UNIVERSITÄT STUTTGART

Institut für Parallele und Verteilte
Höchstleistungsrechner

Prüfer : *Prof. Dr. A. Reuter*

Betreuer : *Dipl.-Inform. F. Schwenkreis*

Synchronisationsprädikate in APRICOTS

Holger Schwarz

Studienarbeit Nr. 1421

CR-Klassifikation : *C.2.4, E.1, H.2.3, H.2.4*

Begonnen am : *17.10.1994*

Beendet am : *16.4.1995*

Inhaltsverzeichnis

1 ÜBERBLICK 1

1.1 Gliederung der Arbeit 1

2 CONTRACTS UND APRICOTS 3

2.1 ACID-Transaktionen 3

2.2 Das ConTract-Modell 4

2.2.1 Der Aufbau eines ConTracts 5

2.2.2 Recovery 7

2.2.3 Kompensation 7

2.2.4 Synchronisation 8

2.3 APRICOTS 10

2.3.1 Die Benutzeroberfläche 10

2.3.2 Der ConTract-Manager 11

2.3.3 Die Step-Server 11

2.3.4 Die Resource-Manager 11

2.3.5 Der Transaktionsmanager 11

2.3.6 Das Kommunikationssystem 12

3 AUSWAHL DER DATENTYPEN 13

3.1 Basistypen 14

3.1.1 Integer 14

3.1.2 Floating 14

3.1.3 Character 15

3.1.4 Time 15

3.2 Zusammengesetzte Typen 17

3.2.1 String 17

3.2.2 Array 18

3.2.3 Record 18

3.2.4 Set 18

3.3 Nicht berücksichtigte Typen 19

3.3.1 Boolean 19
3.3.2 Aufzählungstypen 19
3.3.3 Zeigertypen 19
3.3.4 Komplexe Datentypen 19

4 DATENTYPEN UND ZUGELASSENE OPERATIONEN 21

4.1 Integer 22

4.2 Floating 23

4.3 Character 23

4.4 Time 23

4.5 String 24

4.6 Array 26

4.7 Record 27

4.8 Set 28

4.9 Typunabhängige Operatoren 30

4.10 Operatoren 31

4.11 Grammatik für Invarianten 33

5 ABBILDUNG AUF BASISTYPEN 35

5.1 String 35

5.2 Array 36

5.3 Record 38

5.4 Set 39

6 VEREINFACHUNG VON INVARIANTEN 41

6.1 Vereinfachungen bei NOT 42

6.2 Vereinfachungen bei AND-Verknüpfung 42

6.2.1 Objekte vom Typ Integer 43
6.2.1.1 Objekt repräsentiert kein Aggregat 43
6.2.1.2 Objekt repräsentiert ein Aggregat 47
6.2.1.3 Vergleich möglicher Zusammenfassungen bei Aggregaten und Nicht-Aggregaten 49
6.2.1.4 Zusätzliche Vergleichsoperatoren 50
6.2.2 Objekte vom Typ Floating 50
6.2.3 Objekte vom Typ Character 51

6.2.4 Objekte vom Typ Time ... 51

6.3 Vereinfachungen bei OR-Verknüpfung ... 52

6.3.1 Objekte vom Typ Integer ... 52

6.3.2 Objekte vom Typ Floating oder Character ... 53

6.3.3 Objekte vom Typ Time ... 54

6.4 Komplexere Invarianten ... 54

7 INVARIANTENVERWALTUNG ... 55

7.1 Position der Verwaltungskomponente ... 55

7.1.1 Zentrale Verwaltungskomponente ... 55

7.1.2 Dezentrale Verwaltungskomponente ... 56

7.2 Verwaltung vereinfachter Invarianten ... 57

7.3 Änderung von Invarianten ... 58

7.4 Daten für die Invariantenverwaltung ... 59

7.5 Datenstrukturen ... 60

7.6 Adressierungsverfahren ... 62

8 AUSBLICK ... 63

9 LITERATURVERZEICHNIS ... 65

Tabellenverzeichnis

TABELLE 1: OPERATOREN UND ZUGELASSENE OPERANDEN 32
TABELLE 2: ZUSAMMENFASSUNGEN BEI NOT-OPERATOR 42
TABELLE 3: MÖGLICHE ZUSAMMENFASSUNGEN FÜR INVARIANTEN ÜBER INTEGEROBJEKTE 44
TABELLE 4: ZUSAMMENFASSUNGEN FÜR NICHT-AGGREGATE 47
TABELLE 5: ZUSAMMENFASSUNGEN FÜR AGGREGATE 49
TABELLE 6: ALLGEMEINE ZUSAMMENFASSUNGEN FÜR INTEGEROBJEKTE 50
TABELLE 7: ZUSAMMENFASSUNGEN FÜR SKALARE TYPEN 51
TABELLE 8: ZUSAMMENFASSUNGEN BEI ZEITLICH BESCHRÄNKTEN INVARIANTEN (AND) 52
TABELLE 9: ZUSAMMENFASSUNGEN BEI OR-VERKNÜPFUNG 53
TABELLE 10: ZUSAMMENFASSUNGEN BEI ZEITLICH BESCHRÄNKTEN INVARIANTEN (OR) 54

Abbildungsverzeichnis

ABBILDUNG 1: BEISPIELSKRIPT „BUSINESS TRIP RESERVATION“ 6
ABBILDUNG 2: INVARIANTEN IM BEISPIELSKRIPT 9
ABBILDUNG 3: ARCHITEKTUR VON APRICOTS 10
ABBILDUNG 4: GRAMMATIK FÜR INVARIANTEN 34
ABBILDUNG 5: ZENTRALE VERWALTUNGSKOMPONENTE 56
ABBILDUNG 6: DEZENTRALE VERWALTUNGSKOMPONENTE 57
ABBILDUNG 7: EINFACHE LISTENSTRUKTUR 60
ABBILDUNG 8: ZWEISTUFIGE LISTENSTRUKTUR 61
ABBILDUNG 9: DATENSTRUKTUR FÜR EINFACHE INVARIANTEN 62

1 Überblick

Große verteilte Systeme stellen erhöhte Anforderungen an Zuverlässigkeit und Fehlertoleranz. Diesen wird das klassische Transaktionskonzept nur begrenzt gerecht. Für Anwendungen mit großer Verweildauer im System wurde darum das ConTract-Modell entwickelt[1]. APRICOTS[2] stellt eine prototypische Implementierung eines ConTracts verarbeitenden Systems dar. Eine wichtige Eigenschaft des Systems ist, daß anwendungsabhängige „Isoaltionsgrade“ programmiert werden können. Diese werden mit Hilfe von Prädikaten über globale Datenobjekte definiert, den sogenannten Invarianten.

In dieser Arbeit soll ermittelt werden, welche Prädikate über Objekte verschiedener Datentypen zugelassen werden sollen. Darüber hinaus werden Aspekte der Verwaltung von Invarianten betrachtet. Hierbei handelt es sich einmal um die Frage, wie Änderungen an Invarianten vorgenommen werden können, ohne daß inkonsistente Zustände auftreten. Zum zweiten werden für die Verwaltung geeignete Datenstrukturen untersucht und Verfahren zur Adressierung globaler Objekte betrachtet. Zunächst aber ein Überblick über den Aufbau der Arbeit.

1.1 Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen für das Verständnis der weiteren Arbeit gelegt. Hierzu gehört eine Motivation für die Einführung des ConTract-Modells, sowie die Erläuterung der wichtigsten Aspekte des Modells. Hinzu kommt eine kurze Beschreibung der Architektur von APRICOTS.

Auf dieses einführende Kapitel folgt in Kapitel 3 die Auseinandersetzung mit der Frage, Objekte welcher Datentypen für Invarianten zugelassen werden sollten.

Kapitel 4 beschäftigt sich dann für die gewählten Datentypen mit den jeweils zugehörigen Operatoren. Dieses Kapitel schließt mit einer Grammatik für Invarianten ab. Das Aussehen der Invarianten ist damit fixiert.

Es kann dann betrachtet werden, wie Invarianten verändert werden können, ohne daß inkonsistente Zustände auftreten. Dies erfolgt einmal in Kapitel 5, wo die Möglichkeiten der Abbildung von Prädikaten über zusammengesetzte Typen auf Prädikate über Basistypen dargestellt werden. Es folgt in Kapitel 6 die Betrachtung, wie verschiedene Invarianten zusammengefaßt werden können. Hier ist es das Ziel, die Überprüfung von Invariantenverletzungen zu erleichtern.

Das Kapitel 7 schließlich beschäftigt sich mit anderen Fragen der Verwaltung von Invarianten. Hierzu gehört die Lokalisierung der Verwaltungskomponente für Invarianten im Gesamtsystem, die Wahl einer Verwaltungsstruktur und das Adressierungsverfahren, über das globale Objekte angesprochen werden können.

---

[1] Vgl. [WäRe91]

[2] A PRototype Implementation of the COnTract System

2 ConTracts und APRICOTS

Dieses Kapitel stellt die notwendigen Grundlagen für das Verständnis der weiteren Kapitel bereit. Hierzu werden die Grenzen des klassichen Transaktionskonzeptes aufgezeigt, sowie das ConTract-Modell und die prototypische Implementierung APRICOTS vorgestellt.

2.1 ACID-Transaktionen

ACID-Transaktionen stellen das in den 70-er Jahren entwickelte Transaktionskonzept für Datenbanksysteme dar. Ausgangspunkt für dieses Konzept war ein Szenario, in dem mehrere parallele oder quasi-parallele Anwendungen konkurrierend auf die Elemente einer gemeinsamen Datenbasis zugreifen. Diese Datenbasis enthielt eine große Anzahl einfach aufgebauter Elemente, die zudem alle gleich strukturiert waren. Diese Voraussetzungen führten zu den folgenden Anforderungen an das Transaktionskonzept:

1. Synchronisation konkurrierender Zugriffe der verschiedenen Anwendungen auf gemeinsame Daten.
2. Nach der erfolgreichen Ausführung von Datenbankoperationen müssen stabile Ergebnisse hinterlassen werden.
3. Die Zusammenfassung logisch zusammenhängender Datenbankoperationen zu einer ausführbaren Einheit muß ermöglicht werden.

