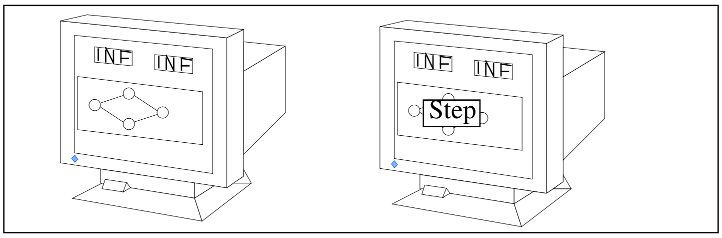
### 3.2.1 Fenstertechnik

Fenstertechniken stellen eine stärkere Orientierung an den Benutzer dar. Diese Technik wird hauptsächlich auf bitmap-orientierten[5] Bildschirmen verwendet und bietet viele Möglichkeiten, um Symbole und Schriften in unterschiedlicher Art und Größe darzustellen. Der eigentliche Bildschirm wird dabei in mehrere virtuelle Bildschirme eingeteilt, die als selbständig aufzufassen sind. Eine weitere Einteilung unter diesen in Fenster, die Aktionen tatsächlich zulassen und solche, die nur informellen Charakter besitzen, erleichtert dem Benutzer die Übersicht auf dem Gesamtbildschirm. Zur Erleichterung können zusätzlich Piktogramme[6] verwendet werden, die durch Auswahl mit einem Zeiger eine bestimmte Aktion starten können. Solche Kommandofunktionen sind ***Menuitems, Buttons*** und ähnliches.

Das Hauptfenster, also dasjenige, welches durch die Hierarchie anhand der Aufgabe als das Wichtigste klassifiziert wird, sollte daher auch den *besten* Platz auf dem Bildschirm haben. Wo der beste Platz auf dem Bildschirm ist, hängt von der Anzahl und Art der Fenster, die zum System gehören, ab und muß vom Entwickler bestimmt werden. Die graphische Oberfläche des APRICOTS plaziert das Hauptfenster, das den ConTract-Verlauf skizziert, am linken Rand in der Mitte des Bildschirms. So bleibt oberhalb dieses Fensters Platz für Informationsfenster, die dem Benutzer Hinweise über die Abarbeitung geben. Fordert der Benutzer Informationen über einen Step an, so erscheinen diese über dem Hauptfenster aufgesetzt und verschwinden nach Bestätigung wieder. Die Aufmerksamkeit des Benutzers ist für einen Moment auf einen Step beschränkt, was dazu führt, daß dieser an Wichtigkeit gewinnt und kuzzeitig im Vordergrund plaziert wird.

4. Beziehungsweise das graphische Objekt, das den Step repräsentiert.
5. Jeder Punkt auf dem Bildschirm kann einzeln angesprochen werden.
6. Diese werden vom System meist auch als Fenster betrachtet.

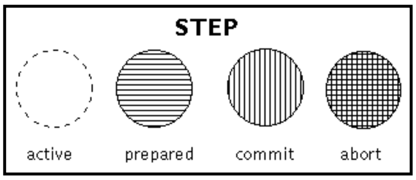
**ABBILDUNG 7** Positionierung innerhalb der APRICOTS-Oberfläche




### 3.2.2 Graphische Symbole

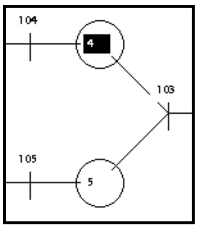
Graphische Symbole sind für den Benutzer eine Erleichterung bei der Orientierung innerhalb der Oberfläche. Sie haben durch ihren Wiedererkennbarkeitswert einen großen Einfluß auf das Verhalten und das Verständnis des Benutzers. Es gibt einige grundlegende graphische Gebilde, aus denen sich in der Regel die tatsächlich verwendeten zusammensetzen. Es handelt sich dabei um Linien, Kreise (auch Ellipsen), Vierecke und Schriftkästen. Dazu können Füllungen von variabler Schattierung dazu beitragen, daß *gleiche* Objekte sich unterscheiden lassen.

**ABBILDUNG 8** Unterscheidung gleicher Objekte durch unterschiedliche Schattierungen




Graphische Symbole sollten zum besseren Verständnis so gestaltet sein, daß mit wenig Kontextwissen ihre Funktionen leicht aus ihrem Aussehen abzuleiten sind. Was die metrische Gestalt (Länge, Breite, usw.) angeht, sollte das jeweilige Objekt dem ihm eigenen Abstraktionsgrad, der durch die Stellung im System vorgegeben ist, entsprechen. Objekte, die sich mit dem Zeiger ansprechen lassen, sollten sich nach Auswahl, optisch durch verschiedene Zustandskennungen unterscheiden lassen.

**ABBILDUNG 9** Unterscheidung vom Benutzer aktivierter und nicht-aktivierter Objekte




Diese Abbildung zeigt einen aktivierten Step und einen nicht aktivierten Step. Durch die Aktivierung erhält der Benutzer Detailinformationen über der Step.

## 3.3 Oberflächen im X Window-System

Das Konzept der ganzen Arbeit hat vorgesehen, daß die Implementierung unter einem System erfolgen soll, das sowohl im graphischen Bereich, wie auch unter Netzwerkaspekten am besten dafür geeignet ist.

### 3.3.1 Besonderheiten des X Window-Systems

Die Besonderheiten des X Window-Systems gliedern sich nach [Mans90] in vier Merkmalen:

- Netzwerktransparenz: Die Anwendung kann gleichzeitig innerhalb des Netzes auf mehreren Bildschirmen eine Ausgabe machen. Die Anwendung kann ohne weiteres auf einem Rechner laufen, der nicht die Ausgabe betreibt.
- Die ganze Bildschirmgestaltung wird dem Benutzer überlassen, das heißt dieser kann problemlos alle Fenster, deren Aussehen und Verwaltung bestimmen.
- Das X Window-System ist nicht Teil des Betriebssystems, sondern wird von diesem als Anwendungsprogramm verwaltet.
- Das X Window-System verfügt über einen hierarchischen Aufbau, der beim Implementieren von Anwendungen benutzt werden kann. So müssen keine neuen Verfahren erstellt werden.

Die eigentliche Anwendung (beispielsweise die graphische Oberfläche) hat in bezug auf des X Window-System die Rolle eines Clients. Der X-Server stellt diesem alle verfügbaren Resourcen zur Verfügung. Somit können alle im vorigen Unterkapitel beschriebenen Funktionen leicht implementiert werden.

### 3.3.2 Besonderheiten von Ingres Windows/4GL

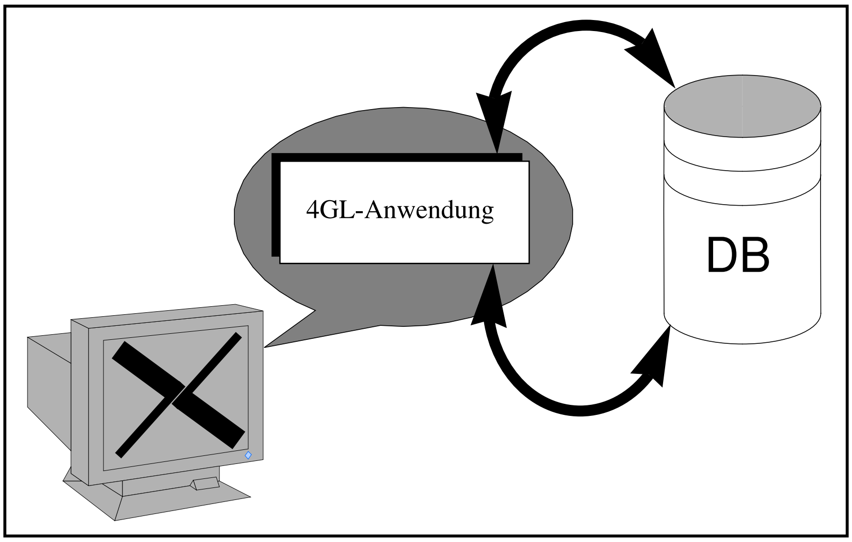
Zur Implementierung der Aufgabe wurde Windows/4GL verwendet. Es handelt sich dabei um ein Werkzeug, mit dem **ereignis-basierte** Anwendungen geschrieben werden können. Basierend auf einer Xlib-Implementierung bietet es den besonderen Vorteil an eine Datenbank gekoppelt zu sein. Das heißt zum einen, daß die Verwaltung aller Bestandteile einer Anwendung durch die Datenbank erfolgt und zum anderen, daß eine Abfrage an diese leichter gemacht wird. Da das Gesamtkonzept von APRICOTS Datenbankanwendungen massiv miteinbezieht, war diese Lösung so gesehen eine Ideallösung. Der Vorteil der Koppelung an die Datenbank besteht darin, ohne Zwischenschritte(Precompiler-Anweisungen) alle Operationen auf die Datenbank ausführen zu können. Es werden innerhalb der Implementierung alle SQL-Befehle genauso verwendet, wie sie bei einer interaktiven SQL-Abfrage auch verwendet würden.

Die Testumgebung des Systems ist objektorientiert aufgebaut, alle Komponenten können sofort nach Erstellen einzeln interaktiv getestet werden. Durch die Verwaltung der Anwendung durch die Datenbank, kann der Entwickler alle Bestandteile so implementieren, wie sie logisch in der Spezifikation konzipiert wurden. Es kann bei korrekter Vorgehensweise der logische Aufbau durch Zusammenfügen der einzelnen Bestandteile einer Anwendung sofort nachvollzogen werden. Die Diskrepanz der Spezifikation zum ausführbahren Programm wird vergleichsweise gering.

Es ergeben sich daher eine ganze Reihe von Merkmalen, die die Auswahl der verwendeten Software begründen:

1. Das Verhalten entspricht dem eines typischen X Window-Clients, mit allen Funktionalitäten und Hierarchieeigenschaften.
2. Optimale Anbindung, Verwaltung der Komponenten und der gesamten Anwendung durch die Datenbank.
3. Der modulare Aufbau ermöglicht eine flexible Entwicklung und Instandhaltung des erstellten Systems. Es können Hierarchien durch Schachtelung der Komponenten gebildet werden.

**ABBILDUNG 10** Eigenschaften einer Ingres Windows/4GL-Anwendung




## 3.4 Aspekte des Software Engineerings

Für heutige Systeme ist es wichtig, für jedes Projekt im Software-Bereich Betrachtungen zum Thema Software Engineering anzustellen. Dazu wird im allgemeinen der sogenannte Life-Cyle eines Produktes analysiert. Eine der gängigsten Gliederungen dazu [Somm89] wird im folgenden vorgestellt:

1. Erstellen der Spezifikation: Dabei muß eine klare Gliederung der Aufgabe in einer sowohl für den Benutzer, als auch für den Entwickler, verständlichen Art festgelegt werden.
2. System und Software Design: Es werden die zu verwendenten Hard- und Software-Bestandteile festgelegt. Dabei werden auf der einen Seite für die Aufgabe geeignete Rechner ausgesucht und auf der anderen Seite die Software, die verwendet werden soll, bestimmt.
3. Implementierung: Die Aufgabe wird programmatisch gelöst und alle erstellten, testbaren Einzelteile für sich getestet. Dafür ist es notwendig, daß jeder einzelne Baustein eine eigene Spezifikation erhält, um ein sinnvolles Testen zu ermöglichen.

4. Testen im System: Die erstellte Software wird innerhalb des Systems dem sie angehört integriert und getestet.
5. Einführung und Wartung: Die erstellte Software wird dauerhaft installiert und gewartet, das heißt nichtgefundene Fehler werden nachträglich entfernt und eventuell durch neue Anforderung benötigte Zusätze erweitert.

Die Spezifikation ergab sich bei der vorliegenden Arbeit aus der Aufgabenstellung und der gemeinsam durch Betreuer und Bearbeiter vereinbarten Detailanforderungen. Die Hardware ist durch die vorhandenen Möglichkeiten des Instituts beschränkt, wäre aber auch bei freier Auswahlmöglichkeit empfehlenswert. Als Programmierwerkzeug wurde eine unter objektorientiertem Ansatz und für Oberflächen in Zusammenhang mit Datenbankanwendungen besonders geeignete Software verwendet.

Die Implementierung erfolgte weitgehend modular, was unter objektorientiertem Ansatz eine geforderte Eigenschaft darstellt [Mey88]. Die Integration ins Gesamtsystem ist durch klar strukturierte Schnittstellen und durch die relative Eigenständigkeit der Oberfläche begünstigt. Sie ist stark abhängig von der Stabilität der Kommunikation des Gesamtsystems. Die Wartung ist durch die benutzerfreundliche Entwicklungsumgebung der verwendeten Software für einen eingearbeiteten Entwickler verhältnismäßig einfach. Dadurch daß die Berechnungen unabhängig von den graphischen Objekten sind, könnte eine Erweiterung auf andere graphische Objekte ohne Änderungen erfolgen. Ebenso könnte die Berechnung auf beliebige graphische Symbole erfolgen, wenn nur die Beziehungen unter den Objekten bekannt sind.

KAPITEL 4

# Betrachtungen aus graphentheoretischer Sicht

Die Theorie der Graphen, meist kurz Graphentheorie genannt, liefert ein mathematisches Modell für Systeme, die sich auf zweistellige Relationen abbilden lassen[Har74]. Zweistellige Relationen sind Paare der Form (a,b), wovon a und b Elemente aus einer Menge sind und die Anordnung eine Beziehung zwischen beiden darstellt. Eine Beziehung kann sich in vielerlei Formen ausdrücken, beispielsweise als Größer-Kleinerbeziehung oder wie im vorliegenden Problemfall, als Verbindung. Darüberhinaus gibt es viele Klassifikationen hinsichtlich Gestalt und Komplexität, von denen im folgenden einige im Zusammenhang mit der gestellten Aufgabe beschrieben werden.

## 4.1 Einige Definitionen

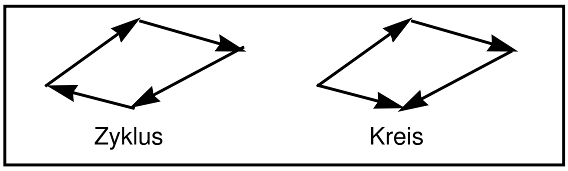
Um das Verständnis der im folgenden häufig wiederkehrenden Terminologien bezüglich Graphen zu erleichtern, werden hier einige Definitionen kurz aufgeführt[Jung87].

Ein Graph G ist ein Paar G = (V,E) aus einer endlichen Menge V <> Ø und einer Menge E von zweielementigen Teilmengen von V. Die Elemente von V heißen Knoten (auch Punkte oder Ecken genannt), die von E Kanten.

Ein gerichteter Graph ist ein Graph, dessen Teilmenge E aus *geordneten* Paaren besteht, das heißt ein Paar (a,b) ist nicht gleich (b,a); a,b ∈ V.

Ein Zyklus in einem gerichteten Graphen ist eine nicht triviale, geschlossene Kantenfolge, in der alle Ecken (außer der ersten und der letzten) verschieden sind [Har74]. Bei einem Kreis ist die Kantenfolge zwar geschlossen, aber nicht in einer Richtung gerichtet.

**ABBILDUNG 11** Unterscheidung zwischen Kreis und Zyklus




## 4.2 Die Beziehung zwischen Skript und Graph

Wesentlich für die Aufgabe war herauszufinden, welche Gestalt die graphische Abbildung eines ConTract-Skripts annehmen kann. Daraus kann anhand gängiger Sätze und Regeln die **Komplexität** der Berechnungen, die zur Darstellung notwendig sind, abgeleitet werden. Es wurde dabei von vornherein von einer eingeschränkten graphischen Struktur des resultierenden Gebildes ausgegangen, denn eine Darstellung, die einen maximalen Freiheitsgrad der graphischen Struktur erlauben würde, wäre wegen der vielen Möglichkeiten sehr schwierig zu implementieren. Mit einem maximalen Freiheitsgrad ist eine beliebige Verbindung der Knoten ohne jede Einschränkung gemeint.

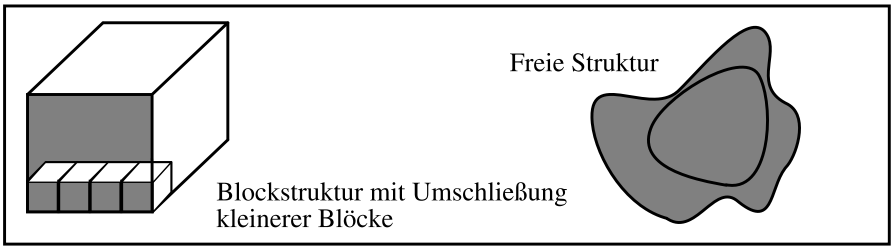
Bei einer Implementierung zur Lösung eines solchen Problems müßte diese maximale Anzahl der Möglichkeiten ohne Einschränkung berücksichtigt werden. Die Vorgehensweise richtete sich daher dahingehend, die Struktur soweit wie möglich einzuschränken. Dies hatte aus programmatischer Sicht die Folge, daß an vielen Stellen Sonderfälle klassifiziert werden mußten, um diese Einschränkungen verwerten zu können. Dieser vermeintlich hohe Preis wird aber mit dem Vorteil aufgewogen, aufwendige Schleifen soweit wie möglich zu vermeiden oder nur unter maximal quadratischer Komplexität auszuführen.

### 4.2.1 Die Blockstruktur des ConTracts

Die wichtigste Eigenschaft eines ConTracts aus graphentheoretischer Sicht ist dessen Struktur, sie ist für die Berechnung von zentraler Bedeutung. Gemeint ist die **Blockstruktur**.