ACID-Transaktionen erfüllen diese Anforderungen. Sie schaffen kleine überschaubare Einheiten. So entsteht ein anwendungsunabhängiger Kontrollmechanismus für Datenbanksysteme. Jede ACID-Transaktion weist die folgenden vier Eigenschaften auf:

Atomicity : Jede Transaktion wird entweder komplett oder gar nicht ausgeführt. Sie gilt als kleinste Ausführungseinheit in einem Datenbanksystem.

Consistency : Jede Transaktion findet die Datenbank in einem konsistenten Zustand vor und hinterläßt wieder einen solchen Zustand. Dieser wird entweder mit der erfolgreichen Ausführung der Transaktion oder mit deren vollständigem Zurücksetzen erreicht.

Isolation : Die Zustandsübergänge der Datenobjekte während der Bearbeitung durch eine Transaktion werden vor den anderen Transaktionen bis zum Ende der bearbeitenden Transaktion verborgen. Es wird nur der Endzustand der Datenobjekte sichtbar. So entsteht ein virtueller Einbenutzerbetrieb.

Durability : Alle von einer Transaktion durchgeführten Änderungen an Datenobjekten sind stabil gespeichert.

Die Grenzen dieses Transaktionskonzeptes werden in [Gray81] ausführlich diskutiert. Hier sollen lediglich zwei wichtige Aspekte herausgegriffen werden. Voraussetzung für den sinnvollen Einsatz von ACID-Transaktionen sind unter anderem:

- Eine Transaktion berührt wenige Elemente der Datenbasis. Hierdurch wird garantiert, daß sich beim Abbruch der Transaktion die notwendige Recovery auf eine überschaubare Zahl von Datenobjekten beschränken kann.
- Eine Transaktion hat eine kurze Verweildauer im System. Dies führt zu einem geringen Konfliktpotential bei konkurrierenden Transaktionen, da Zugriffskonflikte nur kurzzeitig bestehen.

Diese Voraussetzungen werden von Standardanwendungen voll und ganz erfüllt. Beispiele solcher Anwendungen sind: Reservierungssysteme, Lager- und Buchhaltungssysteme, u.a. Diesen stehen die sogenannten „Nicht-Standard-Anwendungen“ gegenüber, wie z.B. Büroautomatisierung, Fertigungssteuerung und -kontrolle, Simulationen oder CAD-, CAM- und CIM-Systeme. Eine Realisierung dieser Anwendungen mit Hilfe von ACID-Transaktionen führt zu Transaktionen mit einer großen Verweildauer im System. Während die Transaktion im System ist, werden viele Zustandsänderungen auf einer großen Anzahl von Datenobjekten vorgenommen. Alle diese Objekte müssen bis zum Ende der Transaktion vor Zugriffen durch andere Transaktionen geschützt werden. Eine unerträgliche Performance des transaktionsverarbeitenden Systems ist die Folge. Hinzu kommt, daß eventuell notwendige Recoverymaßnahmen bei Systemausfällen o.ä. ebenso lange dauern, wie die ursprüngliche Transaktion. Für die Lösung dieser Probleme mit langlebigen Transaktionen erweisen sich zwei Eigenschaften der ACID-Transaktionen als nachteilig:

- Sie sind kontextfrei, d.h. der Kontext außerhalb von Transaktionsgrenzen kann in der Transaktion nicht berücksichtigt werden. Kontrollfluß zwischen Transaktionen kann somit nicht dargestellt werden.
- Sie können nicht verzweigt oder verschachtelt werden. Dies stellt vor allem bei großen Anwendungen und in verteilten Systemen ein Problem dar.

Dies sind die Ansatzpunkte für das ConTract-Modell, das eine formale Basis zur Definition und Kontrolle langlebiger komplexer Bearbeitungen liefert.

2.2 Das ConTract-Modell

Definition eines ConTracts[3] :

Ein ConTract ist eine konsistente und fehlertolerante Ausführung einer beliebigen Sequenz vordefinierter Aktionen (Steps), gemäß einer explizit spezifizierten Kontrollflußbeschreibung (Skript).

Die Grundidee des Modells besteht darin, umfangreiche Anwendungen aus kurzen ACID-Transaktioen aufzubauen und einen anwendungsunabhängigen Systemdienst zur Verfügung zu stellen, der die Ausführung kontrolliert. Hierdurch wird die Ausführung selbst zu einem wiederherstellbaren Objekt und nicht nur die Datenobjekte, wie dies bei ACID-Transaktionen der Fall ist. Alle Aufgaben der Recovery, der Regelung des Kontrollflusses, der Synchronisation von Zugriffen auf gemeinsame Ressourcen sollen vor dem Anwendungsprogrammierer verborgen werden.

[3] Vgl. [WäRe91], S. 9

2.2.1 Der Aufbau eines ConTracts

Ein ConTract weist einen zweistufigen Aufbau auf, bestehend aus mehreren Steps und einem Skript.

Steps

Dies sind die elementaren Teilschritte einer Anwendung. Ein Step umschließt logisch zusammengehörende Aktionen. Ausgeführt wird ein Step unter Transaktionsschutz, eventuell zusammen mit anderen Steps. Zu den Aufgaben eines Steps gehört auch die Verwaltung des relevanten Kontextes, sowie das Aufstellen von Invarianten.

Skript

Das Skript beschreibt den Kontrollfluß der Anwendung. Hierzu müssen folgende Elemente enthalten sein:

- Eine Skript-Beschreibung zur Kontrollflußbeschreibung. Damit wird festgelegt, unter welchen Bedingungen die Steps in welcher Reihenfolge ablaufen sollen.
- Eine Kontext-Definition für persistente Variablen. Mit deren Hilfe wird der aktuelle Verarbeitungszustand des ConTracts festgehalten.
- Eine Transaktionsbeschreibung, die Transaktionsgrenzen und Abhängigkeiten zwischen Transaktionen festlegt.
- Eine Invariantenbeschreibung, die die Synchronisation parallel ablaufender Steps durch Konsistenzprüfungen ermöglicht.
- Eine Kompensationsbeschreibung, die zu jedem Step eine Kompensationsaktion bereitstellt. Hierdurch wird im Fall eines Abbruchs Konsistenzerhaltung gewährleistet.

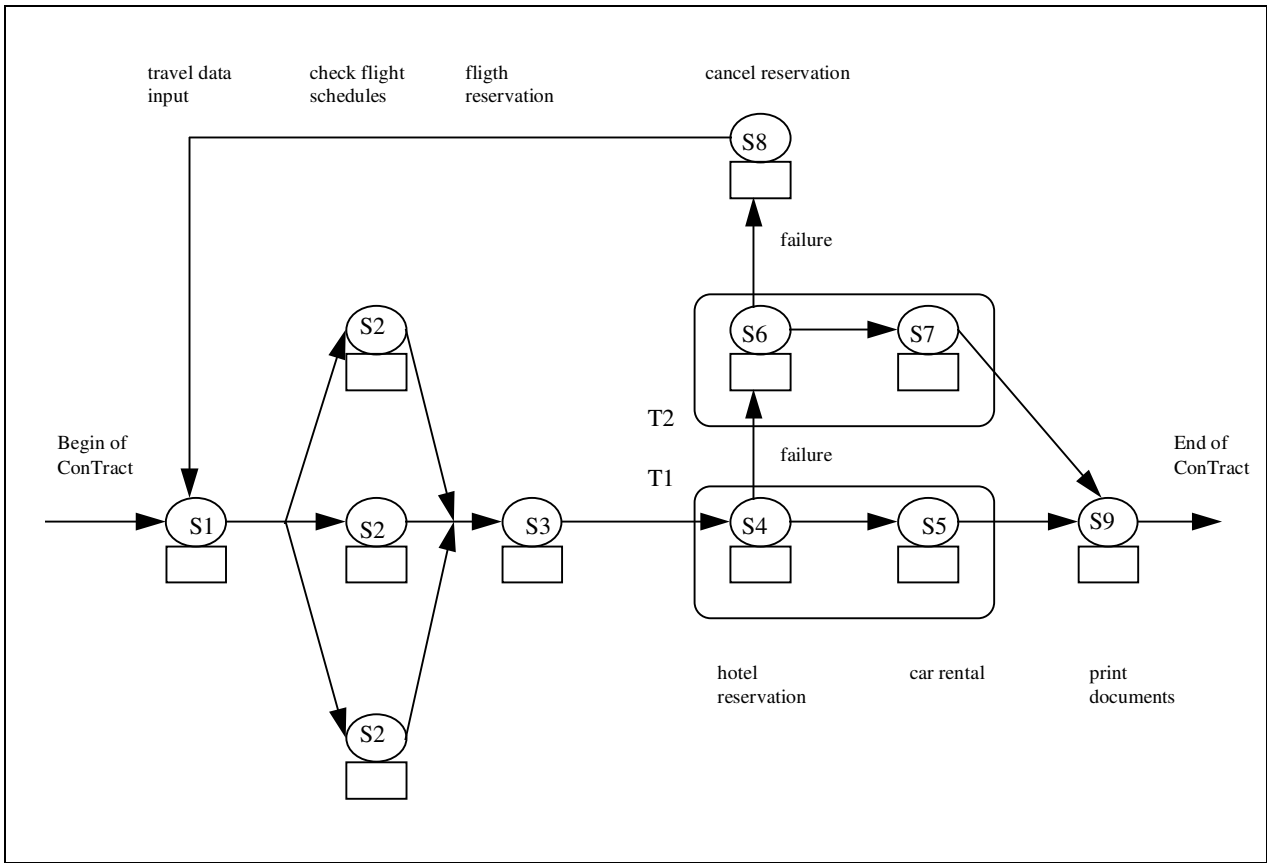

Abbildung 1: Beispielskript „Business Trip Reservation“

Ein Beispiel für ein Skript in einer graphischen Repräsentation ist in Abbildung 1 dargestellt[4]. Gezeigt ist eine einfache Form eines Reisebuchungssystems. Die einzelnen Steps sind durch Kreise dargestellt. Die Kompensationsaktionen durch Rechtecke. Die Pfeile zeigen den Kontrollfluß in Abhängigkeit davon, ob ein Step erfolgreich war oder nicht. Standardmäßig wird jeder Step innerhalb einer Transaktion ausgeführt. Ausnahmen sind hier die Steps S_4 und S_5, sowie S_6 und S_7. Kontextdefinition und Invariantenbeschreibung werden in dieser Darstellung nicht berücksichtigt.

Der zweistufige Aufbau eines ConTracts führt zu einem Programmiermodell, das ebenfalls zwei Stufen aufweist. Das Programmieren der Steps wird von der Erstellung des Skripts für den Kontrollfluß getrennt. Die Erarbeitung der Steps und des Skripts kann von unterschiedlichen Personen ausgeführt werden.

Sowohl für den Step- als auch für den Skriptprogrammierer wird die Arbeit erleichtert. Bei der Erstellung der Steps müssen Dinge, wie Parallelität, Kommunikation, verteilte Ressourcen, Synchronisation und Recovery nicht berücksichtigt werden. Der Skriptprogrammierer kann dagegen auf vorgefertigte Steps zurückgreifen und ist lediglich dafür verantwortlich, deren korrekte Abfolge festzulegen. Die interne Realisierung der Steps muß hierfür nicht berücksichtigt werden.

Nach der Darstellung des Aufbaus von ConTracts soll nun eine kurze Beschreibung der drei Bereiche Recovery, Kompensation und Synchronisation erfolgen, soweit dies zum Verständnis der weiteren Arbeit notwendig erscheint.

[4] Das Beispiel stammt aus [WäRe91]

2.2.2 Recovery

Das Ziel bei ConTracts ist, die Recovery so zu gestalten, daß Veränderungen am System, Kommunikationsfehler, Knotenausfälle und andere Fehler nicht zu einem undefinierten Zustand einer Anwendung führen. Bei klassischen Transaktionen ist dies der Fall, wenn die Anwendung nicht selbst besondere Vorkehrungen trifft.

- Ein gewöhnlicher Betriebssystemprozeß ist nach einem Systemausfall verloren. Der Anwender muß wissen, welche Anwendungen betroffen sind, in welchem Zustand die Anwendung abgebrochen wurde und wie erneut aufgesetzt werden kann.
- Ein Transaktionssystem stellt nach einem Systemausfall einen konsistenten Zustand dadurch wieder her, daß alle nicht beendeten Transaktionen auf ihren Anfangszustand zurückgesetzt werden. Bei langdauernden Aufgaben geht hierdurch zu viel Arbeit verloren.

Ein ConTracts verarbeitendes System soll demgegenüber versuchen, nach der Unterbrechung eines ConTracts einen konsistenten Zustand wiederherzustellen, der eine Fortsetzung entsprechend der Festlegungen im Skript ermöglicht. Hierbei soll möglichst wenig, der bereits erfolgreich abgeschlossenen Arbeit verloren gehen. Diese Art der Recovery wird als Forward-Recovery bezeichnet. Voraussetzung hierfür ist, daß jegliche Zustandsinformation, die für die Ausführung eines Steps von Bedeutung ist, so festgehalten wird, daß ein Wiederherstellen möglich ist. Diese Daten, die den Zustand eines ConTracts beschreiben werden als dessen Kontext bezeichnet. Für das Wiederaufsetzen eines abgebrochenen ConTracts müssen folgende Informationen vorhanden sein[5] :

- Der globale Systemzustand.
- Der lokale Zustand des ConTracts, d.h. alle Objekte wie z.B. Programmvariablen, die von mehr als einem Step genutzt werden, aber nicht in der Datenbank gespeichert sind.
- Der globale Verarbeitungszustand der Anwendung. So muß z.B. festgehalten werden welche Steps bereits ausgeführt wurden und welche nicht.

2.2.3 Kompensation

Datenobjekte können nicht für die Dauer kompletter ConTracts vor dem Zugriff durch konkurrierende ConTracts geschützt werden. Diese Zugriffe wären dann erst nach dem Ende des gerade zugreifenden ConTracts wieder möglich, was zu einem erheblichen Performance-Verlust und zu einer hohen Zahl von Abbrüchen zur Deadlockbeseitigung führen würde. Datenobjekte müssen also bereits im Verlauf eines ConTracts wieder freigegeben werden. Dies geschieht immer am Ende einer Transaktion, die einen oder mehrere Steps umfaßt. Danach können andere ConTracts diese Datenobjekte verändern. Trotzdem soll jederzeit ein Rücksetzen des ConTracts oder von Teilen davon möglich sein. Dies setzt voraus, daß die Effekte von Steps, die bereits abgeschlossen sind, noch rückgängig gemacht werden können. Hier kann die bei klassischen Transaktionen verwendbare Methode des Zurücksetzens auf den Zustand vor der Transaktionsausführung nicht weiterhelfen. Dieses Vorgehen würde dazu führen, daß alle nach dem zurückzusetzenden Step an einem Datenobjekt vorgenommenen Veränderungen ebenfalls zurückgenommen würden. Da diese aber im Allgemeinen erhalten bleiben sollen ist eine andere Methode angebracht. Es werden Kompensationsaktionen eingeführt. Eine solche Kompensation muß für jeden Step existieren. Sie führt ein semantisches Undo der von einem Step vorgenommenen Veränderungen an globalen Datenobjekten durch.

---

[5] Vgl. [WäRe91], S. 20

Beispiel: Von einem Step wird der Inhalt eines Objekts durch die Operation +10 von 10 auf 20 erhöht. Als Kompensation reicht es nicht aus einfach den Wert 10 zurückzuschreiben, sondern es muß die inverse Operation -10 ausgeführt werden. Diese setzt den Step auch dann korrekt zurück, wenn sich der Wert des Objekts inzwischen auf z.B. 50 erhöht hat.

Bei der Verwendung von Kompensationsaktionen werden folgende Anforderungen an das System gestellt, damit ein korrektes Funktionieren des Mechanismus gewährleistet ist:

- Für jeden Step im Skript muß genau ein Step zur Kompensation existieren.
- Am Ende eines Steps müssen alle Daten vorliegen, die für dessen Kompensation notwendig sind. Aktuelle Werte globaler Objekte müssen bei Bedarf im Kontext gespeichert werden.
- Für alle von einem Step angefaßten Objekte, die auch für dessen Kompensation von Bedeutung sind muß deren Existenz bis zum Ende des ConTracts sichergestellt sein.
- Soll ein ConTract komplett zurückgesetzt werden, so dürfen auf anderen Knoten ablaufende Steps keine weiteren Steps mehr anstoßen. Alle Steps müssen abgebrochen oder kompensiert werden, sobald die Entscheidung über das Rücksetzen des ConTracts auf allen betroffenen Knoten bekannt ist. Vorher kann ein ConTract nicht abgeschlossen werden.
- Kompensationsaktionen können ebenfalls abgebrochen werden. Für diesen Fall ist die Durchführung der Kompensation solange zu versuchen, bis sie erfolgreich verlaufen ist.
- Läßt sich eine Kompensation auch nach der als Obergrenze festgelegten Anzahl von Versuchen nicht durchführen, so muß dieses Problem von außen durch einen Systemadministrator behandelt werden.

2.2.4 Synchronisation

Da innerhalb eines ConTracts Sperren auf Objekte nicht erst am Ende des ConTracts, sondern am Ende eines jeden Steps[6], freigegeben werden, muß ein zusätzlicher Mechanismus zur Synchronisation von gemeinsam auf Datenobjekte zugreifenden Steps eingeführt werden. Hierzu werden über Stepgrenzen hinaus keine Sperren gehalten, sondern es werden sogenannte Invarianten eingeführt. Mit deren Hilfe kann jeder Step angeben, welche Bedingungen an ein Objekt zu stellen sind, so daß eine korrekte Ausführung der weiteren Steps im ConTract bzw. des zugehörigen Steps zur Kompensation möglich ist. Wird die Einhaltung dieser Invarianten vom System überprüft, so ist eine für die Anwendung ausreichende Isolation sichergestellt.

Es können zwei Arten von Invarianten unterschieden werden: Ausgangsinvarianten legen Eigenschaften von Datenobjekten fest, die nach der Ausführung eines Steps gelten sollen. Eingangsinvarianten legen Eigenschaften von Datenobjekten fest, die für die korrekte Ausführung eines Steps als Voraussetzung gelten.

[6] Hier wird vereinfachend angenommen, daß jeder Step innerhalb einer ACID-Transaktion abläuft.

Für das in Abbildung 1 gezeigte Beispielskript sind einige mögliche Invarianten in Abbildung 2 aufgeführt[7]. Hier wird die Notation aus [WäRe91] verwendet. Diese Arbeit beschäftigt sich, wie bereits erwähnt, mit der Erstellung einer Grammatik für die Invarianten. Nach dieser Grammatik ergeben sich leichte Unterschiede in der Syntax.