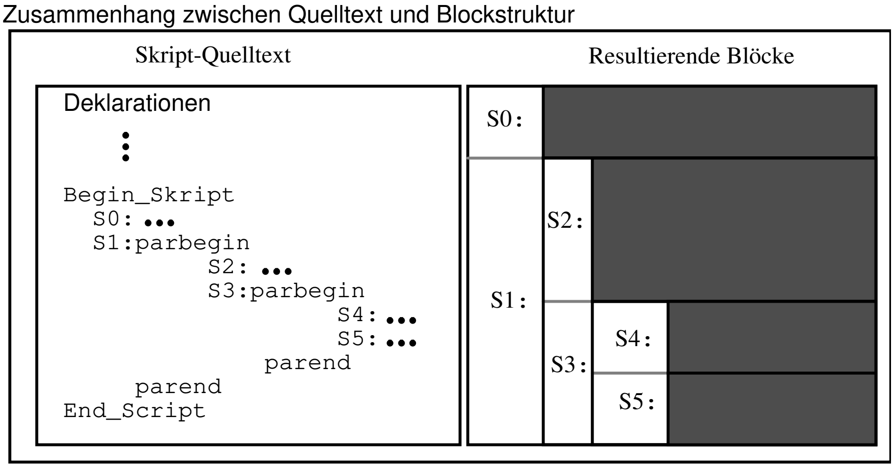
Unter einer Blockstruktur versteht man eine Struktur, die sich aus einem oder vielen Blöcken zusammensetzt. Diese Zusammensetzung entspricht einer Kapselung mehrere Blöcke, wodurch ein neuer übergeordneter Block entsteht. Ein Block kann sich dabei wieder aus Blöcken zusammensetzen oder aus einer Sequenz bestehen. Die einfachste Sequenz ist ein Step. Durch vollständige Dekomposition eines Skripts erhält man eine Menge von Steps. Betrachtet man diesen Zusammenhang von der anderen Seite, so ergibt sich nach Zusammensetzen der Steps eine hierarchische Beziehung dieser untereinander, beziehungsweise, es ergeben sich Regeln unter den resultierenden Blöcken. So ist auch das ConTract-Konzept aus vereinfachter allgemeiner Sicht zu verstehen. Die Events können dabei als Hilfskonstrukte zur Umrahmung der Blöcke aufgefaßt werden.

**ABBILDUNG 12** Blockstruktur und freie Struktur



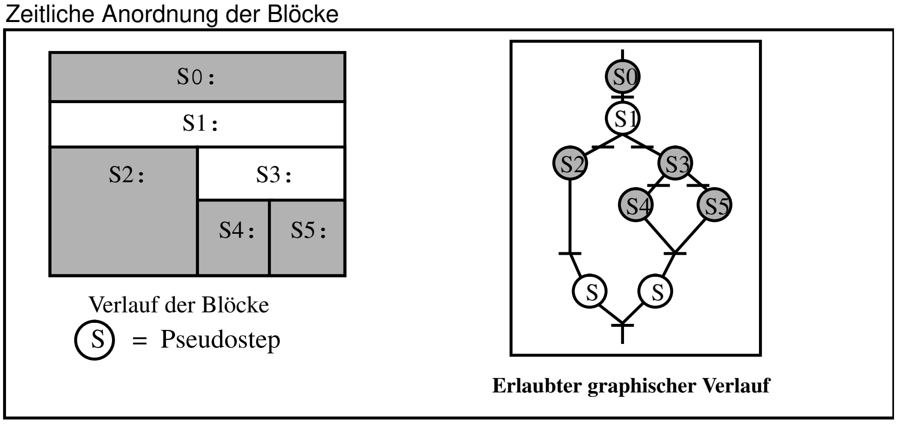

Um zu zeigen, daß nur solche graphischen Gebilde entstehen können, die sich wie einzelne Bausteine zusammenfügen lassen, muß ein Skript im Ganzen und in seinen Einzelteilen betrachtet werden. Da die Programmierung eines ConTracts durch die **prozedurale** Beschreibung des Skripts erfolgt, können Schlüsse bezüglich des Aufbaus durch diese konzeptionelle Einschränkung gezogen werden. Dazu ist es wichtig, den Aufbau eines Skripts zu betrachten, wie er durch die Programmierung entsteht.

**ABBILDUNG 13** Zusammenhang zwischen Quelltext und Blockstruktur




Die einzelnen Blöcke erscheinen so, wie sie hierarchisch strukturiert durch die verwendete Schachtelung angeordnet wurden. Betrachtet man jetzt den tatsächlichen zeitlichen Verlauf der Blöcke, entsteht ein etwas anderes Bild:

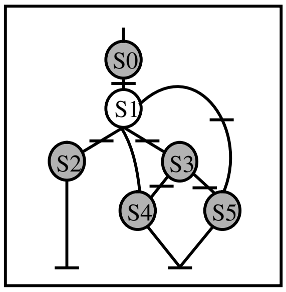
**ABBILDUNG 14** Zeitliche Anordnung der Blöcke



Was sich durch Betrachtung der obigen Abbildung vermuten läßt, ist, daß sich beim Verlauf, der nach jeder möglichen graphischer Anordnung der Blöcke entsteht, keine Überschneidungen auftreten. Eine solche Überschneidung könnte nur auftreten, wenn innerhalb der Anordnung im Skript ein Aufbrechen der syntaktischen Regeln, die die Blockbildung betreffen, erfolgen würde. Das hierarchische Einordnen in eine Blockstruktur, das durch Einrücken nach einem Kontrollkonstrukt implizit erfolgt, läßt solche mehrschichtigen Querverbindungen nicht zu. Drückt man diesen Sachverhalt durch eine graduelle Angabe, die die Einrücktiefe repräsentiert, aus, so ergibt sich für jeden Block eine eindeutige qualitative Zuordnung. Es ist nicht möglich, einem Block mehrere Zuordnungen in dieser Hinsicht zu erlauben

**TABELLE 3** Beispieltabelle der Zuordnung des erlaubten und des nicht erlaubten Graphen

| Steps | Zuordnung des erlaubten Graphen | Zuordnung des **nicht** erlaubten Graphen |
|---|---|---|
| S0 | 1 | 1 |
| S1 | 1 | 1 |
| S2 | 2 | 2 |
| S3 | 2 | 2 |
| S4 | 3 | 3,**2** |
| S5 | 3 | 3,**2** |



**Nicht erlaubter graphischer Verlauf**

Allgemein betrachtet kann davon ausgegangen werden, daß ein Programmiermodell wie das, das dem ConTract-Modell zugrunde liegt, nur Sinn macht, wenn Einschränkungen im Hinblick auf die Struktur erfolgen. Jede Einschränkung bringt auch einen Vorteil. Erst durch die Blockstruktur des ConTract-Modells ist es möglich, eine Vielzahl sinnvoller Funktionalitäten konzeptionell zu definieren. So wären keinerlei Recovery-Aktionen möglich, wenn diese sich nicht auf streng gekapselte Einheiten beziehen könnten. Ebenfalls wäre es schwer realisierbar, Kompensationen über einen nicht klar definierten Kontrollfluß auszuführen.

## 4.3 Graphentheoretische Klassifikation der Aufgabe

Aus der Sicht der Graphentheorie entspricht die graphische Abbildung eines ConTracts einem gerichteten, planaren Graphen mit Zyklen. Die zu bewältigende Aufgabe entspricht der, einen Graphen in die Ebene einzubetten.

### 4.3.1 Planare Graphen

Planare Graphen werden oft auch als plättbar oder als in die Ebene einbettbar bezeichnet. Sie wurden von Euler bei seinen Untersuchungen über Polyeder entdeckt.

> Definition: Ein Graph heißt *planar* oder *plättbar*, wenn er in die Ebene eingebettet werden **kann**.

> Definition: Ein Graph heißt in eine Fläche S *eingebettet*, wenn er so auf S gezeichnet **ist**, daß sich keine zwei Linien schneiden [Har74].

Wichtig für die vorliegende Arbeit ist folgender Satz:

**Satz:** Ein Graph ist genau dann planar, wenn jedes seiner Glieder planar ist [Har74].

Wichtig ist: Ein Graph kann planar sein, ohne in der Ebene eingebettet zu sein [Wag90].

Wie im vorigen Abschnitt beschrieben wurde, setzt sich ein ConTract (bedingt durch die Blockstruktur des Skripts) aus einzelnen Gliedern (Blöcken) zusammen. Diese Glieder können sich wiederum aus weiteren Gliedern zusammensetzten. Wendet man darauf den obigen Satz an, so muß, um die Planarität des Ablaufgraphen eines ConTracts zu beweisen, nur noch bewiesen werden, daß alle theoretisch möglichen Glieder, die sich nicht weiter aus anderen Gliedern aufbauen, planar sind. Anders formuliert bedeutet dies, das Resultat der graphischen Darstellung eines Blocks muß in jedem Fall einen planaren Graphen liefern. Dieser planare Graph muß dann im weiteren Verlauf eingebettet werden.

### 4.3.2 Planarität der Einzelteile

Bedingt durch die endliche Anzahl Kontrollkonstrukte (parbegin, if usw.), die im Skript die Blockstrukturierung bewirken, können auch nur eine endliche Anzahl graphischer Konstrukte daraus entstehen. Diese sollen hier näher untersucht werden. Um die Darstellung einheitlich zu gestalten, wird im folgenden jedes graphische Konstrukt immer durch einen Anfangs- und einen Endevent umrahmt. Dies geschieht in Analogie zum ConTract-Konzept, bei welchem der ConTract vom Anfangsevent 0 und vom Endevent 1 eingekapselt wird.

#### 4.3.2.1 Grammatikalische Darstellung eines ConTracts

**ConTract :== Event$^0$ Block Event PS Event$^1$**

**Block :== Simple_Seq | Conflict_Block | Seq_Block | Verz_Block | Par_Block**

**Simple_Seq :== Step | Block Event Block**

**Conflict_Block :== Step Event C_Step Event PS**

**Seq_Block :== BS Event Block Event ES**

**Verz_Block :== BV {Event Block Event PS} Event EV**

**Par_Block :== PB Event {PS Event Block} Event EP**

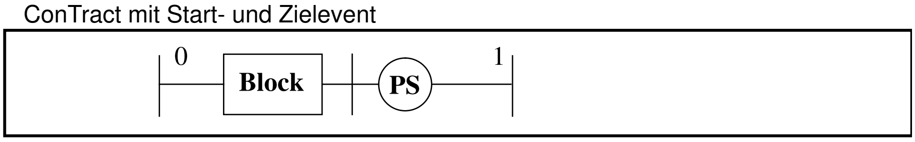
**Event, Step, PS, C_Step, BS, ES, BV, EV, PB, EP ∈ Terminale.**

**PS , C_Step, BS, ES, BV, EV, PB, EP sind Pseudosteps.**

#### 4.3.2.2 Der ConTract

Ein ConTract setzt sich aus dem Startevent 0, dem Zielevent 1, einem Block und einem den Block abschliessenden Pseudostep zusammen.

**ABBILDUNG 15** ConTract mit Start- und Zielevent

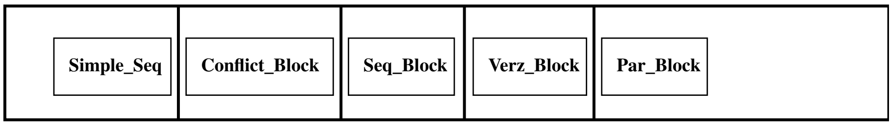


Es ist trivial, daß ein Graph, der nur aus einer Linie besteht, auf der mehrere Knoten angeordnet sind, planar ist.

#### 4.3.2.3 Der Block

Ein Block ist ein Überbegriff für spezielle Blöcke. Er umfaßt folgende fünf Blöcke, die alle noch einzeln behandelt werden.

**ABBILDUNG 16** Der Block

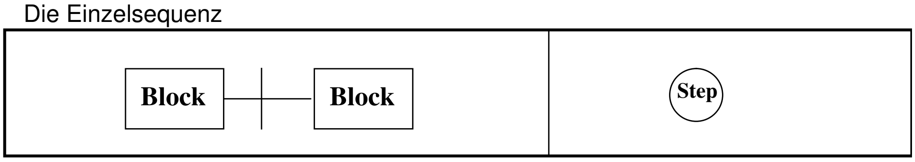



Ein Block kann, durch die oben getroffene syntaktische Einordnung, graphentheoretisch als Knoten aufgefaßt werden. Ein einzelner Knoten ist immer planar, deshalb auch ein Block.

#### 4.3.2.4 Die Einzelsequenz

Die kleinste Einheit, die innerhalb des Skripts als Block erzeugt werden kann, ist ein einzelner Step. Dieser wird im Skript sequentiell aufgeführt.

**ABBILDUNG 17** Die Einzelsequenz

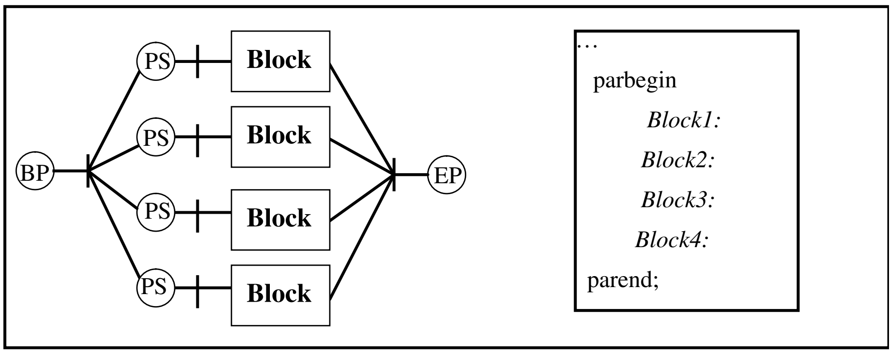



Es ist trivial, daß ein Graph, dessen zwei Kanten vom einem Knoten (Event) ausgehen, eingebettet darzustellen ist. Ein einzelner Knoten ist immer planar.

#### 4.3.2.5 Der Parallelblock

Ein Block, der parallele Ausführungen erlaubt, setzt sich aus mehreren Blöcken, die parallel ausgeführt werden können, zusammen. Am Anfang steht ein Begin_Par-Step gefolgt von einem Event, der wiederum die am Beginn jedes parallelen Zweiges stehenden Pseudo-Steps triggert.

**ABBILDUNG 18** Der Parallelblock



Die so entstehenden Flächen innerhalb des ganzen Blocks, sind immer durch acht Kanten begrenzt. Diese Regelmäßigkeiten im Aufbau, kann man sich zum Beweis der Planarität, zunutze machen.

Zum allgemeinen Beweis, daß ein solches Gebilde immer über Planarität verfügt, kann hier ein Korollar, das aus der Eulerschen Polyederformel abgeleitet wird, verwendet werden. Die Eulersche Polyederformel besagt, daß ein sphärischer Polyeder mit V Ecken, E Kanten und F Flächen (Fläche = vollständig von Kanten eingeschlossener Bereich) folgender Formel genügt[Har74]:

$$V - E + F = 2$$

Die hieraus abgeleitete Formel über planare Graphen, ausgegangen von p Ecken, q Kanten und inneren Gebieten, die jeweils ein n-Eck darstellen, also Kreise der Länge n bilden, besagt:

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

Liefert die Gleichung unter Benutzung der gegebenen Parametern ein korrektes Ergebnis, so ist die Planarität des Parallelblocks bewiesen:

q = 8,
p = 8,
n = 8:

$$8 = 8\left(\frac{8-2}{8-2}\right)$$

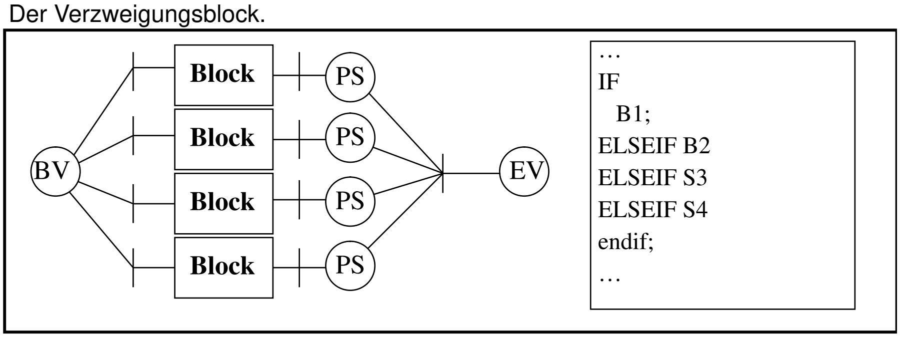
q.e.d.

#### 4.3.2.6 Der Verzweigungsblock

Blöcke, die eine bedingte Verzweigung darstellen, sind graphisch abgebildet den Parallelblöcken sehr ähnlich. Der Unterschied liegt allein darin, daß Parallelblöcke die Verzweigung über einen Event einleiten, der einem Pseudostep folgt, und Blöcke bedingter Verzweigung in einem Step beginnen. Für das Aussehen und deshalb auch für den graphischen Aufbau hat dieser Umstand nur wenig Bedeutung, bei der Steuerung des Kontrollflusses dagegen, zeigt sich dann der Unterschied:

- Beim Parallelblock werden alle Verzweigungen aktiviert.
- Beim Verzweigungsblock werden nur die Kanten aktiviert, die durch Abprüfen des Ausgangsprädikats qualifiziert werden, siehe Seite 7 ff

**ABBILDUNG 19** Der Verzweigungsblock.



Der Beweis der Planarität verläuft analog zu dem unter 4.3.2.5., nur daß hier zehn Kanten und zehn Ecken auftreten:

$$q = n\frac{(p-2)}{(n-2)} \qquad 1$$

Liefert die Gleichung unter Benutzung der gegebenen Parameter ein korrektes Ergebnis, so ist die Planarität des Verzweigungsblocks bewiesen:

q = 10,
p = 10,
n = 10:

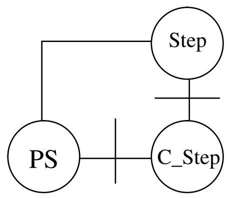
$$10 = 10\left(\frac{10-2}{10-2}\right) \quad \text{q.e.d.}$$

Graphentheoretisch macht es keinen Unterschied, ob eine Ecke einen Step darstellt oder einen Event. Bei der Berechnung des Ablaufgraphen wird deshalb auch so vorgegangen, das heißt, die Berechnung der Positionierung der Objekte Event und Step verlaufen völlig analog.