In Abbildung 2 ist die Ausgangsinvariante für Step S_1 *„budget > cost_limit“* angegeben. Hierdurch wird ausgesagt, daß das zur Verfügung stehende Budget größer sein muß als die Obergrenze für die Reisekosten. Dies muß auch noch gelten, wenn ein Flug gebucht wird. Darum hat Step S_3 dieselbe Invariante als Einganginvariante. Innerhalb des Steps S_3 wird der Preis für das Flugticket vom Budget abgezogen. Für das Budget muß am Ende des Steps dann nur noch die Ausgangsinvariante *„budget > cost_limit - ticket_price“* gelten. Soweit zu diesem Beispiel.

Es gibt nun drei Strategien, mit den Invarianten umzugehen:

1. Der Wert eines Objekts wird mit Hilfe von Sperren oder anderen semantischen Synchronisationstechniken[8] „eingefroren“. Damit müssen Sperren über Transaktionsgrenzen hinaus gehalten werden.

2. Bei jedem Zugriff auf Objekte, für die eine Invariante existiert, wird überprüft, ob nach der Veränderung des Objekts eine Verletzung der Invariante auftritt. Ist dies der Fall, so wird der Zugriff auf das Objekt verweigert. Damit wird eine Verletzung von Invarianten ausgeschlossen.

3. Die dritte Technik verhindert keine Zugriffe auf Objekte, auch wenn dadurch Invarianten verletzt werden. Die Überprüfung, ob eine Invariante noch eingehalten wird findet erst statt, wenn ein Step ausgeführt wird, der diese als Eingangsinvariante aufweist. Bei Verletzung der Eingangsinvariante muß eine Strategie zur Auflösung des Konflikts gefunden und im Skript angegeben werden.

Für die weitere Arbeit wird gemäß der zweiten Strategie angenommen, daß mit jeder Änderung von Datenobjekten überprüft wird, ob hierdurch eine Verletzung von Invarianten entsteht.

```
CONTRACT Business_Trip_Reservations
...
SYNCHRONISATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
S1:       EXIT_INVARIANT( budget > cost_limit )
S3:       ENTRY_INVARIANT(   (budget > cost_limit) AND
                             (cost_limit > ticket_price) );
          CONFLICT_RESOLUTION: S8: Cancel_Reservation(...);
          EXIT_INVARIANT( budget > cost_limit – ticket_price );
S4, S6:   ENTRY_INVARIANT( hotel_price < bidget ),
          CONFLICT_RESOLUTION: S110:
          Call_Manager_to_Increase_Budget(...);
S5, S7:   ENTRY_INVARIANT( car_price < budget ),
          CONFLICT_RESOLUTION: S110:
          Call_Manager_to_Increase_Budget(...);
END_ SYNCHRONISATION_INVARIANTS_&_CONFLICT_RESOLUTIONS
...
END_OF_CONTRACT Business_Trip_Reservations
```

Abbildung 2: Invarianten im Beispielskript

[7] Vgl. [WäRe91], S. 31

[8] Vgl. Escrow Locking in [GrRe93], S. 432

2.3 APRICOTS

APRICOTS ist die prototypische Implementierung eines ConTracts verarbeitenden Systems. Die Architektur des Prototypen ist in Abbildung 3 dargestellt. Die Darstellung orientiert sich an [Schü93][9]. In dieser Abbildung sind alle für das weitere Verständnis dieser Arbeit notwendigen Module des Prototypen dargestellt. Durch den modularen Aufbau des Gesamtsystems wird eine große Flexibilität erreicht. So können leicht neue Module hinzugefügt und einzelne Module durch eine veränderte Version ersetzt werden. Für die verschiedenen Module soll nun kurz beschrieben werden, welche Funktionalität sie aufweisen, welche Schnittstellen sie besitzen und wie die Zusammenarbeit mit anderen Modulen prinzipiell gestaltet ist.

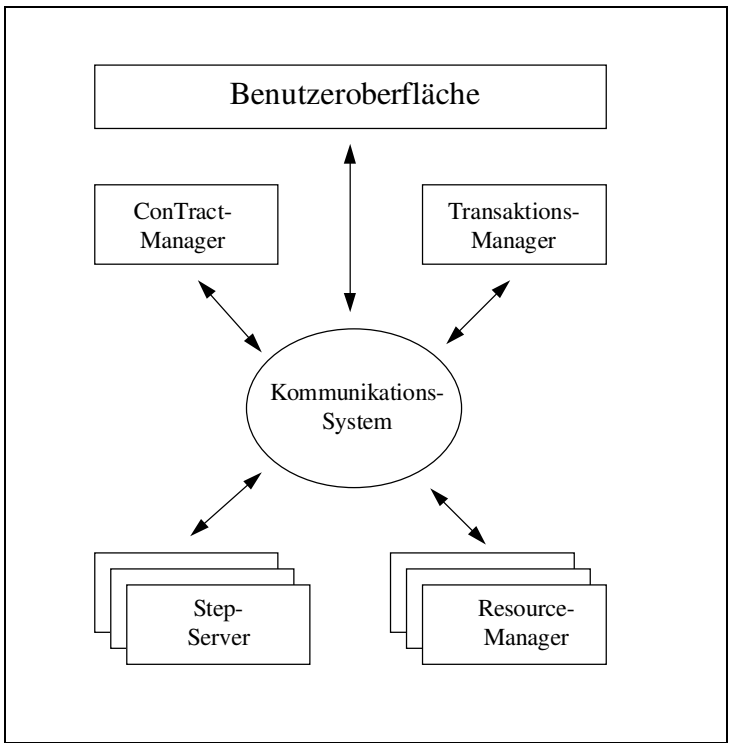

Abbildung 3: Architektur von APRICOTS

2.3.1 Die Benutzeroberfläche

Die Aufgabe der Benutzeroberflächen ist es, dem Benutzer statisches und dynamisches Konfigurationsmanagement, sowie dynamische Prozeßverwaltung zu ermöglichen[10]. Hierzu kann eine graphische Visualisierung des ConTract-Skripts genutzt werden. Mit Hilfe des statischen Konfigurationsmanagements kann der Benutzer Services, Resource-Manager und andere Komponenten definieren, modifizieren und eliminieren. Aufgabe des dynamischen Konfigurationsmanagements ist es die dynamische Server-Konfigurationen und Abhängigkeiten zur Laufzeit sichtbar zu machen. Das Starten, Stoppen und Migrieren von Komponenten kann der Benutzer mit Hilfe der dynamischen Prozeßverwaltung erledigen.

---

[9] Vgl. [Schü93], S. 15-19

[10] Vgl. [Eitl92] und [Zell92]

2.3.2 Der ConTract-Manager

Die Aufgabe dieses Moduls ist die korrekte und zuverlässige Abwicklung eines ConTracts. Ausgehend vom vorliegenden Skript muß der Kontrollfluß interpretiert werden. Steps und Transaktionsgrenzen sind zu initiieren. Darüber hinaus hat der ConTract-Manager für die Weiterführung eines ConTracts nach einem Systemausfall oder ähnlichen Fehlern zu sorgen. Hinzu kommt das Kontextmanagement und das Invariantenmanagement. Beim Kontextmanagement muß der für die Ausführung eines Steps notwendige Kontext bereitgestellt und aktualisiert werden. Das Invariantenmanagement umfaßt die Überprüfung der Eingans- und Ausgangsinvarianten, wann immer dies notwendig ist[11] .

Zwischen ConTract-Manager und Step-Servern findet ein bidirektionaler Nachrichtenfluß statt.

2.3.3 Die Step-Server

Die Step-Server verwalten die Implementierung der Steps, die für die Ausführung von ConTracts notwendig sind. Die Ausführung von Steps wird durch den ConTract-Manager angestoßen. Damit erfüllt ein Step-Server aus Sicht des ConTract-Managers die Rolle eines Servers. Die Steps selbst nehmen bei der Ausführung die Dienste verschiedener Resource-Manager in Anspruch. Für diese stellt der Step-Server damit einen Client dar.

Zwischen den Step-Servern und dem ConTract-Manager findet ebenfalls bidirektionale Kommunikation statt.

2.3.4 Die Resource-Manager

Die Aufgabe von Resource-Managern ist die Verwaltung globaler Objekte. Dies können z.B. Fertigungsmaschinen, Datenbanken, Drucker oder auch Scanner sein. Gegenüber den Step-Servern sind die Resource-Manager in der Rolle von Servern, denen sie Dienste zur Verfügung stellen. Es kann viele verschiedene Resource-Manager geben, die in Klassen eingeteilt werden können. Eine solche Klasse kann z.B. Druckerdienste anbieten. Diese Klasse kann dann Resource-Manager für Laserdrucker und Resource-Manager für Matrixdrucker enthalten.

Die Kommunikation zwischen den Resource-Managern und ihrer Umgebung ist bidirektional, d.h. der Resource-Manager wird - mit Parametern versorgt - aufgerufen und beendet seinen Aufruf mit einer Parameterrückgabe bzw. einer Statusmeldung.

2.3.5 Der Transaktionsmanager

Die Kontrolle und Abwicklung von Transaktionen ist Aufgabe des Transaktions-Managers. Hierzu wird das verteilte 2-Phasen-Commit-Protokoll verwendet. In einem ConTract-Skript können auch Beziehungen zwischen verschiedenen Transaktionen aufgeführt sein. So kann z.B. der Abbruch einer Transaktion den Abbruch anderer, parallel ablaufender Transaktionen nach sich ziehen. Der Transaktions-Manager ist für die Verwaltung solcher Beziehungen zuständig.