#### 4.3.2.7 Die Konfliktschleife

Eine Ausnahme, die streng genommen nicht als Block aufzufassen ist, ist die sogenannte Konfliktschleife *(conflict resolution)*. Sie ist deshalb nicht exakt als Block zu bezeichnen, weil sie nicht exakt abgegrenzt ist. Der Step, der Ausgangspunkt für die Konfliktschleife ist, gehört zusätzlich zu einem anderen Block. Die Konfliktschleife erscheint nur an Stellen, an denen die sogenannte **Eingangsinvariante** eines **Steps** fehlgelaufen ist

**ABBILDUNG 20** Die Konfliktschleife




Konfliktschleife

```
…
S5 : …
ENTRY_INVARIANT …
…
BEGIN_CONFLICT_RESOLUTION
 S5 : BEGIN
       S100 : …
      END
END_CONFLICT_RESOLUTION
…
```

Es liegt hier eine weitere Möglichkeit zur Verzweigung vor. Anders als beim trigger-event-Mechanismus, wo nach Abschluß eines Steps ein Prädikatabfrage durchgeführt wird, wird hier ein Step nicht ausgeführt. Erst nach erfolgreichem Abschluß der Konfliktschleife und Durchlaufen der Eingangsinvariante des Steps, kann dieser ausgeführt werden. Betrachtet man eine Konfliktschleife als Block, so entspricht der innere Verlauf einem einfachen Sechseck, das immer planar abbildbar ist.

### 4.3.3 Planaritätsbeweis des Gesamtblocks

Nachdem der Beweis der Planarität der Einzelteile erbracht ist, muß noch der allgemeine Beweis erfolgen. Dies geschieht mittels vollständiger Induktion über die Tiefe n der Blöcke. Innerhalb des Graphen können Events und Steps als Knoten unterschieden werden, aus diesen setzen sich die oben beschriebenen Blöcke zusammen.

1. Induktionsanfang: Die Knoten des Graphen sind planar. Das ist trivial.
2. Induktionsannahme: Es seien Blöcke der Tiefe n planar.

   Induktionsparamter: Die Tiefe soll dabei so definiert sein, daß (ausgegangen von den Steps mit Tiefe 1) die Tiefe der übergeordneten Blöcke nach der Blockbildung inkrementiert wird. Beispielsweise:

   Tiefe (parbegin S1 S2 S3 … Sn parend)
   hat Tiefe = max{Tiefe (S1,S2…Sn)}+1.
3. Induktionsschritt: Seien Blöcke der Tiefe n planar, dann auch Blöcke der Tiefe n+1, die durch Anordnung mit den erlaubten Konstruktionen (if, parbegin…) entstehen, was oben im einzelnen gezeigt wurde.

## 4.4 Komplexitätsabschätzung

Hat man die Planarität eines Graphen bewiesen, so ist damit noch nicht gesichert, daß dieser auch **eingebettet** darstellbar ist. Die formale Darstellung eines Graphen ist abhängig von der gewählten Datenstruktur. Zu Beschreibungen von Problemen und Algorithmen wird die unter Bereich 4.1 auf Seite 23 aufgeführte Darstellung am häufigsten verwendet. Welche Datenstrukturen bei der Implementierung gewählt werden, hängt stark von den verwendeten Algorithmen ab. Die Vorgehensweise verläuft so: Erbringung des Nachweises der Planarität, daraufhin muß der Graph noch in die Ebene eingebettet werden.

### 4.4.1 Die st-Numerierung

Zur Prüfung der Planarität kann zuerst die sogenannte **st-Numerierung** verwendet werden. Dieses allgemeine Verfahren ermittelt durch eine Tiefensuche für jeden Knoten, abhängig vom Vaterknoten, einen Index und errechnet dadurch die st-Numerierung jedes Knotens. Gibt es eine gültige st-Numerierung, dann ist der Graph planar und kann anhand der Numerierung eingebettet werden, sonst ist er nicht planar. Die st-Numerierung kann jedoch bewirken, daß an den Außenseiten gebogene Verbindungen sind.

Sei ein beliebiger Graph G(V,E) mit Ecken V und Kanten E gegeben. Er sei durch Adjazenz-Listen repräsentiert, das heißt durch Paare $(v,w) \in E$ und $v,w \in V$. Die Liste der Nachbarknoten erscheint jeweils im Uhrzeigersinn, so wie sie momentan angeordnet sind. Eine Numerierung wird st-Numerierung genannt, wenn zwei Ecken 1 und n adjazent sind und jedes j aller anderen Ecken adjazent zu zwei weiteren Ecken i und k $(i < j < k)$ ist . Es wird eine Kante ausgewählt, die eine beliebige Ecke 1 mit einer beliebigen Ecke 2 verbindet. Die Ecke 1 wird mit s, die Ecke n wird mit t bezeichnet.

**ABBILDUNG 21** Die Prozedur zur st-Numerierung

*procedure* st_number(G);

*begin*
markiere s,t und (s,t); alle anderen Knoten und Kanten sind unmarkiert.
lege t und s auf den Stapel; Zähler = 1;
nehme oberstes Element v vom Stapel;
*while* v ≠ t *do*
*begin*
*if* Weg(v) = ∅ *then*
*begin*
st_num(v) := Zähler; Zahler = Zähler + 1;
*end*;
*else*
*begin*
*if* Weg(v) = $vu_1u_2\ldots u_kw$;
lege $u_k,u_{k-1},\ldots,u_1$,v auf den Stapel {in dieser Reihenfolge, v liegt oben}
*end*;
nehme oberstes Element vom Stapel;
*endwhile*;
*end*.

Dazu gibt es eine Funktion Weg:
*function* Weg(v: Knoten): {[Knoten]};
*begin*
*case* 1: es existiert eine neue Rückwärtkante (v,w);
markiere (v,w); Weg = vw;
*case* 2: es existiert eine neue Vorwärtskante (v,w) mit $ww_1w_2\ldots w_k$ , wobei $w_k$ eine Rückwärtskante zu einem Knoten u hat.
markiere $ww_1w_2\ldots w_k$; Weg = $vww_1w_2\ldots w_k$;
*case* 3: es existiert eine neue Rückwärtskante (w,v) mit $ww_1w_2\ldots w_k$ wobei $w_k$ eine Rückwärtskante zu einem bereits markierten Knoten u hat;
markiere $ww_1w_2\ldots w_k$; Weg = $vww_1w_2\ldots w_k$;
*case* 4: Weg = ∅;
*end;*

Mit diesem Algorithmus (siehe auch Näheres in [NisChi88]) kann in linearer Zeit eine Darstellung erreicht werden, die es ermöglicht, einen mindestens zweifach-zusammenhängenden[1] Graphen anhand der st-Numerierung planar darzustellen. Zweifach zusammenhängend bedeutet, daß mindestens zwei Kanten entfernt werden müssen, um den Graphen in zwei Teile zu zerlegen. Bei einfach zusammenhängenden Graphen reicht es aus, eine Verbindung zu entfernen, um zwei Teilgraphen zu erhalten. Der graphische Verlauf eines ConTracts ist einfach zusammenhängend, weshalb die st-Numerierung nicht ausreicht, um eine geeignete Darstellung für das gestellte Problem zu finden.

1. Ein ConTract ist nicht zweifach-zusammenhängend.

Weiterhin einschränkend ist die Tatsache, daß bei Verwendung der st-Numerierung unter Umständen *krumme* Kanten am Rande verwendet werden müssen. Krumme Kanten sind gebogene Verbindungen, die zwar ohne eine andere Kante zu schneiden eine Verbindung zwischen zwei Knoten herstellen, aber eben keinen geraden Verlauf im Sinne einer Streckenverbindung darstellen. Zur sicheren korrekten Darstellung werden aufwendigere, sogenannte Graphlayout-Algorithmen verwendet.

### 4.4.2 Graphlayout-Algorithmen

Ein Graph wird **Streckengraph** genannt, wenn er eine Einbettung besitzt [Wag90]. Ein Streckengraph ist ein konvexer Streckengraph, wenn der Randkreis eines jeden Gebietes ein konvexes Polygon in der Ebene ist. Mit anderen Worten bedeutet das, daß jede Verbindung zwischen zwei Knoten, die Länge der Distanz beider Knoten hat und sich keine zwei Verbindungen schneiden. Ein ConTract-Ablaufgraph kann unter Berücksichtigung des vorher ausgeführten Planaritätsbeweises als konvexer Streckengraph bezeichnet werden.

Es muß nach einer speziellen Strategie vorgegangen werden, wenn ein Graph mit einheitlichen Verbindungslinien dargestellt werden soll. Einheitlich bedeutet dabei, daß bei Verzweigungen diese die gleiche Länge besitzen und der Winkel der Verzweigung symmetrisch zu einer Mittelachse verläuft. Durch eine solche Strategie wird die Tiefe (X-Koordinate) des Knotens mit einer Tiefensuche und daraufhin die Breite (Y-Koordinate) des Knotens durch Miteinbeziehung des Verzweigungsgrades errechnet. Dabei können Doppelbelegungen eines Koordinatenpunktes auftreten. Um dieses zu verhindern, muß nach jedem Schritt eine Prüfung und eventuell eine Korrektur durchgeführt werden.

**ABBILDUNG 22** Die Berechnung der Koordinaten der Knotenpunkte

```
for i = 1 to n do
  errechne Tiefe;
endfor;
for i = 1 to n do
  errechne Breite;
  for j = 1 to n do
    überpfüfe ob Platz schon belegt ist.;
    if belegt then
      korrigiere;
    endif
  endfor;
endfor:
```

Die Berechnungen, die in der vorliegenden Arbeit zur Lösung des Problems anfallen, werden aufgrund des Detailwissen über das System und durch die geeignete Wahl der Datenstrukturen nie höher als von quadratischer Komplexität sein.

KAPITEL 5

# Aufbau des Graphen innerhalb der Anwendung

Ist das Einarbeiten in APRICOTS erfolgt und der theoretische Hintergrund über Graphen geschaffen, so kann mit der *eigentlichen* Arbeit, das heißt dem Erstellen der Anwendung begonnen werden. Dabei ist ein schrittweise, abgestufter Plan der Bearbeitung sehr hilfreich.

Der Aufbau des Ablaufgraphen erfolgt in drei Schritten:

1. Einlesen der in der Datenbank relevanten Daten über Events und Steps.
2. Verarbeiten und Umwandeln dieser Daten in Datenstrukturen, wie sie für den Aufbau gebraucht werden. Schaffung von Objekten, auf die im weiteren Verlauf Methoden angewandt werden.
3. Anwenden der Algorithmen, auf die geschaffenen Datenstrukturen, zum Aufbau des Graphen aus den Objekten.

Bei der Implementierung wurde deshalb in entsprechender Weise vorgegangen. Dazu mußte zuerst die APRICOTS-Datenbank im Hinblick auf die für den Ablaufgraphen relevanten Datenbestände überprüft werden.

## 5.1 Die Relationen der Datenbank APRICOTS

Innerhalb der Datenbank sind nicht alle Relationen für den Aufbau des Ablaufgraphen relevant. Die Datenbank enthält Relationen, die Daten mit **dynamischem** Charakter enthalten. Das sind solche, die während eines ConTract-Ablaufs abgelegt und danach wieder gelöscht werden. Es handelt sich dabei um Informationen, die den momentanen Ablauf mitprotokollieren, sie dienen in der Regel zu Recovery-Zwecken. Die **statischen** Daten werden normalerweise dauerhaft gespeichert, da es sich um Daten handelt, die Abhängigkeiten unter den Ausführeinheiten repräsentieren. Für den Ablaufgraphen sind nur die statischen Daten von Bedeutung und es werden im folgenden nur diese besprochen. Sie sind bei jeder ConTract-Ausführung, sofern es sich um den gleichen ConTract handelt, gleich.

Ändern sich die statischen Daten eines ConTracts, so kann dies eine Änderung des Ablaufs nachsichziehen. Eine Änderung des Ablaufs wiederum kann eine Änderung des Ablaufgraphen bewirken, was sich selbstverständlich im optischen Erscheinungsbild auswirkt.

**TABELLE 4** Die Relation **Start_Events**

| Contract Name | Step Id | Start Event |
|---|---|---|
| Reisebuchung | 1 | 0 |
| Reisebuchung | 2 | 100 |
| Reisebuchung | 3 | 101 |
| Reisebuchung | 4 | 104 |
| Reisebuchung | 5 | 105 |
| ⋮ | ⋮ | ⋮ |

Aus dieser Relation lassen sich die eindeutigen Vorgänger-Events jedes einzelnen Steps ablesen. Weil jeder Step nur einen einzigen Vorgänger besitzt, kann dieser als einfacher Parameter in der internen Datenstruktur repräsentiert werden und braucht nicht in Form einer Liste an die interne Stepstruktur gehängt werden. Es lassen sich außerdem für jeden Event die Nachfolgersteps ermitteln und somit auch die Anzahl der Nachfolger, was bei einer Verzweigungssituation wichtig ist. Diese Nachfolger werden in eine Liste übernommen.

**TABELLE 5** Die Relation **Termination_Events**

| Contract Name | Step Id | Condition | Termination Event |
|---|---|---|---|
| Reisebuchung | 1 | TRUE | 100 |
| Reisebuchung | 2 | TRUE | 101 |
| Reisebuchung | 3 | TRUE | 102 |
| Reisebuchung | 7 | (kombinationsnr : LONG <> 0) | 109 |
| Reisebuchung | 7 | ELSE | 112 |
| ⋮ | ⋮ | ⋮ | ⋮ |

Aus der Relation Termination_Events lassen sich die direkten Nachfolgerevents jedes Steps und somit auch die Anzahl der nachfolgenden Verzweigungen ablesen. Ebenso können für alle Events die Vorgänger und deren Anzahl festgestellt werden. Indem durch eine Selektion der Steps, sortiert nach Termination_Events, für jeden Event die Vorgänger bestimmt werden. Daraus ergibt sich die Anzahl der hereinkommenden Verzweigungen für die Events.Sowohl die Vorgängersteps, wie auch die Nachfolgerevents, werden in Listen geführt und jeweils den internen Strukturen angehängt.

**TABELLE 6** Die Relation **Steps**

| Contract Name | Step Id | Step Name | Flag | Nesting Level | Entry Inv | Exit Inv | Policy | Sched Hint |
|---|---|---|---|---|---|---|---|---|
| Reisebuchung | 1 | Flugdateneingabe | 0 | 0 | | select… | CHECK_REVALIDATE | |
| Reisebuchung | 2 | Flugverbindung_ermitteln | 0 | 0 | | | | |
| Reisebuchung | 3 | PARBEGIN | 1 | 1 | | | | |
| Reisebuchung | 4 | Flugbuchung | 0 | 1 | | | | |
| Reisebuchung | 5 | Flugkosten_abbuchen | 0 | 0 | select count ( * ) from kunde where budget >= : limit and : limit >= : betrag_flug and name = : abtname | select budget from kunde where name = : abtname and budget >= : limit - : betrag_flug | CHECK_REVALIDATE | |
| Reisebuchung | 8 | Hotelbuchung_vornehmen | 0 | 1 | select count ( * ) from kunde where name = : abtname and budget >= : betrag_hotel | select budget from kunde where name = : abtname and budget >= : limit - ( : betrag_flug + : betrag_hotel ) | CHECK_REVALIDATE | |
| Reisebuchung | 11 | Buchungsdaten_ausgeben | 0 | | | | | |
| ⋮ | ⋮ | *⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

Aus der Relation Steps lassen sich Detailinformationen über die einzelnen Steps ablesen. So kann aus Step_Name abgeleitet werden, welche semantische Funktion der jeweilige Step hat, das heißt was passiert während des Ablaufs dieses Steps. Aus der Spalte Entry_Inv lassen sich, bei einem vorhandenen Eintrag, Schlüsse auf eine mögliche Konfliktbehandlung ziehen. Das ist wichtig, weil diese Art der Verzweigung graphisch anders als die üblichen Verzweigungen behandelt werden. *Übliche* Verzweigungen sind solche, die nach einer bedingten Abfrage oder nach einer parallelen Triggerung mehrerer Steps, einen Knotenpunkt verlassen. Siehe “Der Parallelblock” auf Seite 28.

Bei üblichen Verzweigungen wird die horizontale Tiefe (der Ablaufgraph ist von links nach rechts orientiert) immer vergrößert, das heißt nach rechts ausgeweitet. Hier erscheint die Verzweigung sozusagen auf gleicher Höhe, da es für den Ablauf auch keinen *Fortschritt* darstellt, wenn eine Konfliktbehandlung eingeleitet wird. Sondern nur ein Versuch einen nicht ausführbaren Step ausführbar zu machen, in dem die nötigen Voraussetzungen dafür geschaffen werden. Dies geschieht, durch Ausführen eines vom ConTract-Programmierer erstellten *Behebungssteps*. Es werden damit die von der Eintrittsinvariante verlangten Voraussetzungen geschaffen.

## 5.2 Die interne Datenstrukturen

Die Datenstrukturen, die innerhalb der Anwendung benötigt werden, lassen sich unter zwei Gesichtspunkten gruppieren:

1. Es werden Strukturen benötigt, die konkret für jeden Step und für jeden Event eine Repräsentantenfunktion übernehmen können. Diese Strukturen werden in Form von numerierten Listen angeordnet und können als Objektlisten bezeichnet werden. Sie sind aus darstellerischer Sicht besonders wichtig, da damit über die Step- beziehungsweise Eventnummern ein graphisches Objekt direkt angesprochen werden kann. Beispielsweise muß bei einer Zustandsänderung nur die Nummer des Steps und die Art der Änderung angegeben werden, um diese Änderung auch sichtbar zu machen.
2. Es werden Strukturen benötigt, die die Berechnungen aus graphentheoretischer Sicht begünstigen. Es wurden bei dieser Prototyp-Implementierung Matrixstrukturen verwendet, da sie die Möglichkeit bieten, direkt die Abhängigkeit zweier Knotenpunkte durch Angabe der Indizes abzufragen.

### 5.2.1 Die Objektlisten

Ingres Windows/4GL erlaubt es dem Entwickler Strukturen zu definieren, die sich wiederum aus anderen zusammensetzen. Diese Möglichkeit wurde benutzt, indem graphische Gebilde (Kreise, Linien und Textfelder) und die Vor- und Nachfolger eines jeden als Objekt klassifizierten Steps beziehungsweise Events in einem Verbund zusammengefaßt wurden. Für die Steps und die Events ergeben sich die folgende, abgebildeten Verbunde.

**ABBILDUNG 23** Der Aufbau der internen Struktur für die Steps

| Attribute Name | Type | Data Type/Class Name |
|---|---|---|
| step_id | SIMPLE | smallint |
| start_event | SIMPLE | smallint |
| status | SIMPLE | smallint |
| kreis | REFERENCE | ellipseshape |
| step_name | SIMPLE | varchar(255) |
| flag | SIMPLE | smallint |
| nesting_level | SIMPLE | smallint |
| entryname | REFERENCE | entryfield |
| anz_nach | SIMPLE | integer |
| sonder | SIMPLE | smallint |
| list_id | SIMPLE | varchar(5) |
| eventnach | ARRAY | nachliste |

Steps

**ABBILDUNG 24** Der Aufbau der internen Struktur für die Events

| Attribute Name | Type | Data Type/Class Name |
|---|---|---|
| event_id | SIMPLE | smallint |
| expected_tokens | SIMPLE | smallint |
| trigger_state | SIMPLE | smallint |
| balken | REFERENCE | segmentshape |
| anz_nach | SIMPLE | integer |
| list_id | SIMPLE | varchar(5) |
| stepnach | ARRAY | nachliste |
| entryname | REFERENCE | entryfield |
| anz_vor | SIMPLE | integer |

Events

Die obigen Abbildungen zeigen die Strukturen, wie sie in die Listen aufgenommen werden. Mit anderen Worten bedeutet dies, daß, wenn man über einen Index auf die Listen zugreift, eine Struktur zurückerhält, die von einer der oben aufgeführten Gestalt ist. Innerhalb des Ablaufs werden die Listen am **Anfang** der Berechnung durch die Information aus der Datenbank aufgefüllt. Sind die benötigten Informationen nicht in der gewünschten Form in der Datenbank vorhanden, so lassen sie sich, wie unter Tabelle 4 bis Tabelle 6 beschriebenen Verfahren, einfach berechnen. Während des weiteren Ablaufs werden weitere Anfragen an die Datenbank nicht mehr benötigt[1]. Somit sind die Berechnungen im weiteren Verlauf nur noch rein intern, das heißt, es werden nur noch innerhalb der Programmierumgebung bereitgestellten Funktionen verwendet. Dies hat den Vorteil, daß die Datenbank[2] nicht länger als unbedingt notwendig blockiert wird, da es sich bei erneuten Anfragen ohnehin um die gleichen, wie sie vorher schon verwendet wurden, handeln würde.

1. Mit einer Ausnahme, die im folgenden noch besprochen wird.
2. beziehungsweise die verwendeten Relationen.

### 5.2.2 Die Struktur der Adjazenzmatrix

Für die Berechnungen des Ablaufgraphen von zentraler Bedeutung ist die Adjazenzmatrix. Die einfachste Form einer Adjazenzmatrix besteht aus einer n x n - Matrix, das heißt, sie ist symmetrisch. An jeder Stelle (i,j), das heißt in der i-ten Zeile und der j-ten Spalte, wo sie einen Eintrag hat, lassen sich Rückschlüsse auf die Nachfolger (beziehungsweise die Vorgänger) machen. Beispielsweise bedeutet ein Eintrag an der Stelle (2,3), daß der zweite Knoten ein Vorgänger (beziehungsweise Nachfolger) vom dritten Knoten ist. Ob es sich bei einem Eintrag um den Vorgänger oder den Nachfolger handelt ist implementierungsabhängig.

Die Programmierung unter Ingres Windows/4GL läßt keine vereinfachten Deklarationen, wie sie beispielsweise in Pascal oder C möglich sind, zu. Es ist nicht möglich eine Deklaration folgender Gestalt zu machen:

```
matrix : array[1..n,1..m] of integer;
```

Um trotzdem eine matrixförmige Struktur zu definieren, werden Listen generiert, die über Attribute verfügen, wovon eines wieder eine Liste darstellt. Es entsteht folgende Struktur:

**ABBILDUNG 25** Der Aufbau der Adjazenzmatrix

| Attribute Name | Type | Data Type/Class Name |
|---|---|---|
| zeile | ARRAY | spalte |
| zve_id | SIMPLE | varchar(5) |
| ebene | SIMPLE | smallint |
| zflag | SIMPLE | smallint |
| hoehe | SIMPLE | smallint |
| vorz | SIMPLE | smallint |
| nachz | SIMPLE | smallint |
| level | SIMPLE | smallint |

Der sich dabei ergebende Unterschied ist für die Programmierung nur syntaktisch von Bedeutung. Die Möglichkeit, über zwei Indizes direkt auf eine Zelle der Matrix zuzugreifen, bleibt erhalten. Dieser Umstand ist für die Implementierung wichtig, da dadurch ein Durchsuchen verketteter Listen entfällt.

**ABBILDUNG 26** Der Aufbau von spalte

| Attribute Name | Type | Data Type/Class Name |
|---|---|---|
| flag | SIMPLE | smallint |
| sve_id | SIMPLE | varchar(5) |

Will man mit zwei Indizes i und j direkt auf eine Matrixzelle zugreifen, so wäre folgende Pascal-Zuweisung

```
n := matrixvar[i,j].flag;
```

äquivalent zu folgender Windows/4GL-Zuweisung:

```
n := matrixvar[i].spalte[j].flag;
```

### 5.2.3 Die Hilfsrelation

Während der Initialisierung der eingeführten Datenstrukturen durch die Informationen aus den statischen Daten, werden zusätzlich alle die Knotenpunkte, die zu irgendeiner Konfliktschleife gehören, in einer Relation gespeichert. Das sind sowohl Steps wie auch Events. Diese Relation hat nur den Zweck die Namen die an der Konfliktschleife beteiligten Knoten bereitzustellen. Es sind dies die Namen **mit** einem Anfangsbuchstaben ‘s’ oder ‘e’. Innerhalb der statischen Daten werden nur Nummern als Kennzeichnung geführt. Dabei sind die Steps und die Events immer getrennt. Vermischt man aber die Nummern der Events mit denen der Steps, so können Doppelbelegungen auftauchen.

Das ist der Grund dafür, warum alle Objekte einen Anfangsbuchstaben vor die Nummer gestellt bekommen, um sie nach dem Vermischen, das heißt nach dem jeder Step und jeder Event als graphisches Objekt klassifiziert wurde, auseinanderhalten zu können. Intern wird nach der Initialisierung eine weitere Liste mit den Namen der an den Konfliktschleifen beteiligten Knoten erstellt. Diese wird im weiteren Verlauf, das heißt, beim Durchlaufen der Algorithmen, verwendet.

**TABELLE 7** Der Aufbau der Hislfrelation für die Konfliktknoten

| P Id | I Id | P Id | I Id | P Id | I Id |
|---|---|---|---|---|---|
| e116 | 116 | e121 | 121 | s100 | 100 |
| e117 | 117 | e2 | 2 | s101 | 101 |
| e118 | 118 | s-10 | -10 | s102 | 102 |
| e119 | 119 | s-9 | -9 | | |
| e120 | 120 | s-8 | -8 | | |

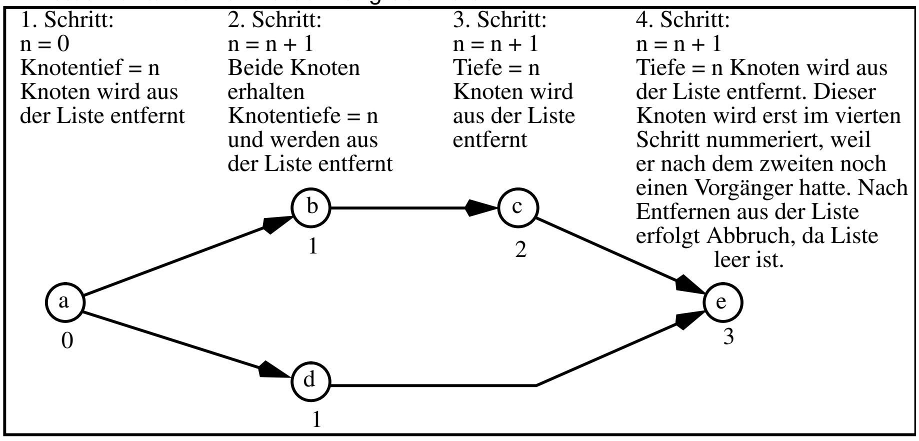
## 5.3 Die Berechnungen zur Tiefe der Knoten

Die Tiefe eines Knotens ist gekennzeichnet durch die Anzahl Schritte, die ein Knoten vom Startknoten entfernt ist. Dabei ergeben sich Schwierigkeiten, wenn ein Knoten auf verschiedenen Wegen, die unterschiedlich lang sind, erreicht werden kann. Für die Plazierung der Objekte auf der Fensterfläche ist dabei die maximale Tiefe aller direkten Vorgängerknoten wichtig. Beispielsweise muß ein Knoten die Tiefe $n + 2$ erhalten, wenn der weiteste Weg zu einem direkten Vorgänger $n + 1$ beträgt. Dabei darf jeder Knoten des Weges nur einmal durchlaufen werden, das Erreichen des Zielknotens muß also ohne Zyklen stattfinden. Es können zwei verschiedene Verfahren zur Berechnung der Tiefe unterschieden werden:

1. Die Tiefe eines Knoten wird erst gekennzeichnet, wenn er keine Vorgänger mehr hat. Begonnen wird mit dem ersten Knoten, das heißt dem Knoten, der keine Vorgänger hat. Dieser bekommt die Tiefe $n = 0$. Daraufhin wird der Knoten aus der Vorgängerliste entfernt und es wird nach den Knoten gesucht, die keinen Vorgänger mehr haben. Diese, (mindestens einer, da der Ablaufgraph

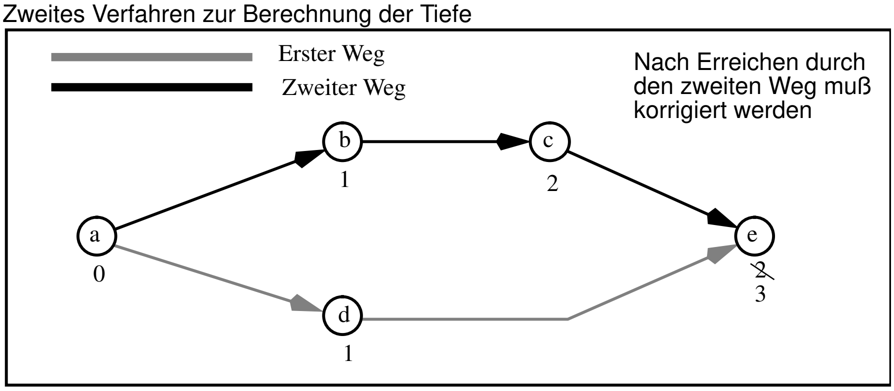
zusammenhängend ist) erhalten die Tiefe n = n + 1 und werden aus der Liste gelöscht. Dieses Verfahren wird solange angewandt, bis keine Vorgänger mehr in der Liste vorhanden sind.

**ABBILDUNG 27** Erstes Verfahren zur Berechnung der Tiefe




**2.** Das zweite Verfahren arbeitet ohne das Löschen der Vorgängerinformation und wird deshalb zur Berechnung herangezogen. Es wird, ausgehend von Startknoten, ein Weg bis zum Zielknoten durchlaufen und dabei jeder Knoten markiert und mit der Tiefe versehen. An der ersten Stelle, wo verzweigt wurde, wird ein neuer Weg verfolgt, bis ein schon markierter Knoten erreicht wird. Dort wird die eingetragene Tiefe mit dem verglichen, den er durch das Erreichen des neuen Weges bekommen würde. Ist der eingetragene Wert kleiner, so muß er korrigiert werden und alle seine schon markierten Nachfolger müssen ebenfalls korrigiert werden. Dieses Verfahren wird solange angewandt, bis keine unbehandelten Wege (Verzweigungen) mehr existieren.

**ABBILDUNG 28** Zweites Verfahren zur Berechnung der Tiefe




Da die Informationen, die durch Löschen aus der Vorgängerliste (beziehungsweise Adjazenzmatrix) verloren gehen würden, für die weiteren Berechnungen noch gebraucht werden, ist bei der Implementierung das zweite Verfahren gewählt worden.

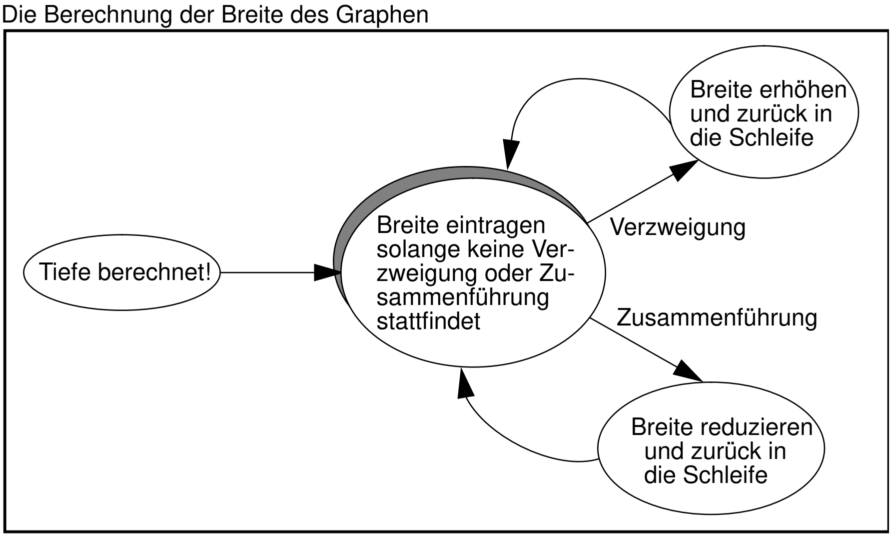
## 5.4 Die Berechnungen zur Breite des Graphen

Ebenso wie für jeden Knoten die Tiefe berechnet werden mußte, muß dies auch für die Breite geschehen. Unter der Breite wird der Grad der Verzweigung verstanden, indem ein Knoten sich befindet. Beispielsweise erhalten nach einer Verzweigung alle Knoten der entstehenden Äste die Breite 2, wenn die Verzweigung in zwei Äste mündet (bei drei Ästen die Breite 3 usw.).

Erst wenn eine Zusammenführung stattfindet, das heißt der Nachfolger des einen Astes ist derselbe wie der Nachfolger des anderen Astes (bei zwei Verzweigungen), wird die Breite reduziert. Fallen innerhalb der Äste weitere Verzweigungen an, dann werden die Werte dieser Verzweigungen, um den Grad dieser neuen Verzweigung erhöht. Dies ist natürlich nicht ausreichend für die Plazierung der Äste auf der Fläche, da nach jeder Verzweigung beide Äste die gleiche Breite bekommen.

Es muß also ein weiteres Verfahren zur Bestimmung des absoluten Wertes bezüglich der Breite eingesetzt werden. Hilfreich dabei sind die Vorgängerrelationen, die aus der Adjazenzmatrix abgelesen werden können. Deshalb wurde auch bei der Tiefenberechnung das oben an zweiter Stelle besprochene Verfahren verwendet, um eben diese Vorgänger an dieser Stelle noch präsent zu haben.