---

[11] Vgl. hierzu 2.2.4

2.3.6 Das Kommunikationssystem

Das Kommunikationssystem stellt die Verbindung zwischen den einzelnen Modulen her. Hierzu müssen Nachrichten korrekt und zuverlässig vermittelt und versandt werden. Die Hauptaufgaben des Kommunikationssystems sind die Installation von Kommunikationskanälen, die Empfänger- und Adreßbestimmung und die Nachrichtenversendung zur Dienstinitiierung.

3 Auswahl der Datentypen

Invarianten legen bestimmte Eigenschaften für Objekte fest, die für einige Zeit - maximal bis zum Ende eines ConTracts - erfüllt sein müssen. Jedem dieser Objekte muß ein Datentyp zugeordnet sein. In diesem Kapitel wird dargestellt, welche Datentypen in Invarianten berücksichtigt werden sollten. Neben einer kurzen Beschreibung der einzelnen Typen steht jeweils eine Begründung dafür, warum es sinnvoll ist, sie für Invarianten vorzusehen. Darüber hinaus ist angegeben, wie Konstanten dieses Typs aussehen. Hierzu wird die zugehörige Syntax aufgeführt. Die für einen bestimmten Datentyp zugelassenen Operatoren sind in Kapitel 4 beschrieben.

Zur besseren Übersicht sind die Datentypen hier nach ihrer Komplexität klassifiziert und in drei Gruppen eingeteilt:

1. Basistypen

   Hierbei handelt es sich um Datentypen, die intern nicht weiter strukturiert sind. Dies bedeutet, daß ein Objekt eines Basistyps keine Teilobjekte aufweist, auf die separat zugegriffen werden kann. Beispiele für Basistypen sind Ganzzahltypen, Gleitkommatypen, Zeichentypen und Aufzählungstypen.

2. Zusammengesetzte Typen

   Hierunter fallen alle Typen, die eine interne Struktur aufweisen. Diese Struktur wird für ein Objekt eines solchen Typs einmal festgelegt und danach nicht wieder verändert. Beispiele für zusammengesetzte Typen sind Vektortypen, Strukturtypen und Mengen.

3. Komplexe Typen

   Als komplexe Datentypen seien alle Typen verstanden, bei denen sich Struktur und Größe während der Laufzeit verändern können. Dies kann z.B. daraus resultieren, daß Form und Ausprägung der Daten von vornherein nicht bekannt sind. Typische Beispiele für komplexe Datentypen sind Listen und Bäume.

Da es schon aus Gründen der Komplexität nicht sinnvoll ist, alle Typen, die in eine der drei Gruppen fallen, für die Invarianten vorzusehen, muß eine Auswahl getroffen werden. Die Wahl der letztlich zugelassenen Typen orientiert sich an den Typsystemen der Programmiersprache C[12] und der Datenbanksprache SQL. Die wesentlichen Gründe hierfür sind:

- C ist eine bekannte und häufig eingesetzte Sprache, so daß das Typsystem von C als weitgehend bekannt vorausgesetzt werden kann. Dies erleichtert dem Anwendungsprogrammierer von ConTracts die Formulierung der Invarianten.
- Typen von C existieren auch in den meisten anderen höheren Programmiersprachen. Es handelt sich somit um eine Art „Standardtypsystem“. Dies ermöglicht es auch Programmierern, die weniger mit C vertraut sind, schnell die notwendigen Invarianten zu formulieren.

[12] Alle Angaben zur Programmiersprache C stützen sich auf [HaSt91]

- ConTracts werden häufig auf Daten operieren, die in einer relationalen Datenbank gespeichert sind. Die Tatsache, daß wesentliche Datentypen die in Invarianten zugelassen sind auch in SQL existieren erleichtert eine Abbildung der Invarianten oder Teilen davon auf SQL-Anfragen.

3.1 Basistypen

In diesem Abschnitt werden sämtliche Datentypen eingeführt, die keine interne Struktur aufweisen. Objekte der hier betrachteten Typen besitzen demnach keine Komponenten, auf die separat zugegriffen werden kann. Als Basistypen werden die Typen Integer, Floating, Character und Time berücksichtigt.

3.1.1 Integer

Hierbei handelt es sich um einen vorzeichenbehafteten Ganzzahltyp. Für diesen wird im weiteren angenommen, daß er dem Typ int von C entspricht. Als minimaler Wertebereich ist der Bereich von -32767 bis 32767 zugrunde gelegt. Dieser Typ ist der einzige hier betrachtete Ganzzahltyp. Da sich die Ergebnisse der Arbeit problemlos auf alle anderen Ganzzahltypen der Sprache C übertragen lassen ist deren separate Betrachtung nicht notwendig. Um anzudeuten, daß es sich hier um einen allgemeinen Ganzzahltyp handelt und nicht um den speziellen Typ int von C wird der Bezeichner Integer benutzt.

Eine Konstante vom Typ Integer besteht aus einem Vorzeichen (optional) und einer der Ziffern 1 bis 9 gefolgt von einer beliebigen Anzahl beliebiger Ziffern. Auf die Möglichkeiten zur Angabe von Oktal- und Hexadezimalkonstanten und eines Suffix für andere Ganzzahltypen wurde zugunsten der Einfachheit der entstehenden Grammatik verzichtet.

Syntax:[13]

integer constant	=	[sign] (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) { digit }
digit	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

3.1.2 Floating

Floating bezeichnet den in dieser Arbeit betrachteten vorzeichenbehafteten Gleitkommatyp. Dieser Typ orientiert sich an float von C. Dies gilt sowohl für die Genauigkeit als auch für den minimalen Wertebereich von -10^{-37} bis 10^{+37}. Wie bei den Ganzzahltypen wird Floating stellvertretend für alle Gleitkommatypen betrachtet. Aus diesem Grund wird auch hier mit Floating ein Bezeichner gewählt, der in C selbst nicht existiert.

Konstanten vom Typ Floating bestehen aus einem Vorzeichen und mindestens einer Ziffer vor oder nach dem Dezimalpunkt. Hierauf folgt der vorzeichenbehaftete Exponent mit beliebig vielen Ziffern.

Syntax:

floating constant	=	[sign] digit { digit } [.{ digit }] [E [sign] digit { digit }] | [sign] { digit } . digit { digit } [E [sign] digit { digit }]
digit	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

[13] Erläuterungen zur Darstellung der Syntax befinden sich in Abschnitt 4.11

3.1.3 Character

Hierbei handelt es sich um den Typ für vorzeichenlose Einzelzeichen, der dem Typ char von C entspricht. Die Ergebnisse für diesen Typ gelten aber auch für die Typen signed char und unsigned char, so daß diese hier nicht weiter betrachtet werden.

Eine Konstante besteht aus dem in einfachen Anführungszeichen eingeschlossenen Zeichen. Hierbei kann die Angabe des Zeichens entweder direkt erfolgen oder indirekt über einen Escape-Mechanismus.

Syntax:

character constant	=	‘ character ‘ | ‘ escape ‘
escape	=	\ [n | t | b | r | f | v | \ | ‘ | „ | a | ?] | \ octal | \ hex
octal	=	{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 }
hex	=	x { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F }

3.1.4 Time

Dieser Datentyp, der weder in C noch im SQL-Standard enthalten ist, stellt eine Kombination aus Datums- und Zeitangabe dar. Grundsätzlich kann eine Zeitangabe auf zweierlei Art und Weise erfolgen[14]:

1. Es erfolgt die Angabe eines Zeitpunkts. Eine solche Zeitangabe kann z.B. *1.1.1995* sein
2. Die Zeitangabe stellt ein Zeitintervall dar. Es sind dann Angaben, wie z.B. *(1.1.1995 , 31.1.1995)* denkbar.

Für den hier betrachteten Typ Time wird die erste Variante gewählt. Dies hat folgende Gründe:

- In den typischen Anwendungen für ConTracts treten eher Zeitpunkte als Intervalle auf. Beispiele sind Lieferzeitpunkte, Bestellzeitpunkte, Abflugzeiten und Geburtstage.
- Betrachtet man die Relationen, in denen zwei Zeitangaben stehen können, so unterscheiden sich diese für Zeitpunkte und Zeitintervalle. Für Zeitpunkte gibt es die drei Basisrelationen =, <, und >. Bei Zeitintervallen existieren dagegen 13 Basisrelationen. Dies führt zu 2^{13}=8192 zusammengesetzten Relationen, die zwischen zwei Zeitintervallen bestehen können, während zwischen zwei Zeitpunkten lediglich 2^3=8 Relationen möglich sind[15]. Diese Relationen existieren ebenso für Integer- und Floating-Objekte. Die Wahl von Zeitpunkten für den Typ Time ermöglicht also für die Darstellung der Relationen den Rückgriff auf die wenigen vom Typ Integer bekannten Vergleichsoperatoren. Die Grammatik für Invarianten kann hierdurch einfacher gehalten werden

[14] Vgl. [RoWa93], S. 3-5

[15] Diese acht Relationen sind: ∅, ≤, <, =, >, ≥, ≠, ?. Hierbei bezeichnet ∅ die Tatsache, daß keine Basisrelation gilt, während ? für das Gelten aller Basisrelationen steht. Praktische Bedeutung haben in erster Linie die anderen sechs Relationen.

Neben der grundsätzlichen Festlegung auf die Betrachtung von Zeitpunkten soll nun begründet werden, warum die Einführung eines separaten Typs für Zeitangaben überhaupt sinnvoll ist. Hierfür sprechen zwei Gründe:

1. Einfacher Umgang mit Objekten, die Zeitangaben repräsentieren

   Mit der expliziten Einführung eines Datentyps für die Zeit können Invarianten über Zeit-Objekte wesentlich vereinfacht werden. Dies soll hier am Beispiel einer möglichen Invariante gezeigt werden.