**ABBILDUNG 29** Die Berechnung der Breite des Graphen




Wendet man das obige Verfahren auf den gegebenen Beispielgraphen an, so ergibt sich folgende Einstufung:

**ABBILDUNG 30** Einstufung der Breite und der Tiefe

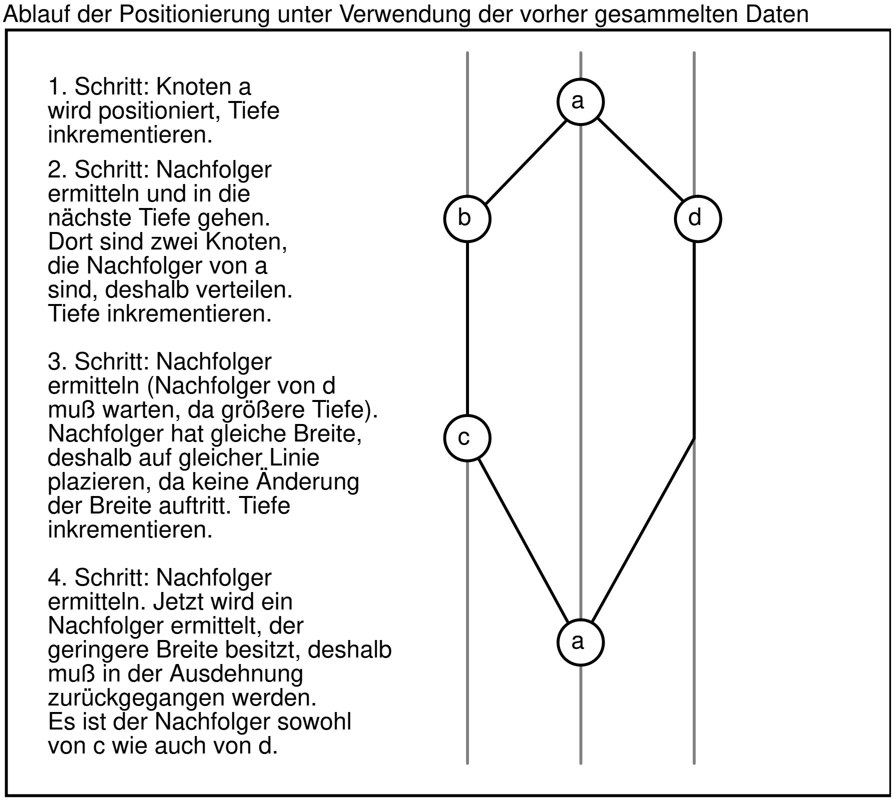
| | Tiefe 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Breite 1 | a | | | e |
| Breite 2 | | b,d | c | |

Man sieht schon bei diesem einfachen Beispiel, daß Doppelbelegungen in einzelnen Zellen auftreten. Würde man nun diese Parameter einfach dazu verwenden, die graphischen Objekte zu positionieren, würden ebenfalls Doppelbelegungen auf der Bildschirmfläche auftreten. Es würden einzelne Steps und Events übereinander gelegt werden, so daß die Darstellung unvollständig wäre.

Die weitere Vorgehensweise sieht nun vor, die Positionierung nach der Nachfolgerrelation und der Tiefe vorzunehmen. Das bedeutet, daß (bezogen auf obige Abbildung) zuerst Knoten a positioniert wird, daraufhin dessen Nachfolger, die die Tiefe 2 haben. Da dies zwei sind (was auch durch die Breite 2 impliziert wird), kann die Positionierung nicht auf der selben Linie stattfinden. Es werden deshalb beide Knoten auf der Fläche jeweils einer anderen Linie zugeordnet.

Im nächsten Schritt werden die Nachfolger von a und d positioniert, wobei zuerst der Knoten mit der am wenigsten entfernten Tiefe, das ist der Knoten c, angebracht wird, und zwar auf der gleichen Linie wie sein Vorgängerknoten b. Aus der Sicht des Knotens d wird mit der Positionierung des Nachfolgerknotens e solange gewartet, bis der Tiefenzähler den Wert des Nachfolgerknotens erreicht hat. Ist die Positionierung in der Tiefe des Nachfolgers angelangt und wird dann die Breite reduziert, dann handelt es sich um eine Zusammenführung, wie bei obigen Beispiel.

**ABBILDUNG 31** Ablauf der Positionierung unter Verwendung der vorher gesammelten Daten



Die Verteilung der Äste auf andere Linien hängt davon ab, ob der Grad der Verzweigung ungerade oder gerade ist. Im obigen Beispiel ist er gerade. Wäre er aber ungerade, so würde die mittlere Linie zur Positionierung mitbenutzt werden.

Welcher Knoten dabei auf die mittlere Linie plaziert würde, hängt davon ab, welcher Nachfolger an erster Stelle in der Nachfolgerliste des Knotens auftaucht. Eine sinnvolle, wenn auch aufwendige Strategie wäre es, die Äste nach ihrem maximalen Verzweigungsgrad zu durchsuchen und diese Äste dann am Rande zu plazieren. Denn die mittlere Linie ist besonders kritisch einzustufen, da bei Verzweigungen der beiden äußeren, deren Äste sich der mittleren Linie kritisch annähern könnten.

Die Strategie, die der Verteilung der Objekte auf der Fläche zugrunde liegt, arbeitet bei gleichförmiger Verzweigung gut. Problematisch wird die Positionierung mehrerer unterschiedlicher Verzweigungen. Beispielsweise, wenn nach einer Verzweigung mit zwei Ästen eine weitere Verzweigung in einem der beiden Äste stattfindet, die drei weitere Verzweigungsäste produziert. Es müssen dann, wenn man obiges Verfahren anwendet, unter Umständen (falls mehrere Äste auf der gleichen Linie arbeiten) aufwendige Korrekturen durchgeführt werden.

Koorigiert wird durch Reorganisation: Dabei werden die Äste *auseinandergezogen*, die Überschneidungen aufgewiesen haben. Es findet eine schrittweise Verbreiterung des gesamten Graphen statt. Eventuell sind mehrere Schritte notwendig, wenn nach einem Schritt noch kein korrektes Ergebnis vorliegt.

# KAPITEL 6 Objektorientierte Realisierung der Anwendung

Die Implementierung der Arbeit erfolgte in einer objektorientierten Sprache. Daraus ergibt sich die Notwendigkeit, einige Überlegungen über objektorientierte Ansätze in der Software-Entwicklung anzustellen. Die traditionelle Denkweise konzentriert sich darauf, **wie** die Lösung eines Problems implementiert werden soll. Die objektorientierte Denkweise versucht dagegen das Problem in einer verständlichen Form zu modellieren und stellt sich die Frage: **Was** soll programmiert werden? Das Aufkommen der sogenannten objektorientierten Programmierung deckt somit den Bedarf einer Bereitstellung von Objekten höherer Ebenen zur Repräsentierung abstrakter Schnittstellen, bei der Erstellung von Anwendungen. Anders ausgedrückt: Anwendungsquelltext, der um jedes Objekt herumgelegt ist, verringert die konzeptionelle Distanz zwischen Spezifikation und ausführbarem Programm [Mey88].

## 6.1 Das Konzept der Objektorientierung

Das Konzept der Objektorientierung baut im wesentlichen auf der Definition **des** Objektes auf. Ein Objekt ist eine wohlstrukturierte Menge von Daten. Diese können elementar sein (Integer, Char …) oder komplexere Strukturen (C-struct, graphische Objekte in Windows/4GL…) haben. Ein Objekt gehört immer einer vordefinierten *Klasse* an, auf die ganz bestimmte Operationen, sogenannte *Methoden*, zugelassen sind. Ein Objekt verbirgt seine Daten nach außen *(information hiding)*, dies wird als Einkapselung bezeichnet *(encapsulation)*. Der Zugang zu den Daten ist nur über die Methoden möglich. Das Erscheinungsbild eines Objekts hängt von den Daten, -wie sich ein Objekt verhält, hängt von den Methoden ab. Ein Objekt kann sich wieder aus anderen Objekten zusammensetzen, wodurch sich Hierarchien zwischen den Klassen ergeben können. Die Vererbung entspricht einer Weitergabe von Datenstrukturen und Operationen des Objektes an das Unterobjekt.

Anwendungen und Programme im objektorientierten Sinne sind eine Ansammlung von Objekten, die durch Interaktion untereinander die gestellte

Aufgabe erfüllen sollen. Die Interaktion zwischen den Objekten wird durch den Austausch von Nachrichten realisiert. Wird von einem Objekt eine Nachricht an ein anderes Objekt geschickt, so kann auf diesem eine Methode ausgeführt werden. Dadurch können sich die Daten des Objekts ändern, wodurch das Schicken einer Nachricht an ein in der Hierarchie höhereingeordnetes Objekt ausgelöst werden kann.

Durch Senden einer Nachricht an ein in der Hierarchie hocheingestuftes Objekt, kann ebenso (durch Vererbung der Nachricht auf alle untergeordnete Objekte) bewirkt werden, daß das angesprochene Objekt und alle seine Unterobjekte ihre Daten ändern. Die Änderung eines jeden Objektes kann dabei unterschiedlich erfolgen. Es liegt hier eine als **polymorph** bezeichnete Abstraktion vor, das heißt unterschiedliche Objekte können unterschiedlich auf die gleiche Nachricht reagieren.

### 6.1.1 Ingres Windows/4GL

Zur Implementierung der Aufgabe wurde Ingres Windows/4GL verwendet. Es handelt sich dabei um ein ereignisbasiertes Anwendungs-Entwicklungssystem, das eine große Anzahl Klassen vordefiniert hat. Der Austausch der Nachrichten wird über sogenannte Ereignisse (*events*[1]) gesteuert. Es wird dabei zwischen *systembedingten* und *benutzerdefinierten* Ereignissen unterschieden. Systembedingte Ereignisse werden bei Änderung der Daten eines Objektes ausgelöst, sie können vom Entwickler nicht unterdrückt werden. Benutzerdefinierte Ereignisse werden vom Entwickler explizit eingeführt und erhalten einen frei gewählten Namen.

Die Behandlung der Ereignisse (Erstellen eines Ereignisblocks durch Programmierung von Aktionen) wird vom Benutzer vollzogen. Das bedeutet, daß Aktionen, die nach Auslösen eines Ereignisses erfolgen sollen, vorher festgelegt werden müssen. Insofern ist es möglich, systembedingte Ereignisse zu *ignorieren*, indem auf Ereignisblöcke für manche Ereignisse verzichtet wird.

Die Klassen, die das System bereitstellt, repräsentieren hauptsächlich unterschiedlichste graphische Objekte (Linien, Kreise, Ellipsen, Textfelder, Fenster usw.), die über spezielle Methoden ihr Erscheinungsbild verändern können. Es gibt auch verschiedene Objektklassen, die zur Interaktion mit dem Benutzer verwendet werden. Beispielsweise werden Felder generiert, aus denen mit dem Zeiger eine Auswahl getroffen werden kann oder die zur Bestätigung einer Anfrage dienen. Darüberhinaus ist es dem Benutzer gestattet, selbst Klassen zu definieren und Hierarchien zwischen mehreren Objekten zu schaffen, indem Objekte verschiedener Klassen zu einer Klasse zusammengefaßt werden. Durch das System vordefinierte Hierarchien können jedoch nicht geändert werden.

Zu jeder Anwendung wird vom Entwicklungssystem ein Katalog erstellt, der alle statischen Objekte aufnimmt. Das sind solche, die während des Ablaufs der

1. Es sind hier nicht die Events des ConTracts gemeint, sondern Ereignisse, die auf Objekte der Implementierung basieren.

Anwendung nicht gelöscht werden können. Innerhalb des Katalogs werden auch benutzerdefinierte Klassen aufgenommen. Die Einteilung der Komponenten des Katalogs gliedert sich in:

- Klassen
- Variablen
- Frames (Fenster)
- Prozeduren (Windows/4GL- und 3GL-Prozeduren)

Während der Laufzeit können Objekte aller oben aufgeführten Komponenten dynamisch generiert und zur weiteren Verarbeitung verwendet werden. Sie werden nach Ablauf der Anwendung wieder gelöscht, so daß keine Information über sie mehr vorliegt.

Ingres Windows/4GL faßt alle beteiligten Datenstrukturen als Objekte auf, so auch Prozeduren, die global deklariert werden. Demgegenüber werden Quelltexte, die um Objekte herumgelegt sind (die also eindeutig Objekten zugeordnet sind und *script*[2] genannt werden), als **Attribute** bezeichnet.

Einige spezielle Objekte müssen mit sogenannten *Statements* angesprochen werden. Das sind Programmieranweisungen, die den Inhalt eines *scripts* in Winodws/4GL bilden. Mit den Statements werden beispielsweise Prozeduren aufgerufen, Fenster geöffnet oder verlassen.

## 6.2 Die Objekte des Ablaufmonitors

Die Hierarchien, die durch die Windows/4GL-Systemklassen **vorgegeben** sind, beziehen sich ausschließlich auf **graphische** Objekte. Hierarchien zwischen anderen Objekten, wie beispielweise Prozeduren oder Frames gibt es nicht. So besteht ein Menü, das als *Menu Field* bezeichnet wird, aus einem oder mehreren *Menu Items*, einer *Menu Group* und einem *Menu Seperator.* Diese setzen sich selbst wieder aus anderen Objekten zusammen.

### 6.2.1 Die kombinierten, graphischen Objekte

Die wichtigsten, vom Entwickler definierten Objekte sind die speziell für die Anwendung generierten Repräsentanten der Steps und der Events. Sie umfassen Daten, die aus der Datenbank gelesen werden und als Informationen für den Aufbau wichtig sind. Dazu kommen graphische Objekte, wie der Kreis (*ellipseshape*) und das Textfeld (*entryfield*), das den Namen des Steps trägt. Dazu wurden zwei Klassen generiert, deren Aufbau in Kapitel 4 beschrieben wird.

- Step_Class

  und
- Event_Class

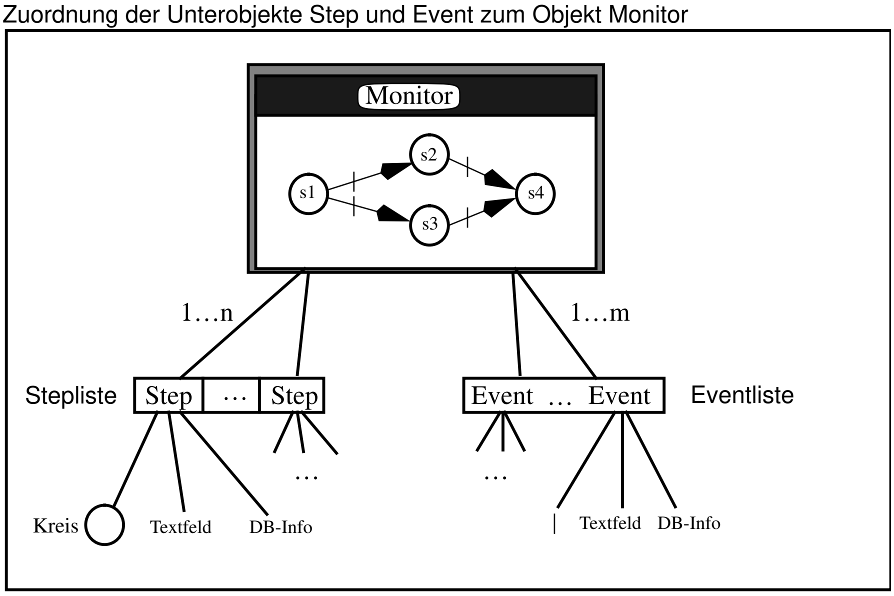
2. Es ist nicht das Skript des ConTracts gemeint, sondern ein spezielles Windows/4GL-Attribut.

Die Events bestehen aus einem Balken (*segmentshape*), einem Textfeld (welches bei den meisten Abbildungen nicht zu sehen ist) und den Informationen aus der Datenbank. Der Monitor gehört zu der systemdefinierten Klasse *FrameExec.*

Die Steps und die Events sind Unterobjekte des Monitors, sie werden unter Angabe von Koordinaten relativ zu dessen linker, oberer Ecke angeordnet. Da sie Unterobjekte des Monitors sind, richtet sich das Verhalten nach diesem. Beispielsweise hängt die Farbe des Hintergrunds der Stepkreise von der Hintergrundfarbe des Monitors ab. Nur wenn ein Step explizit angesprochen wird, kann eine Änderung des ausgewählten Objektes stattfinden.

Der Monitor besteht aus dem Fenster (das den Ablauf aufzeigt), dem Programmcode und den Repräsentanten der Steps und der Events und könnte als der Repräsentant des ausgewählten ConTracts bezeichnet werden.

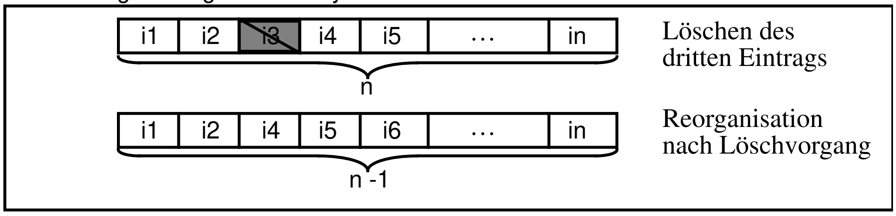
**ABBILDUNG 32** Zuordnung der Unterobjekte Step und Event zum Objekt Monitor




Während des Ablaufs lassen sich alle Steps und Events eindeutig identifizieren. Beispielsweise erfolgt die Suche nach einem Step unter Angabe des Namens. Dieser setzt sich aus dem Buchstaben ‘s’ und der `step_id` der Relation `Steps` aus der Datenbank zusammen. Mit diesem wird die Liste der Steps durchsucht und man kann somit den Zeiger auf den Kreis (den graphischen Repräsentanten des Steps) erhalten.