   Beispiel: A < 11.3.1995

   Ohne den Typ Time gäbe es zwei Möglichkeiten das Objekt A darzustellen:

   - Abbildung auf einen Integer-Wert

     In diesem Fall müßte die Invariante z.B. $A < 1073987$ lauten, wobei die Zahl 1073987 das Datum 11.3.1995 repräsentiert. Die Bedeutung ist hier nicht mehr direkt abzulesen, da Integer-Wert und Datum nach einem bestimmten Algorithmus ineinander umgerechnet werden müssen.

   - Abbildung auf einen Record

     Hier ist A ein Objekt vom Typ Record mit den Integer-Komponenten tag, $monat$ und $jahr$. Die Invariante würde in diesem Fall recht kompliziert:

     (A.jahr < 1995) ||
     (A.jahr = 1995 && A.monat < 3) ||
     (A.jahr = 1995 && A.monat = 3 && A.tag < 11)

   Die Beispiele zeigen, daß durch die Einführung des Typs Time die Invarianten einfacher und übersichtlicher werden.

2. Zeitliche Beschränkung der Gültigkeit von Invarianten

   Durch die Einführung des Typs Time ist es möglich, mit Hilfe einer knapp gehaltenen Syntax folgendes anzugeben:

   - Eine Invariante muß nur zu einem bestimmten Zeitpunkt gelten.
   - Eine Invariante muß bis zu einem bestimmten Zeitpunkt gelten.
   - Eine Invariante muß ab einem bestimmten Zeitpunkt gelten.

   Hierdurch kann der Zeitabschnitt, in dem eine Invariante gelten soll, von den ConTracts explizit angegeben werden. Dies ist auch Voraussetzung dafür, daß ConTracts über die Gültigkeitsdauer von sich widersprechenden Invarianten verhandeln können. Die genaue Syntax für die Angabe der zeitlichen Beschränkung von Invarianten folgt in Kapitel 4.

Der Aufbau einer Konstante des Typs Time gehorcht der folgenden Syntax:

Syntax:

time-constant	=	digit digit digit digit . digit digit .digit .digit : [digit digit . digit digit . digit digit] | [digit digit digit digit . digit digit .digit .digit] : digit digit . digit digit . digit digit
digit	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Dies läßt sich auch kurz zusammenfassen zu **YYYY.MM.DD:hh.mm.ss .** Hierbei haben die einzelnen Komponenten folgende Bedeutung:

YYYY	:	Jahr
MM	:	Monat
DD	:	Tag
hh	:	Stunden
mm	:	Minuten
ss	:	Sekunden

Die Syntax gibt zusätzlich die Möglichkeit, daß entweder die Angabe des Datums oder der Zeit fehlen kann. Dies macht Sinn bei der Verknüpfung von Zeitangaben. Näheres hierzu in Kapitel 4.

Beispiele:

Angabe von Datum und Zeit:	1994.12.11:13.10.59
Angabe des Datums:	1994.12.11:
Angabe der Uhrzeit:	:13.10.59
Angabe von bestimmten Zeiteinheiten, z.B. 1 Stunde	:01.00.00
2 Minuten	:00.02.00
1 Tag	0000.00.01:

3.2 Zusammengesetzte Typen

In den folgenden Abschnitten werden alle für Invarianten wichtigen zusammengesetzten Typen aufgeführt. Im Gegensatz zu den Basistypen weisen diese eine interne Struktur auf, d.h. es gibt Komponenten auf die separat zugegriffen werden kann.

3.2.1 String

Dieser Typ stellt Zeichenketten variabler Länge dar. In C gibt es keinen separaten Typ für Zeichenketten, weshalb diese dort als Zeiger auf char deklariert werden. Zeiger werden aus weiter unten aufgeführten Gründen für Invarianten nicht berücksichtigt. Statische Arrays stellen keine Alternative dar, da sich mit deren Hilfe nur Zeichenketten fester Länge abbilden lassen. So bleiben noch dynamische Arrays. Diese Möglichkeit wurde hier zugunsten eines eigenständigen Typs für Zeichenketten verworfen, da sich die zugelassenen Operatoren bei Arrays und Strings unterscheiden. Auf diese Weise können Ausnahmeregelungen bei den zugelassenen Operatoren verhindert werden.

Eine Konstante vom Typ String ist eine Folge von Einzelzeichen oder Escape-Zeichen, die in doppelten Anführungszeichen eingeschlossen sind.

Syntax:

string constant	=	″ { character | escape } ″
escape	=	\ [n | t | b | r | f | v | \ | ‘ | „ | a | ?] | \ octal | \ hex
octal	=	{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 }
hex	=	x { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F }

3.2.2 Array

Mit dem Typ Array ist es möglich, einen Vektor von Komponenten desselben Typs darzustellen. Als Komponententypen sind zunächst alle anderen für Invarianten ausgewählten Typen zulässig[16]. Dies bedeutet, daß auch mehrdimensionale Arrays möglich sind. Die Indizierung erfolgt mit Hilfe des einzigen hier betrachteten Ganzzahlentyps: Integer.

Konstanten sind für diesen Datentyp nicht vorgesehen.

3.2.3 Record

Objekte vom Typ Record enthalten mehrere Komponenten verschiedener Typen. Jeder Komponente ist ein Komponentenbezeichner zugeordnet. Als Datentypen für die Komponenten kommen zunächst sämtliche anderen für Invarianten ausgewählten Typen in Frage[17]. Bei Record handelt es sich also um einen Datentyp, der den Strukturtypen von C entspricht. Dagegen tauchen Records in einer SQL-Datenbank als Typ nicht explizit auf. Allerdings können sie sinnvoll eingesetzt werden, wenn eine Invariante Aussagen über eine ganze Relation der Datenbank machen soll. In der Invariante muß für diesen Fall die Möglichkeit bestehen, auf ein Objekt vom Typ Relation zuzugreifen. Der erste Schritt hierzu kann mit Hilfe von Records erfolgen. Ein Record hat dieselbe Struktur, wie ein Tupel einer Relation. Steht für jedes Attribut der Relation eine Record-Komponente zur Verfügung, so besteht hier die Möglichkeit einer direkten Abbildung. Der Schritt von einzelnen Tupeln kann dann mit dem nachfolgend aufgeführten Typ Set erfolgen.

Beispiel: Die Relation Mitarbeiter mit den Attributen Nummer, Name und Beruf kann auf einen Record mit den Komponenten Nummer, Name und Beruf abgebildet werden.

Wie für Arrays ist auch für den Typ Record keine Möglichkeit vorgesehen, eine Konstante anzugeben.

3.2.4 Set

Der Typ Set stellt wie Arrays eine Ansammlung von Komponenten desselben Typs dar. Der Unterschied zum Array besteht in der fehlenden Indizierung der Komponenten. Somit ist kein gezielter Zugriff auf ein bestimmtes Element der Menge möglich, und die Elemente eines Sets weisen keine Reihenfolge auf. Für die Komponenten selbst sind alle Typen mit Ausnahme der Menge selbst möglich.

Hier soll nun die Darstellung der Abbildung einer Relation auf die hier verwendeten Datentypen fortgesetzt werden. Der erste Schritt ist die Abbildung der Tupel auf einzelne Records. Wenn diese Records nun als Komponenten eines Sets eingesetzt werden, so ergibt sich die Struktur einer Relation. Hierbei bleiben wichtige Eigenschaften einer Relation erhalten:

> Die Zahl der Tupel ist nicht beschränkt. Dies ist sichergestellt, da die Anzahl der Komponenten eines Sets nach oben nicht beschränkt ist.
>
> Die Tupel weisen keine Reihenfolge auf. Dies ist ebenfalls durch die fehlende Reihenfolge bei Set-Komponenten abgebildet.

Auch für den Typ Set ist wie bei Arrays und Records keine Konstante vorgesehen.

---

[16] Vgl. Einschränkungen hierzu in Kapitel 5 bei der Abbildung von Prädikaten über zusammengesetzte Typen auf Prädikate über Basistypen und dort im Abschnitt über Arrays.

[17] Vgl. Einschränkungen hierzu in Kapitel 5 bei der Abbildung von Prädikaten über zusammengesetzte Typen auf Prädikate über Basistypen und dort im Abschnitt über Records.

3.3 Nicht berücksichtigte Typen

In diesem Abschnitt soll für verschiedene Datentypen begründet werden, warum sie nicht für Invarianten berücksichtigt werden. Natürlich können hier nicht alle Typen aufgezählt werden, die in irgend einem Typsystem vorgesehen sind. Das Hauptaugenmerk liegt hier auf Datentypen der Sprache C, die für die weiteren Betrachtungen bewußt beiseite gelassen werden.

3.3.1 Boolean

Ein Typ für Wahrheitswerte existiert in C nicht. Hier werden Wahrheitswerte einfach auf den Datentyp int abgebildet. Entsprechend kann für Invarianten eine Abbildung von Objekten, die Wahrheitswerte enthalten, auf ein Integer-Objekt einfach durchgeführt werden

3.3.2 Aufzählungstypen

Aufzählungstypen stellen in C lediglich eine Einschränkung des Wertebereichs für Integer-Typen dar. Darüber hinaus wird die Lesbarkeit des Programms durch die Einführung der Aufzählungskonstanten vereinfacht. Im Grunde genommen stecken hinter Aufzählungstypen also Objekte vom Typ Integer, so daß eine separate Betrachtung von Aufzählungstypen nicht notwendig ist. Darüber hinaus ist der Vorteil der besseren Lesbarkeit von Invarianten für diese Arbeit geringer zu bewerten als der Nachteil, der sich durch zusätzliche Komplexität der Invarianten-Syntax ergibt.

3.3.3 Zeigertypen

Zeiger finden ihre Anwendung in C einmal für die Darstellung von Zeichenketten als Zeiger auf char. Durch die Einführung des Typs String, wie weiter oben bereits beschrieben, entfällt dieser Grund für die Einführung von Zeigern.