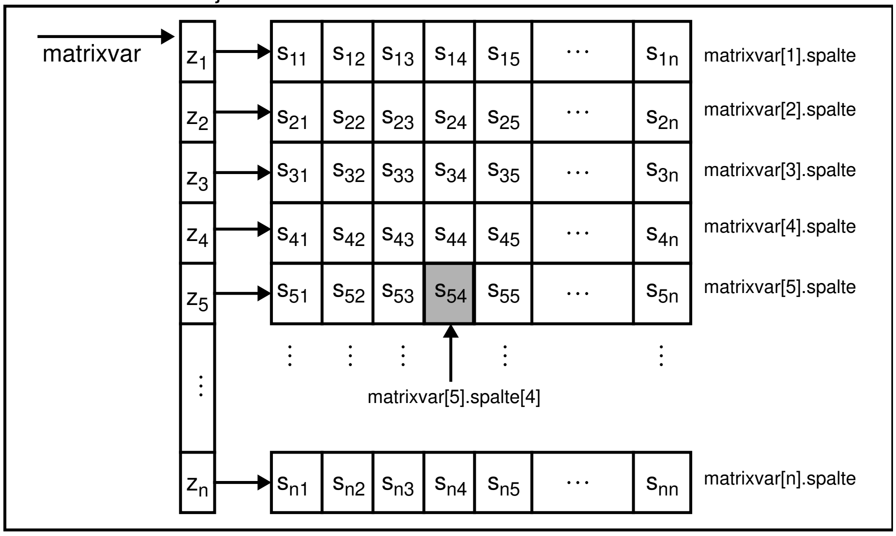
Die Liste der Steps und der Events sind wiederum jeweils ein einziges Objekt und gehören der Klasse *arrayobject* (dynamische Listen) an. Solche Listen lassen sich als Ganzes behandeln, es kann aber auch über einen indizierten Zugriff auf deren Unterobjekte zugegriffen werden.

**ABBILDUNG 33** Selbständige Reorganisation dynamischer Listen




Das dynamische Verhalten der Listen zeichnet sich durch selbständige Reorganisation nach dem Löschen oder Hinzufügen eines Elements aus. Auch die zur Berechnung der Positionen verwendete Matrix setzt sich aus einer Liste zusammen, wobei diese Liste aus weiteren Listen besteht und dadurch erst ihre Matrixstruktur erhält.

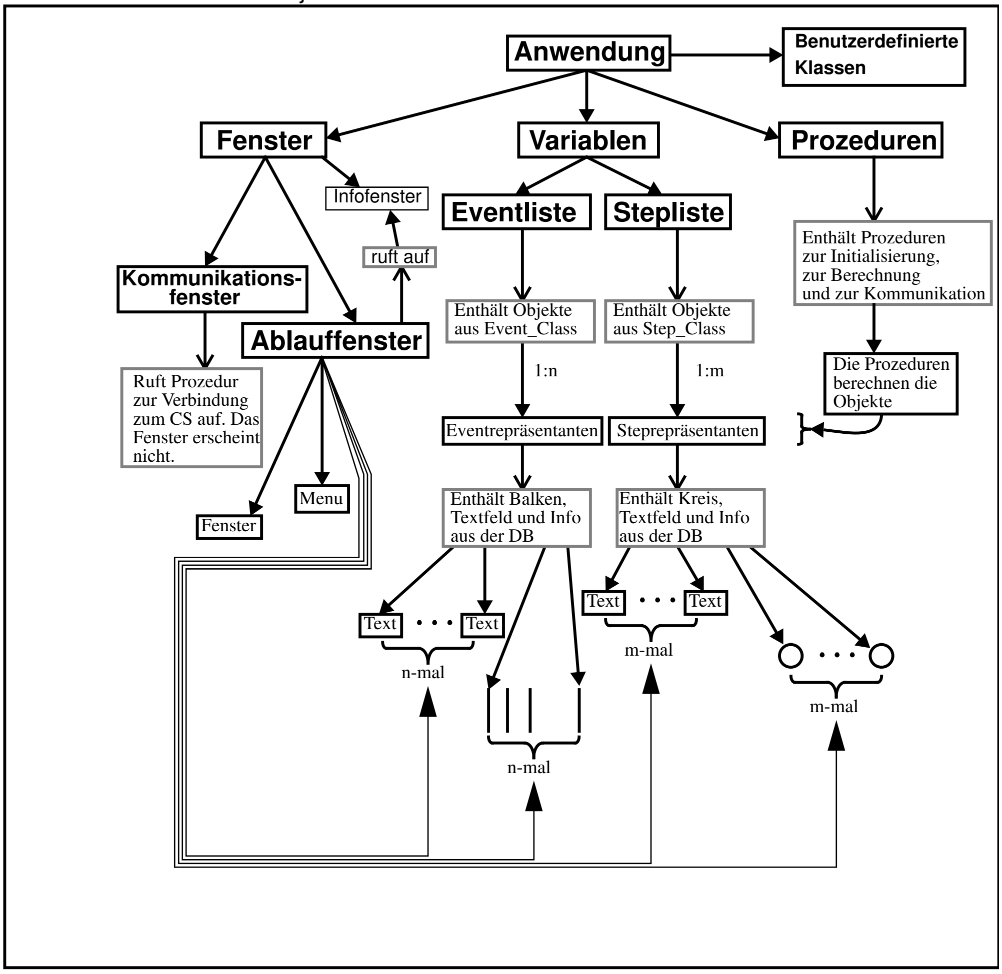
**ABBILDUNG 34** Die Struktur des Objektes `matrixvar`




### 6.2.2 Die Objekthierarchie der Anwendung

Die Objekthierarchie der Anwendungen umfaßt alle Variablen, Klassen, Fenster und Prozeduren. Folgende Abbildung zeigt insbesonders die graphischen Objekte, die als Variablen definiert wurden. Dazu wird deren Beziehung zum Ablauffenster skizziert. Die nähere Beschreibung der Klassen wurde aus Übersichtsgründen nicht mit aufgenommen. Ebenfalls wurden die Prozeduren nicht näher ausgeführt.

**ABBILDUNG 35** Hierarchie der Objekte



# KAPITEL 7 Verbindung zum Gesamtsystem

Bei der Verteilung von einzelnen Bestandteilen eines Systems auf sowohl logisch wie auch physikalisch verschiedene Knoten ist die Verbindung der einzelnen Funktionseinheiten untereinander von hoher Wichtigkeit. Abhängig vom Gesamtsystem muß ein passendes Konzept zur Nachrichtenübermittlung gewählt werden. Im folgenden soll nun eine nach zunehmendem Abstraktionsniveau gegliederte Übersicht die konzeptionelle Möglichkeit der Kommunikation über gängige Kommunikationsmechanismen aufzeigen. Es werden dabei Bezeichnungen verwendet, die üblicherweise häufig im Zusammenhang mit speziellen Betriebssystemen (wie Sockets/Unix, Mailboxes/VMS) benutzt werden. Die hier aufgeführte Beschreibung versucht aber nur unterschiedliche Möglichkeiten aufzuzeigen, ohne auf die speziellen Realisierungen tiefer einzugehen. Es sei hier außerdem darauf hingewiesen, daß einige Mechanismen sich durchaus unter Verwendung anderer realisieren lassen. So werden Pipes im Betriebssyetm Unix unter Verwendung von Sockets realisiert.

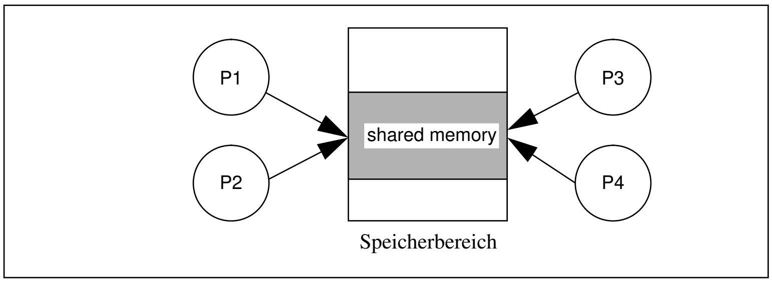
## 7.1 Grundlagen der Kommunikation

Es können einige grundsätzlich verschiedene Konzepte zur Übermittlung von Nachrichten und zur Benutzung gemeinsamer Daten klassifiziert werden. Einige davon werden im folgenden vorgestellt.

### 7.1.1 Nutzung gemeinsamer Speicherbereiche (shared memory)

Die Nutzung **gemeinsamer Speicherbereiche** ist eine sehr effiziente Methode zur Nutzung **gemeinsamer Daten** durch mehrere Prozesse. Dabei wird ein Speicherbereich definiert, der für alle beteiligten Prozesse genutzt werden kann. Die Schwierigkeit liegt bei dieser Methode in der Festlegung einer geeigneten Synchronisierung, da alle Prozesse in der Regel Schreib- und Leseberechtigung für den gemeinsam genutzten Bereich erhalten.

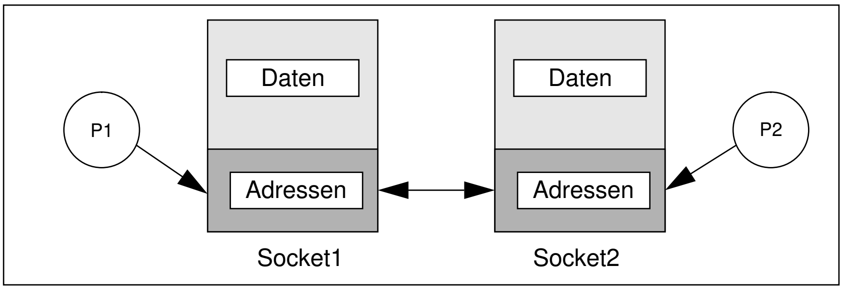
**ABBILDUNG 36** Nutzung durch gemeinsamen Speicher




### 7.1.2 Kommunikation über Sockets

Unter Sockets versteht man einen Bereich, der für den ihm zugeordneten Prozeß einen **Puffer** zuordnet. Dieser Puffer kann von anderen Prozessen unter Angabe spezieller **Adressen** angesprochen werden. Es kann lokal auf einem Unixrechner oder innerhalb eines Netzes (dann unter Verwendung von Netzwerksoftware, beispielsweise TCP oder UDP) mit Sockets kommuniziert werden.

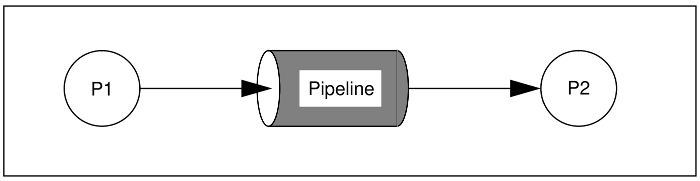
**ABBILDUNG 37** Verbindung über Sockets




### 7.1.3 Verwendung von Pipelines

Eine spezielle Variante, die vorallem unter Unix (pipes) häufig benutzt wird, ist die Verwendung sogenannter Pipelines. Bei dieser Technik wird die **Ausgabe** eines Prozesses in einen Puffer geschoben, der dann nach dem FIFO-Prinzip von einem anderen Prozeß gelesen und als **Eingabe** verwendet wird. Soll dabei in beide Richtungen kommuniziert werden, so müssen mehrere Pipelines aufgebaut werden.

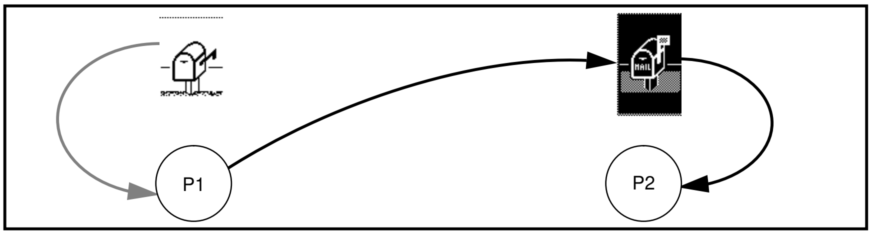
**ABBILDUNG 38** Verbindung mit einer Pipeline




### 7.1.4 Verwendung von Briefkastensystemen

Bei einer Verwendung von Briefkastensystemen werden die versendeten Nachrichten an einer speziellen Stelle abgelegt, von wo aus sie der Empfänger abholen kann. Es findet meist eine Benachrichtigung des Empfängers statt, woraufhin dieser sich die Nachricht nach Bedarf abholt und sie verarbeitet.

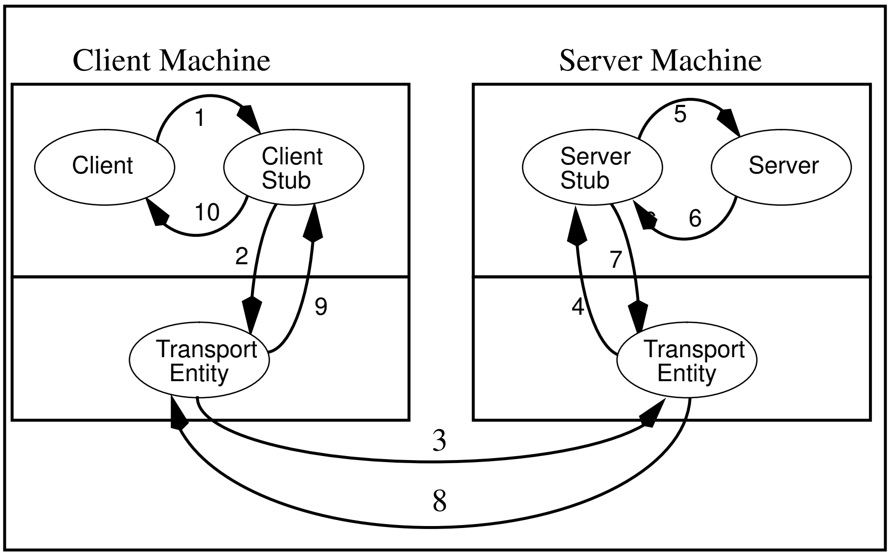
**ABBILDUNG 39** Kommunikation über Briefkastensysteme




### 7.1.5 Rechnerverlagerter Unterprogrammaufruf (Remote Procedure Call)

Der sogenannte RPC-Mechanismus erlaubt es einem Benutzer sowohl auf Kommando-, wie auch auf der Programmierebene Kommandos ausführen zu lassen, die nicht lokal ausgeführt werden. Das bedeutet, daß Dienste, die auf der lokalen Rechenmaschine nicht verfügbar sind, auf einem anderen Rechner ausgeführt und die Ergebnisse bei der lokalen Verarbeitung verwendet werden können.

**ABBILDUNG 40** Der Ablauf eines RPC-Aufrufs[Tanen88]




Obige Abbildung veranschaulicht die Ausführung eines RPC-Aufrufs, wie er normalerweise stattfindet. Der Client ruft ein Unterprogramm mit dessen Namen auf. Die Client-Stubs ver- und entpacken die Parameter zu einer Nachricht und die Transporteinheit fügt der Nachricht noch die benötigten Header (das sind Addressinformationen für die Netzwerksoftware) hinzu.

## 7.2 Notwendige Betriebsumgebung im APRICOTS

Damit innerhalb des gesamten APRICOT-Systems kommuniziert werden kann, müssen zuerst einige Grundvoraussetzungen geschaffen sein. Die Kommunikation selbst findet dann über sogenannte RM_calls [Laube92] statt. Dieser Mechanismus arbeitet ähnlich wie das unter Bereich 7.1.2 auf Seite 52 dargestellte Verfahren.

### 7.2.1 Voraussetzungen

Innerhalb von APRICOTS ist die graphische Oberfläche als **ConTract-Client** einzustufen, sie repräsentiert die Schnittstelle zum System, siehe Kapitel 1. In Bezug auf das Kommunikationssystem [Laube92] kann eine Behandlung als **Komponente** erfolgen. Hierbei liegt die Schwierigkeit in der Einbindung der Oberfläche ins System, da dies normalerweise nach einem festen Schema geschieht.

Damit der Ablauf ordnungsgemäß funktionieren kann, müssen gewisse Voraussetzungen geschaffen werden. Dazu muß sicher gestellt sein, daß das Kommunikationssystem und ein ConTract-Manager gestartet sind. Das Kommunikationssystem (CS) ist als zentrale Einheit gedacht und somit der Vermittler von Nachrichten, die zur Verarbeitung eines ConTracts zwischen den einzelnen Komponenten ausgetauscht werden. Deshalb ist es auch wichtig für die Oberfläche. Es sendet und empfängt Meldungen vom ConTract-Manager. Der ConTract-Manager ordnet jedem initiierten ConTract (mittels *Cid*=ConTract-Id) eine Identifikation zu, führt darauf dessen Kommandos aus und überwacht den Gesamtablauf. Die Oberfläche kann so über das CS die Zustandsänderungen des ConTracts erfahren und anzeigen.

**ABBILDUNG 41** Herkömmliche Einbindung einer Komponente ins System

```
/* Routinen aus dem CS */
main()
RM_call_async()
RM_call_sync()
RM_call_return()
/* Komponentenbeschreibung vom Benutzer*/
Anweisung1();
Anweisung2();
```

Üblicherweise werden alle Serverprozesse (wie in obiger Abbildung zu sehen ist) als C-Programme implementiert. Dabei werden die eigentlichen Anweisungen mit den Routinen, die vom Kommunikationsystem kommen (inklusive der Hauptroutine), in einer Datei zusammengefaßt und zu einem ausführbaren Programm übersetzt. Die Implementierung der Oberfläche wurde aber in einer Sprache der vierten Generation erstellt, die eigene Hauptroutinen verwendet. Es ist deshalb notwendig, ein anderes Verfahren zur Verbindung mit dem System zu finden.

### 7.2.2 Anschluß an die Betriebsumgebung

Der Anschluß an das System ist an die Lösung mehrerer Probleme geknüpft. Diese sind durch die Benutzung verschiedener Programmierwerkzeuge folgendermaßen zu gliedern:

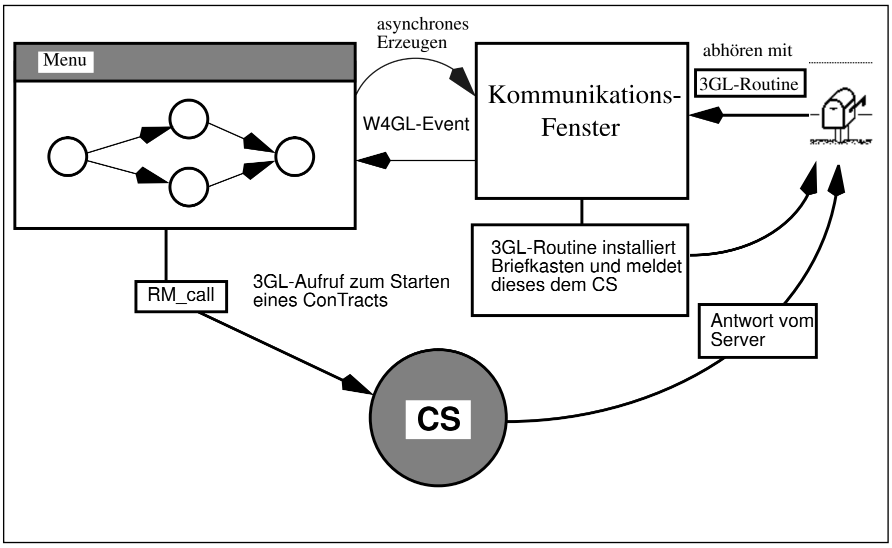
**1.** Die Anwendung muß auf zwei Eingänge *hören*:
- Windows/4GL-Events zur Steuerung der Anwendung.
- Nachrichten des Kommunikationsystems.

**2.** Die Anbindung kann nicht nach der herkömmlicher Methode erfolgen.

Im Detail gesprochen: Es werden zur graphischen Ausgabe der Oberfläche vom Hauptfenster nur 4GL-Events verarbeitet. Beim Aufbau des Hauptfensters wird von diesem asynchron ein *Kommunikationsfenster* erzeugt. Dieses asynchrone Fenster meldet den Prozeß beim CS an und richtet ein spezielles Briefkastensystem ein. Daraufhin schaut dieses bis zur Beendigung des Prozesses fortwährend in den Briefkasten und konvertiert die hereinkommenden Nachrichten in ein für den Prozeß taugliches Format. Die Anmeldung an das CS und das Abhören des Briefkastens werden über C-Unterprogramme realisiert, die von der Hauptroutine aufgerufen werden. Die Konvertierung der Nachrichten erfolgt mit 4GL.Unter einer Konvertierung muß man ein Aufspalten eines Eingabe-Strings in vom Hauptfenster verwendbare Parameter verstehen.

Nach der Konvertierung schickt das Kommunikationsfenster die Nachricht an das Hauptfenster durch Senden eines 4GL-Events. Diese Parameter werden dann einfach dem versendeten Event mitangehängt und erreichen so gleichzeitig mit diesem das Hauptfenster. Das Erhalten eines Events veranlaßt das Hauptfenster zu einer Reaktion, die sich beispielsweise durch die optische Veränderung des graphischen Gebildes eines Steps nach dessen tatsächlicher Zustandsänderung bemerkbar machen kann.

**ABBILDUNG 42** Die Anbindung der Oberfläche



## 7.3 Die Kommunikation während des Ablaufs

Sobald die Anbindung an das System installiert ist, werden Nachrichten zwischen diesem und der Oberfläche auf zwei verschiedene Arten ausgetauscht:

1. Der Benutzer gibt über die Oberfläche Befehle ein, die den ConTract betreffen (start, suspend, resume…). Diese werden an das CS geschickt. Der Aufruf ist ähnlich dem unter ABBILDUNG 39. beschriebenen.
2. Die Bestätigung des Aufrufs vom CS gelangt an das Hauptfenster über das asynchrone Fenster. Diese Nachrichtenübermittlung hat den Charakter der unter 7.1.4. beschriebenen Art.

### 7.3.1 Verbindung der graphischen Oberfläche mit dem System

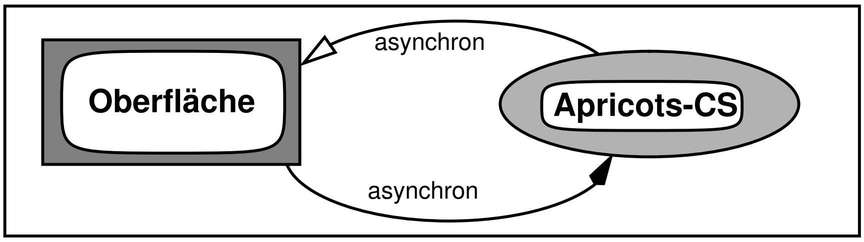
Durch die Benutzung zweier verschiedener Techniken zur Anbindung der Oberfläche entstehen Unterschiede in der Abhängigkeit unter den kommunizierenden Teilen. Der direkte Aufruf der Basisbefehle an das CS zeigt, daß die Oberfläche stärker vom CS abhängig ist.

Die Oberfläche braucht die Kommunikation über das CS, da sie sonst keinen Zugang zum ConTract-Manager hat. Umgekehrt ist die Oberfläche für das CS nur einer unter eventuell vielen Clients, deshalb werden die Meldungen vom CS an die Oberfläche asynchron versendet. Wird die Oberfläche abgehängt, so hängt das weitere Verhalten von der gewählten Fehlerstrategie ab. Es können ganz allgemein zwei Fehlerstrategien definiert werden:

1. Wird die Oberfläche abgehängt, so werden auch alle damit verbundene Prozesse beendet und somit der ConTract angehalten.
2. Der ConTract läuft weiter, obwohl es keine Oberfläche mehr dafür gibt, der den Ablauf steuern kann.

Diesbezüglich sind innerhalb der Arbeit keine Strategien verbindlich verarbeitet worden.

**ABBILDUNG 43** Zustand der Verbindung während des Ablaufs



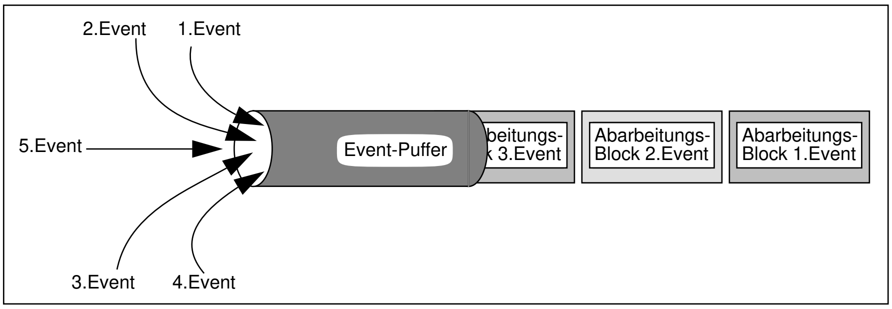

### 7.3.2 Verbindung der Bestandteile des Ablaufmonitors

Innerhalb der Anwendung kommunizieren das Hauptfenster und das Kommunikationsfenster über ein internes, ereignisgesteuertes Verfahren: Es werden Ereignisse ***(Events)*** ausgelöst. Die gesamte Oberfläche läuft innerhalb ihrer Grenzen synchron, das heißt der Kontrollfluß zwischen den einzelnen Komponenten der

Anwendung läßt keine Nebenläufigkeiten zu. Die interne Steuerung unterliegt vollkommen der Kontrolle durch die Events und deren Auftreten wird sequentiell abgearbeitet.

Das läuft folgendermaßen ab: Jede Aktion, die innerhalb der Anwendung stattfindet, löst einen Event aus. Diese Events werden gepuffert und in der Reihenfolge ihres Erscheinens abgearbeitet. Es macht dabei keinen Unterschied, ob Events systembedingt oder ob sie durch den Benutzer ausgelöst wurden.

**ABBILDUNG 44** Der Kontrollfluß innerhalb der Anwendung




Die Abarbeitungsblöcke eines jeden Events müssen explizit vom Entwickler erstellt werden. Events, zu denen keine Abarbeitungsblöcke definiert wurden, sind ohne Wirkung.

KAPITEL 8

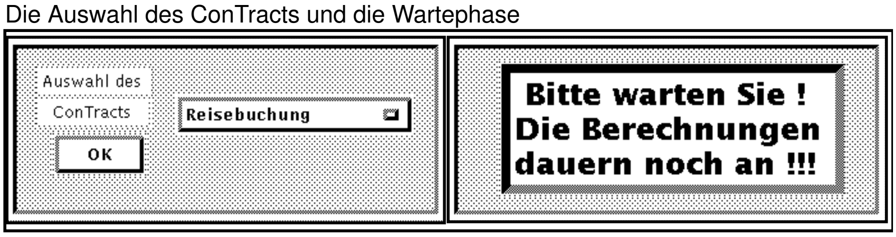
# Arbeitsweise der graphischen Anwendung

Für das Benutzermodell der Oberfläche ist es wichtig, wie die einzelnen Teile geschaffen sind und welche Änderungen sie während des Ablaufs durchlaufen können. Dafür müssen über die Schnittstelle Parameter zur Verfügung gestellt werden, die dem Kommunikationssystem ermöglichen, die Zustandsänderungen in geeigneter Form der Oberfläche mitzuteilen. Für den Benutzer, der unmittelbar die Oberfläche bedient, sollte ein übersichtliche Anordnung von Informationen und Kommandoeinheiten bereitgestellt werden.

## 8.1 Der Ablauf eines ConTracts im Monitor

Der Ablauf mit dem Monitor läßt sich in zwei Phasen unterteilen. In der ersten Phase wird der Contract ausgewählt und daraufhin der Graph erstellt. Die zweite Phase entspricht dann dem eigentlichen Ablauf des ConTracts, in dem sich Zustände ändern und dies auch optisch angezeigt wird. Die erste Phase beginnt mit der Auswahl des ConTracts, der mit dem Zeiger aus einem Listenfeld ausgewählt wird. Nach Bestätigung der Auswahl durch Betätigen des OK-Buttons muß der Benutzer warten, bis die Berechnungen abgeschlossen sind.

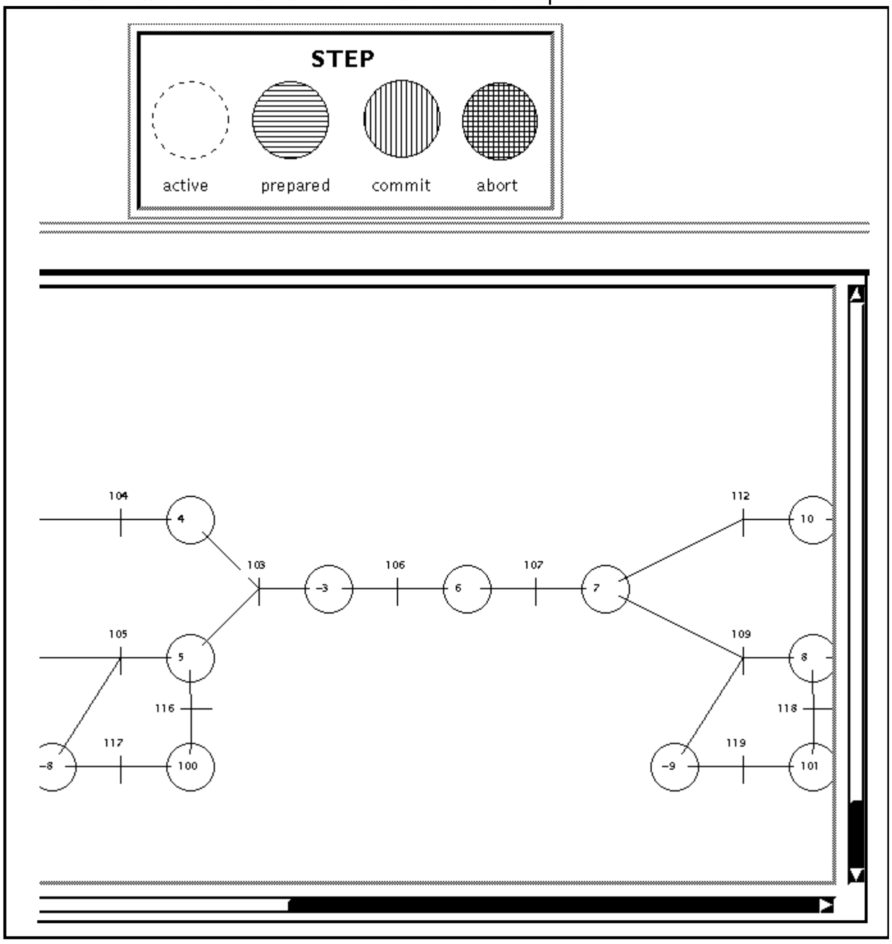
ABBILDUNG 45 Die Auswahl des ConTracts und die Wartephase




Nach Abschluß der Berechnungen erscheint der Monitor des Ablaufgraphen in der Mitte des Bildschirmes. Über dem Monitor befindet sich eine Informationspalette, die die unterschiedlichen Zustände der Steps anzeigt. Die

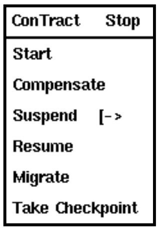
Anordnung auf dem Monitor ist so gestaltet, daß der Step der momentan in Bearbeitung ist, zentral plaziert wird. Am Anfang wird der erste Event knapp neben dem linken Rand des Fensters plaziert. In der Regel reicht der Platz auf dem Bildschirm nicht aus, um einen ConTract ganz auf dem Bildschirm anzuordnen. Abhilfe leisten Laufbalken (*scrollbars*) an den Rändern, die es ermöglichen den Graph hin und her zu schieben.

**ABBILDUNG 46** Ein Ausschnitt des Monitors nach Erstellen des Graphen




Wird der ConTract über das Menu gestartet, so findet eine Anbindung ans Kommunikationssystem, wie es im vorigen Kapitel beschrieben wurde, statt. Nach der Anbindung empfängt der Monitor Informationen des CS, die über das unsichtbare Kommunikationsfenster asynchron an den Monitor geschickt werden. Während des ganzen Ablaufs wartet der Monitor fortdauernd auf die Nachrichten des Kommunikationsfensters. Erhält der Monitor das Signal, daß der Event 1 getriggert wurde, so erscheint eine Meldung über einen erfolgreichen Verlauf des ConTracts. Beim Scheitern einer Konfliktschliefe erhält der Benutzer eine entsprechende Meldung und kann eventuell einen Kompensationsvorgang einleiten. Bei erfolgreichem Abschluß kann der Benutzer des Fenster auffrischen (*reset*) und einen neuen ConTract der gleichen Art nochmal starten.

**ABBILDUNG 47** Das Menu des Monitors




Wählt der Benutzer den Stop-Vorgang bei einem nicht aktiven ConTract aus, so endet die Anwendung und verschwindet vom Bildschirm. Beim Verlassen wird an das CS eine Nachricht geschickt, die das Ende der Anwendung ankündigt. Dies wird nur der Vollständigkeit halber gemacht, um dem CS zu signalisieren, daß keine weiteren Nachrichten mehr empfangen werden können.

Die Anwendung ist so entworfen, daß alle Eingaben des Benutzers entweder mit dem Mauszeiger oder mit Sondertasten erfolgen. Wird textuelle Eingabe, wie etwa bei der Auswahl des ConTracts verlangt, so erhält der Benutzer eine Liste, aus der er den passenden Begriff auswählen kann. Dadurch werden fehlerhafte Eingaben weitgehend vermieden, was die Fehlertoleranz insgesamt erhöht.

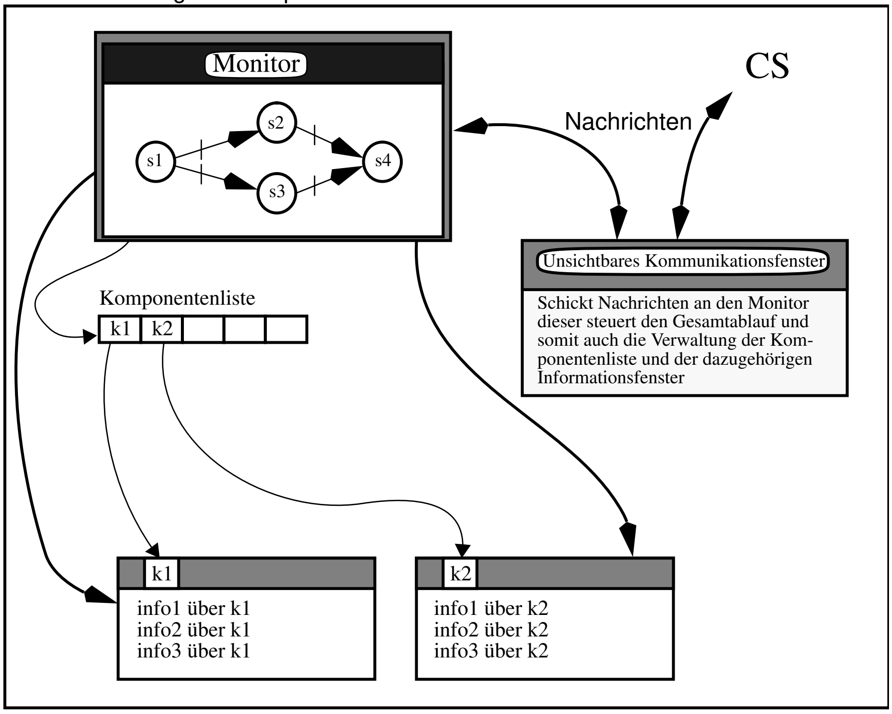
## 8.2 Das freie Monitoring der Komponenten

Der Ablaufmonitor sieht vor, daß alle Komponenten eines ConTracts ein Informationsfenster erhalten. Es werden innerhalb der Anwendung Nachrichten empfangen, die sich in zwei Arten unterteilen lassen. Die erste Art betrifft die Steps und die Events und deren Zustandsänderungen. Die zweite Art betrifft die Komponenten des ConTracts. So soll nach Erhalt einer Nachricht, die eine Komponente betrifft, in einer Liste nach einem Eintrag mit dem Namen der Komponente gesucht werden. Wird ein Eintrag gefunden, so weiß der Monitor, daß es schon ein Informationsfenster für die betreffende Komponente gibt. Ist kein Eintrag vorhanden, so muß ein neues Informationsfenster generiert werden.