Ein zweiter Grund für die Einführung von Zeigern wäre die Möglichkeit, komplexe sich verändernde Strukturen aufzubauen. Da auf die Betrachtung solcher komplexen Datentypen innerhalb dieser Arbeit aus den unter 3.3.4 beschriebenen Gründen verzichtet werden kann, ergeben sich auch hieraus keine zwingenden Gründe zur Einführung von Zeigern.

Ein Zeigerobjekt in einer Invariante würde auch ein viel komplexeres Objekt darstellen, als ein Zeiger in einer Sprache wie C. Hier handelt es sich lediglich um eine Speicheradresse. Ein Zeiger innerhalb einer Invariante müßte berücksichtigen, daß die referenzierten Objekte plattformübergreifend verteilt sind. Ein Zugriff auf sie kann nicht über eine einfache Speicheradresse erfolgen, sondern macht einen separaten Adressierungsmechanismus notwendig.

3.3.4 Komplexe Datentypen

Komplexe Datentypen werden in dieser Arbeit nicht weiter betrachtet, obwohl auch hier interessante Ergebnisse zu erwarten sind. Ihre Betrachtung ist aber zu umfangreich, als daß sie im Rahmen dieser Arbeit möglich wäre. Sie muß darum einer weiterführenden Bearbeitung überlassen werden.

4 Datentypen und zugelassene Operationen

Im vorangegangenen Kapitel wurden die Datentypen Integer, Floating, Character, Time, String, Array, Record und Set als für diese Arbeit relevant ausgewählt. Neben der Angabe des Wertebereichs gehört zu einem Datentyp auch eine Reihe von Operationen, die auf Objekte eines Typs angewandt werden können. Für jeden der oben genannten Datentypen existiert in C und anderen Programmiersprachen eine ganze Reihe von Operatoren. In diesem Kapitel soll nun dargestellt werden, welche dieser Operatoren in Invarianten sinnvoll eingesetzt werden können und welche entbehrlich sind. An einigen Stellen wird die Entscheidung mit Hilfe von Beispielen erläutert.

Wie bei der Wahl der Datentypen orientiert sich auch diese Auswahl zunächst an den Möglichkeiten, in C Ausdrücke zu bilden. Wesentliche Einschränkungen hierbei sind:

- In C kann jeder beliebige Ausdruck verschiedene Arten von Zuweisungen enthalten. Dies ist für Invarianten nicht erwünscht, da diese lediglich den Zustand von Objekten beschreiben sollen. Die Veränderung des Zustands erfolgt außerhalb der Invarianten. Bei Objekten, die in einer Datenbank abgelegt sind erfolgt die Änderung über Datenbankoperationen.

- In C besteht die Möglichkeit, sequentielle Ausdrücke zu bilden, wie z.B. $A < 4 , B > 3$. Das Ergebnis des Gesamtausdrucks ist das Ergebnis des am weitesten rechts stehenden Ausdrucks. Die anderen Teilausdrücke haben also nur dann eine Wirkung, wenn sie nicht nebenwirkungsfrei sind. Da in Invarianten keine Zuweisungen zugelassen werden, kann es aber keine solchen Ausdrücke mit Nebenwirkungen geben. Sequentielle Ausdrücke sind somit für Invarianten nicht sinnvoll.

- Bedingte Ausdrücke stellen eine weitere spezielle Form von C-Ausdrücken dar. Bei Invarianten handelt es sich um Prädikate, darum werden bedingte Ausdrücke hier nur für Prädikate betrachtet. Angenommen, A,b und c seien in C formulierte Prädikate, so ist ein möglicher bedingter Ausdruck: $A ? b : c$. Dies bedeutet, daß für den Fall $A == true$ b das Ergebnis des Gesamtausdrucks ist und c für den Fall $A == false$. Wie leicht zu sehen ist kann $A ? b : c$ auch als $(A \&\& b) || (!A \&\& c)$ dargestellt werden. Jeder bedingte Ausdruck über Prädikate kann also auch mit Hilfe logischer Operatoren dargestellt werden. Aus diesem Grund sind bedingte Ausdrücke für Invarianten nicht vorgesehen.

Nach diesen generellen Einschränkungen gegenüber der Möglichkeit in C Ausdrücke zu bilden, soll nun im Einzelnen betrachtet werden, welche Operatoren für welche Datentypen zugelassen werden. Das Kapitel ist nach Datentypen gegliedert.

4.1 Integer

unäres Plus oder Minus: + , -

Additionsoperatoren: + , -

Multiplikationsoperatoren: * , /

Diese Operatoren stellen Standardoperatoren für arithmetische Typen dar. Ihre Notwendigkeit auch für Invarianten muß also nicht weiter begründet werden.

Bei der Multiplikation und Division zweier Ganzzahlen wird als Ergebnis wieder eine Ganzzahl geliefert. Es ist dies eine der beiden Ganzzahlen, die am nächsten beim Quotienten liegen. Bei positiven Ganzzahlen ist es die kleinere der beiden. Bei negativen Ganzzahlen ist das Ergebnis machinenabhängig.

Ist einer der beiden Operanden vom Typ Floating, so wird der andere nach Floating umgewandelt und die entsprechende Fließkommaoperation ausgeführt.

Restbildung: %

Dieser Operator ist nur für zwei ganzzahlige Operanden zulässig. Seien diese a und b, so ist $a\%b$ äquivalent zu $a-(a/b)$. Der Operator zur Restbildung wurde hier von C übernommen, da sich mit ihm Ausdrücke übersichtlicher aufbauen lassen, als mit der entsprechenden ausführlichen Variante.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Diese Operatoren sind bei arithmetischen Datentypen Standard für die Bildung von Prädikaten. Das Ergebnis eines Vergleichs ist ein Wahrheitswert.

Bitorientierte Operatoren: ~ , << , >> , & , | , ^

$\sim a$ stellt die bitweise Negation des Operanden a dar. $a<<b$ bzw. $a>>b$ steht für eine bitweise Verschiebung der Operanden a um b Stellen nach links bzw. rechts. Bei $a\&b$ handelt es sich um eine bitweise Und-Verknüpfung der beiden Operanden. $a|b$ bzw. $a\hat{}b$ stellen entsprechend die Oder- bzw. Exklusivoderverknüpfung dar.

Diese bitorientierten Operatoren von C werden für Objekte vom Ganzzahltyp Integer auch in Invarianten zugelassen, da es hiermit möglich ist, Aussagen über einzelne Bit eines Integer-Objekts zu machen. Sinnvoll ist dies, wenn das Objekt keine Zahl repräsentiert, sondern als eine Folge von einzelnen Bit gesehen werden soll. In diesem Fall kann dem Zustand jedes einzelnen Bit eine bestimmte Information zugeordnet werden. Mit Hilfe der hier vorgestellten Operatoren ist auch in Invarianten eine Aussage über einzelne Bit möglich.

4.2 Floating

unäres Plus oder Minus: + , -

Additionsoperatoren: + , -

Multiplikationsoperatoren: * , /

Vergleichsoperatoren: < , <= , > , >= , == , !=

Wie bei Integer-Objekten bedürfen diese arithmetischen Standardoperatoren keines weiteren Kommentars. Es gilt das im entsprechenden Abschnitt für die Anwendung dieser Operatoren auf Integer-Objekte Gesagte.

4.3 Character

In C sind an allen Stellen eines Ausdrucks, an denen ein Ganzzahlwert erscheinen kann auch Einzelzeichentypen zugelassen. Dies bedeutet unter anderem, daß auch die Addition von Zeichen möglich ist. Auf diese Vielzahl von Operatoren soll hier für den Datentyp Character verzichtet werden. Hier werden nur solche Operatoren eingeführt, deren Semantik für Zeichen offensichtlich ist. Dies ist in erster Linie bei Vergleichen der Fall. Ausdrücke wie *„ a “ + „ b “* haben hingegen eine unklare Bedeutung. Entweder man interpretiert diesen Ausdruck als die Summe der Zeichencodes für die beiden Zeichen *a* und *b*. Oder man wählt als Interpretation eine Verkettung der beiden Zeichen, so daß das Ergebnis des Ausdrucks die Zeichenkette *„ a b “* ist. Diese zweite Möglichkeit ist für Invarianten mit der Konkatenation von Strings abgedeckt. Die Möglichkeit *„ a “ + “ b “* als Summe zu sehen wird hier nicht weiter betrachtet, da es sich nicht um eine Operation speziell für den Typ Character handelt, sondern um eine Addition von Integer-Werten.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Für den Typ Character sind alle paarweisen Vergleiche von Einzelzeichen möglich. Der Vergleich erfolgt anhand des zugrunde liegenden Zeichencodes, z.B. des ASCII-Codes. Das Ergebnis des Vergleichs ist ein Wahrheitswert.

4.4 Time

Die Operatoren, die für den Typ Time vorgesehen sind, gliedern sich in zwei Gruppen. Die erste Gruppe enthält arithmetische Operationen und Vergleiche für Zeitwerte. In die zweiten Gruppen fallen Operatoren, mit deren Hilfe die Gültigkeitsdauer von Invarianten eingeschränkt werden kann. Andere für diesen Datentyp denkbare Operatoren werden hier nicht berücksichtigt.