Innerhalb des Informationsfensters befindet sich ein Textfeld, das die hereinkommenden Meldungen anzeigt. Finden sich viele Meldungen für eine Komponente ein, so kann auch hier mit einem Laufbalken die Fokusierung verschoben werden. Will ein Benutzer beispielsweise eine Information, die aus der aktuellen Anzeige herausgewandert ist, nachsehen, so kann er einfach die Anzeige zurückdrehen, indem er mit dem Mauszeiger den Balken zurückschiebt. Bei vielen Informationsfenstern kann es dem Benutzer passieren, daß die Anordnung etwas unübersichtlich wird, weil kein freier Platz mehr auf dem Bildschirm vorhanden ist.

Der Benutzer kann nicht von sich aus nähere Informationen über eine beteiligte Komponente einholen. Das heißt, er kann nicht eine Komponente direkt ansprechen und um Information nachfragen. Er kann bestenfalls ein Informationsfenster generieren und so einen Eintrag in die Liste vornehmen. Das Resultat ist dann eben ein Fenster ohne Information. Dieser Umstand liegt daran, daß Komponenten sich während des Ablaufs eines ConTracts nicht direkt ansprechen lassen, wenn dies nicht speziell programmiert wurde.

**ABBILDUNG 48** Freies Monitoring der Komponeten




## 8.3 Das Format der Information

Um eine richtige Abarbeitung der Information zu gewähren, müssen bezüglich des Formats Restriktionen festgelegt werden. Anhand des Formats der Nachricht, wie sie das Kommunikationsfenster der Anwendung erhält, muß die Nachricht entschlüsselt werden können. Es gibt dabei Unterscheidungen zu treffen, ob es sich um eine Nachricht handelt, die den Ablauf betreffen (Steps und Events) oder ob es sich um Hintergrundinformationen handelt (Komponenten). Die Unterscheidung erfolgt allein durch das Format der Nachricht. Es dürfen innerhalb der Nachricht keine Leerzeichen vorkommen, falls doch welche auftreten, werden sie eliminiert.

**ABBILDUNG 49** Das Format einer Nachricht, wie sie Anwendung vom CS empfängt

0@event/stepname@zustand@

Format einer Nachricht, die den Ablauf betreffen

1@Komponentenname@String@

Format einer Nachricht, die den Ablauf betreffen

### 8.3.1 Das Format einer Nachricht, die den Ablauf betrifft

Kommen Informationen über Zustandsänderungen der Steps und Events vom CS in die Anwendung herein, werden sie vom Kommunikationsfenster so umgeformt, daß der Monitor die Information, die er erhält, leicht verarbeiten kann. Das Kommunikationsfenster bekommt immer einen String der Länge 256. Läßt sich erkennen, daß es sich um eine Information für den Ablauf handelt, wird die Nachricht mit dem zugehörigen Stepnamen beziehungsweise Eventnamen versehen und zusammen mit der Zustandskennung an den Monitor geschickt. Handelt es sich um einen Step, dann wird als erster Parameter ein String, der mit einem ‘s’ beginnt, gefolgt von der `step_id` des betreffenden Steps, verschickt. Handelt es sich um einen Event, wird ein String der mit einem ‘e’ beginnt, gefolgt von der `event_id` des Events, verschickt.

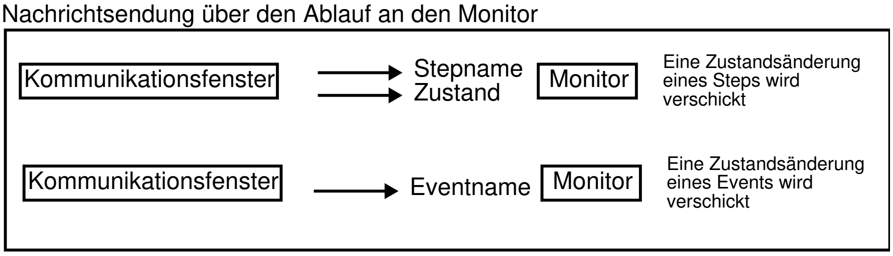
Die Zustände, die ein Step annehmen kann, werden innerhalb der Anwendung als Konstanten geführt und erhalten einen global verwendbaren Namen.

**TABELLE 8** Die Tabelle der Zustandkonstanten

| Name der Konstante | Datentyp | Wert | Anmerkung |
|---|---|---|---|
| ST_INIT | Integer | 1 | Ausgangszustand |
| ST_AKTIVATE | Integer | 2 | Step aktiv |
| ST_PREPARE | Integer | 3 | Step bereit zum Beenden |
| ST_COMMIT | Integer | 4 | Step abgeschlossen |
| ST_ABORT | Integer | 5 | Step abgebrochen |

Für die Events sind keine Konstanten für die Zustandskennung definiert worden. Anfänglich befinden sie sich in einem Initialzustand, den sie nach einer Triggerung verlassen. Es muß zur Benachrichtigung einer Triggerung nur der Eventname verschickt werden, der Monitor weiß dann automatisch, daß er den Event als getriggert anzuzeigen hat.

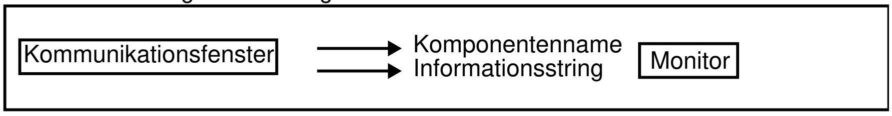
**ABBILDUNG 50** Nachrichtsendung über den Ablauf an den Monitor




### 8.3.2 Das Format einer Nachricht, die Hintergrundinformation liefert

Wenn Nachrichten vom CS in die Anwendung hereinkommen, werden diese vom Kommunikationsfenster in zwei Teile umgeformt. Der eine Teil enthält den Namen der Komponente, der andere besteht aus der Information über die Komponente. Das Format der Nachricht ist ähnlich dem beim Verschicken einer Zustandsänderung eines Steps, da er ebenso aus zwei Teilen besteht.

**ABBILDUNG 51** Nachrichtsendung über Hintergrundinfromation an den Monitor




Das Kommunikationsfenster arbeitet somit nach dem Prinzip: Aufspalten und Einteilen. Die Aufspaltung entspricht der Formatierung der Nachricht in die Form wie sie der Monitor erhält. Das Einteilen entspricht dem Kanalisieren der Nachrichten zum richtigen Abarbeitungsblock.

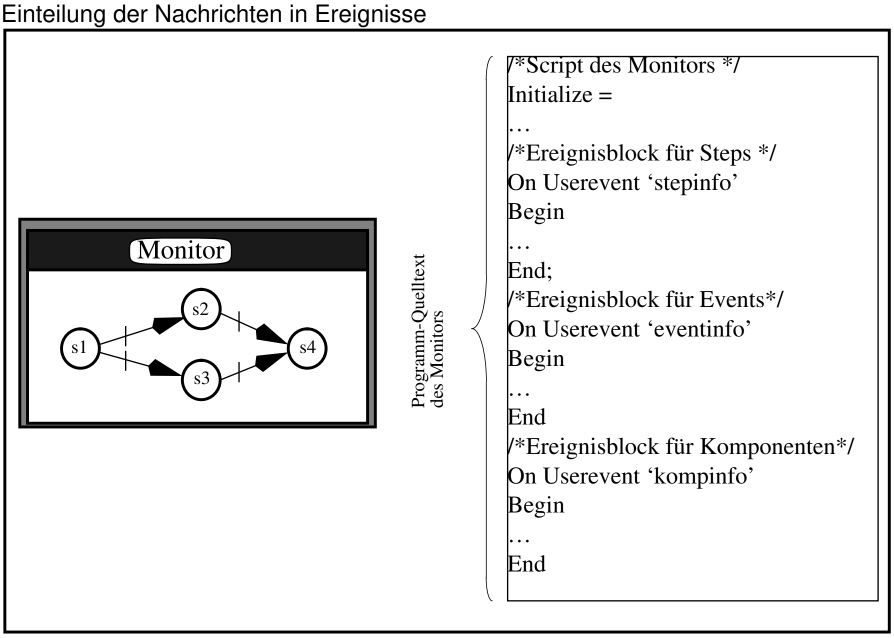
### 8.3.3 Das Einteilen der Nachrichten in Ereignisse

Da das ganze System ereignisgesteuert ist, bietet es sich an, eine Einteilung der Nachrichten in verschiedene Ereignisse vorzunehmen. Es findet dadurch eine Vorselektierung statt, die ohne diese Einteilung vom Monitor vorgenommen werden müßte. Das Kommunikationsfenster teilt die Nachrichten in drei Ereignisse ein:

1. Ereignis, das einen Step betrifft.
2. Ereignis, das einen Event betrifft.
3. Ereignis, das eine Komponenete betrifft.

Der Vorteil dieser Einteilung liegt darin, dem Monitor innerhalb der Ereignisabarbeitungssequenz die Unterscheidung der Nachrichten abzunehmen. Es werden dadurch wesentlich kürzere und übersichtlichere Abarbeitungssequenzen innerhalb des Monitors benötigt. Durch die Vorselektierung weiß der Monitor schon bei der Ankunft der Nachricht, welche Aktionen nur in Frage kommen können.

**ABBILDUNG 52** Einteilung der Nachrichten in Ereignisse



## 8.4 Zusammenfassung und Ausblick

Ziel der Arbeit war es, für die vorhandenen (das heißt, die zum momentanen Zeitpunkt vorhandenen) ConTracts der APRICOTS-Datenbank einen lauffähigen Monitor zu erstellen. Dazu mußte eine Einarbeitung in das APRICOTS erfolgen, um die Zusammenhänge und die Hintergründe beim Ablauf eines ConTracts zu verstehen und daraus die Anforderungen an die Funktionalitäten des Monitors zu entwickeln. Zur Erstellung des Ablaufgraphen war es weiter notwendig, die theoretischen Grundlagen über Graphen allgemein zu betrachten und im besonderen auf die Struktur der graphischen Abbildung der ConTracts anzuwenden. Dadurch wurde eine Eingrenzung des Problems auf spezielle Graphen (die planaren Graphen) vollzogen und die algorithmische Berechnung konnte von einer speziellen Struktur ausgehen, was die Programmierung erleichterte. Das zeitliche Problem, das bei der dynamischen Erstellung des Layouts auftritt, konnte dadurch nicht ganz ausgeglichen werden, jedoch sind die Antwortzeiten noch im Rahmen erträglicher Grenzen.

Die Lösung des Problems erfolgte nicht in allgemeinster Form, sodaß es bei ConTracts mit Verzweigungen, wie sie unter dem bisherigen Datenbestand noch nicht vorgekommen sind, durchaus zu einer nicht befriedigenden Darstellung kommen könnte. Unter den gegebenen ConTracts der APRICOTS-Datenbank, die momentan verfügbar sind, liefern die Berechnungen jedoch ordentliche Ergebnisse, was das Zeitverhalten und die graphische Anordnung betrifft.

Die Anbindung an das Kommunikationssystem ist wegen der Unvollständigkeit des CS noch nicht ganz abgeschlossen. Auch wurden einige Konstrukte, wie die parforeach-Schleife und Schleifen mit „for“ und „while“ nicht implementiert.

Des weiteren wäre eine interaktive Gestaltung der Fenster, die das freie Monitoring betreffen, noch sinnvoll. Es sollte dabei die Möglichkeit geschaffen werden, interaktiv auf jede Komponente des ConTracts zugreifen zu können. Beispielsweise wäre es sinnvoll, sofortige Reaktion auf Meldungen einer Komponente zuzulassen (die in dessen Informationsfenster erscheinen), wenn diese direkt aus dem Fenster an die betreffende Komponente erfolgen könnten. Es ist aber noch nicht möglich, Komponenten während des Ablaufs eines ConTracts direkt anzusprechen.

Für die Zukunft sinnvoll wäre eine Umkehrung des bisherigen Verfahrens bei der Generierung eines ConTracts, unter Einbeziehen der Oberfläche. Bisher wird ein ConTract durch Ausprogrammieren der Steps und Zusammenfassung dieser im Skript realisiert. Daraus werden statische Daten erzeugt, die in der Datenbank abgelegt werden. Hieraus erzeugt die Oberfläche die graphischen Objekte für den Ablaufmonitor. Umgekehrt würde ein Benutzer eine beliebige Anzahl vorher bereitgestellter Steps graphisch so anordnen, daß diese Anordnung seinem vorgefertigten Plan entspricht. Aus dieser Anordnung könnten die für den Ablauf eines ConTracts benötigten Daten erzeugt werden und in die Datenbank geschrieben werden. Beim Starten eines so entstandenen ConTracts wären dann genauso alle notwendigen Daten zur Kontrolle des Ablaufs verfügbar. Gerade die zur Entwicklung des Ablaufmonitors verwendete Software wäre für eine solche Aufgabe besonders geeignet.

# Literaturverzeichnis


[BuRai90] H. Burkert und B. Raichle: Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext, Universität Stuttgart, Studienarbeit Nr. 859, 1991

[Bilg92] A. Bilger: Modellierung und Simulation betrieblicher Abläufe als ConTracts mit dem APRICOT-System. Universität Stuttgart, Studienarbeit Nr. 1110, 1992.

[DaHsuLa90] U. Dayal, M. Hsu, R. Ladin: Organizing Long-Running Activities with Trigger and Transactions. SIGMOD 1990.

[Har74] F. Harary: Graphentheorie. Deutsche Übersetzung der englischen Originalausgabe, Oldenbourg Verlag 1974.

[Ingr90] Ingres Application Editor User Guide and Language Reference Manual 1990

[Jung87] D. Jungnickel: Graphen, Netzwerke und Algorithmen. B. I. - Wissenschaftsverlag 1987.

[Laube92] D. Laube: Transaktionsorientierte Kommunikationsmechanismen im verteilten Transaktionssystem APRICOTS. Universität Stuttgart Diplomarbeit Nr. 865, 1992.

[Lutz91] R. Lutz: Transaktionsverwaltung auf der Basis von Transaktionsbeziehungsnetzen. Universität Stuttgart, Diplomarbeit Nr. 814, 1991.

[Mans90] N. Mansfield: User Handbook for the X Window System, Addison-Wesley 1990.

[Masman90] I. Masmanidis: Syntax und Compiler einer Sprache zur Beschreibung parallerer und verteilter Abläufe. Universität Stuttgart, Studienarbeit Nr. 897, 1990.

[Masman91] I. Masmanidis: ConTract-Programmierung in Apricots: Sprachbe schreibung und Programmieranleitung. Universität Stuttgart, Arbeitspapier 1991.

[Masman92] I. Masmanidis: Robuste Kontrollflußverwaltung für langlebige und verteilte DB-Anwendungen. Universität Stuttgart, Diplomarbeit Nr. 885, 1992.

[Mey88] B. Meyer: Object-oriented Software Construction, Prentice Hall 1988.

[NisChi88] Nishizeki-Chiba: Planar Graphs, North Holland 1988.

[Oet88] Oetinger, Ralf: Benutzergerechte Software-Entwicklung, Springer Verlag 1988.

[Reu81] A. Reuter: Fehlerbehandlung in Datenbanksystemen, Hanser Verlag 1981.

[Reu89] A. Reuter: ConTracts: A Means for Extending Control Beyond Transaction on Database Systems, Vol 9, No. 4, December 1984, 526-559 ff.

[ReuWäSchw92] A. Reuter, H. Wächter, F. Schwenkreis: Zuverlässige Abwicklung großer verteilter Anwendungen mit ConTracts - Architektur einer Prototypimplementierung. Arbeitsbericht zum DFG Schwerpunktprogram 1992.

[ReuWä91] A. Reuter, H. Wächter: The ConTract Model. Computer Science Report No. 8/91, Sptember 1991.

[ReuWä90] A.Reuter, H.Wächter: Grundkonzepte und Realisierung des ConTract-Modells. Informatik Spektrum (1990) 5: 202-212.

[Somm89] I. Sommerville: Software Engineering - 3rd Edition, Addison-Wesley 1989.

[Tanen88] A. Tanenbaum: Computer networks -2nd Edition, Prentice-Hall 1988.

[Wag90] K. Wagner: Graphentheorie, Band 2, BI-Wissenschafts-Verlag 1990.

[Wahl91] P.Wahl : Interpretative Abwicklung von Prädikat-Transitionsnetzen. Universität Stuttgart, Studienarbeit Nr. 1021, 1991.

[Zellm92] O. Zellmer : Ein Werkzeug zur Definition und Verwaltung transaktionsorientierter Client-Server-Anwendungen. Universität Stuttgart, Studienarbeit Nr. 1111, 1992.