Additionsoperatoren: + , -

Zeitangaben können addiert und subtrahiert werden. In diesem Zusammenhang wird deutlich, warum Objekte vom Typ Time neben der kompletten Datums- und Zeitangabe auch nur zur Angabe von

Jahre.Monate.Tage oder Stunden.Minuten.Sekunden genutzt werden können. Hiermit wird ein Objekt wie z.B. *Flugdauer* ermöglicht, das den Wert :01.10.30 enthalten kann. Die Bedeutung ist klar: Die im Objekt *Flugdauer* gespeicherte Flugdauer beträgt 1 Stunde 10 Minuten und 30 Sekunden. Gibt es daneben ein Objekt *Abflugzeitpunkt* mit dem Inhalt 1995.01.01:13.49.30, so ist *Abflugzeitpunkt + Flugdauer* ein sinnvoller Ausdruck, der als Ergebnis den Ankunftszeitpunkt angibt.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Objekte vom Typ Time können mit Hilfe aller Vergleichsoperatoren verglichen werden. Der Vergleich erfolgt in der für Zeitangaben naheliegenden Art und Weise, so daß z.B. *1994.12.01:10.10.10 < 1994.12.02:10.10.10* eine wahre Aussage darstellt. Erfolgt ein Vergleich mit Time-Objekten, bei denen entweder die Datumsangabe oder die Uhrzeitangabe fehlt, so werden die fehlenden Positionen als mit 0 belegt betrachtet. Das Ergebnis eines Vergleichs ist in jedem Fall ein Wahrheitswert.

Komponentenzugriff: year , day , month , hour , minute , second

Diese Operatoren sind denkbar, um einen Teil der Zeitangabe zu separieren. So könnte das Ergebnis von day(1994.31.03:14.10.27) der Wert 31 sein. Werden die Funktionen auf diese Weise betrachtet, so stellen sie lediglich eine Abbildung vom Typ Time auf den Typ Integer dar. Da es sich nicht um elementar wichtige Operatoren handelt, die zudem für die weiteren Teile dieser Arbeit keine zusätzlichen Aspekte beisteuern, werden diese Funktionen nicht weiter berücksichtigt.

Zeitliche Beschränkung von Invarianten: at , before , after

Diese Operatoren ermöglichen es, innerhalb eines ConTracts zeitlich beschränkte Invarianten anzugeben. Die drei Operatoren haben jeweils als linken Operanden eine Invariante und als rechten Operanden einen Ausdruck vom Typ Time. Mit dem Operator *at* wird angegeben, daß die Invariante lediglich zu einem bestimmten Zeitpunkt gelten muß. Die beiden anderen Operatoren *before* und *after* beschränken die Gültigkeit dann jeweils auf die Zeit vor oder nach dem angegebenen Zeitpunkt.

Eine Einsatzmöglichkeit für diese Operatoren besteht darin, daß ConTracts über die Gültigkeitsdauer ihrer Invarianten verhandeln können. Dies ist vor allem dann sinnvoll, wenn zwei ConTracts sich widersprechende Invarianten dem Prädikatverwalter übergeben wollen. Folgende Verhandlungsschritte sind in diesem Fall denkbar:

1. ConTract 1 legt als eine Invariante *A >= 10* fest.
2. ConTract 2 möchte als eine Invariante *A < 10* einbringen, was aufgrund des Widerspruchs zur bestehenden Invariante von ConTract 1 nicht möglich ist.
3. ConTract 1 und ConTract 2 „verhandeln“. ConTract 2 teilt ConTract 1 mit, daß für ihn die Invariante nur vor Zeitpunkt1 wichtig ist.
4. ConTract 1 verändert die Invariante in *A >= 10 AFTER Zeitpunkt1*.
5. ConTract 2 kann nun die Invariante *A < 10 BEFORE Zeitpunkt1* festlegen.

4.5 String

Komponentenzugriff: [ ]

Bei Strings ist es möglich, das Zeichen an einer bestimmten Position im String zu separieren. Der Index muß hierbei vom Typ Integer sein. Index der ersten Komponente ist immer 0. Das Ergebnis dieses

Operators ist wieder ein String und zwar der Länge 1. Beim Zugriff auf eine Position nach dem Ende des Strings wird als Ergebnis ein leerer String geliefert. Der Komponentenzugriff für Strings entspricht dem Zugriff auf einzelne Komponenten bei Arrays. Sinnvoll kann dieser Operator z.B. eingesetzt werden, wenn eine Invariante festlegen soll, daß das Objekt *Nachname* mit dem Buchstaben A beginnen muß. Eine mögliche Invariante lautet: *Nachname[0] == „A“*.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Sämtliche möglichen Vergleiche sollen hier für Strings zugelassen werden. Die Semantik der einzelnen Vergleichsoperatoren wurde so gewählt, daß sie möglichst weitgehend mit der üblichen Sortierung von Zeichenketten in Lexika u.ä. übereinstimmt. Die Bedeutung der Vergleichsoperatoren im einzelnen:

==

Gleichheit besteht zwischen zwei Strings, wenn beide dieselbe Länge besitzen und Gleichheit zwischen allen Zeichenpaaren an derselben Position besteht.

!=

Ungleichheit besteht zwischen zwei Strings, wenn sie unterschiedlich lang sind oder wenn sie sich bei gleicher Länge an mindestens einer Position unterscheiden.

<

Die Kleinerbeziehung gilt zwischen zwei Strings, wenn diese auch an der Position gilt, an der die Zeichenpaare erstmals nicht mehr gleich sind. Die Vergleiche werden hierbei mit der Position 0 begonnen. Gilt bis zum Ende eines der beiden Strings die Gleichheit für alle Zeichenpaare, so ist der kürzere String kleiner als der längere.

>

Hier gilt die zur Kleinerbeziehung symmetrische Aussage.

<=

Diese Beziehung gilt, wenn entweder die Gleichheit oder die Kleinerbeziehung gilt.

>=

Diese Beziehung gilt, wenn entweder die Gleichheit oder die Größerbeziehung gilt.

Beispiele: „Tisch“==„Tisch“

„Stuhl“!=„Tisch“

„Stuhl“<„Tisch“

„Tisch“<„Tischbein“

„Tischdecke“>„Tischbein“

„Stuhl“>=„Stuhl“

Konkatenation: +

Mit diesem Operator können einzelne Zeichen oder ganze Strings an andere Strings angehängt werden. Das Ergebnis ist wieder ein String. Damit ist es möglich, mehrere String-Objekte zu verketten und mit anderen String-Objekten zu vergleichen.

Länge: length

Dieser Operator liefert ein Ergebnis vom Typ Integer. Dieses gibt die Zahl der Einzelzeichen im als Operand angegebenen String an. Hiermit können in Invarianten Aussagen über die Länge von String-Objekten gemacht werden.

4.6 Array

Komponentenzugriff: [ ]

Diese Operation erlaubt den Zugriff auf einzelne Komponenten eines Arrays. Der Index muß ein Ausdruck vom Typ Integer sein. Index der ersten Komponente ist immer 0. Das Ergebnis besitzt den Komponententyp des Arrays. Es wird hier davon ausgegangen, daß die Indexgrenzen für Objekte vom Typ Array auf Stepebene bekannt sind. Somit können Zugriffe auf nicht vorhandene Komponenten vernachlässigt werden.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Für den Vergleich von Arrays muß zunächst die Semantik eines Ausdrucks, wie $A == B$ geklärt werden. Folgende Interpretationen sind möglich:

- Verglichen wird die Anzahl der Komponenten der beiden Arrays. Die Aussage des obigen Ausdrucks ist in diesem Fall: „A besitzt dieselbe Anzahl an Komponenten wie B.“
- Verglichen werden die jeweiligen Komponenten der Arrays. Hierbei ist es naheliegend, daß der Vergleich kompletter Arrays auf Vergleiche von Komponentenpaaren mit demselben Index abgebildet wird. Der Beispielausdruck bedeutet dann: „A und B sind in allen ihren Komponenten gleich.“

Für die weitere Arbeit wird von der zweiten Interpretation ausgegangen. Invarianten im Sinne der ersten lassen sich dann formulieren, wenn Operatoren für Arrays zur Verfügung stehen, die die Indexgrenzen als Ergebnis liefern. Auf die Betrachtung solcher Operatoren soll hier verzichtet werden.

Voraussetzungen dafür, daß ein Vergleichsausdruck im Sinne der zweiten Interpretation ausgewertet werden kann, sind:

- Der Komponententyp der beiden Arrays ist gleich.
- Für den Komponententyp der beiden Arrays existiert der gewählte Vergleichsoperator.
- Die Indexgrenzen der beiden Arrays stimmen überein. Da bei jedem Array 0 als die Untergrenze für den Index angenommen wird, ist die Forderung hier in jedem Fall erfüllt. Die Gleichheit der Obergrenzen ist keine zwingende Voraussetzung für die Durchführbarkeit eines Vergleichs, sie macht die Semantik der Vergleichsoperatoren für Arrays aber klarer.

Sind diese beiden Voraussetzungen gegeben, so kann die Semantik für die jeweiligen Vergleichsoperatoren wie folgt angegeben werden:

==

Gleichheit ist gegeben, wenn die Gleichheit für alle Komponentenpaare mit gleichem Index gilt.

!=

Ungleichheit ist gegeben, wenn die Gleichheit für mindestens ein Komponentenpaar mit gleichem Index nicht gilt.

<, >

Die Kleinerbeziehung bzw. die Größerbeziehung gilt, wenn sie für alle Komponentenpaare mit demselben Index erfüllt ist.

<=, =>

Diese Beziehungen lassen sich wie bei Strings auf die einfachen Beziehungen ==, < und > zurückführen.

Diese Definition der Vergleichsoperatoren für Arrays bringt es mit sich, daß Paare von Arrays existieren, zwischen denen keine der Relationen ==, < oder > gilt. Ein Beispiel sind zwei Arrays mit Integer-Komponenten: Zwischen *(1, 2, 3)* und *(2, 1, 4)* gilt lediglich die Ungleichheit. Dies stellt einen wichtigen Unterschied zu den Vergleichen bei Integerwerten oder auch bei Strings dar. Die Abbildung des Array-Vergleichs auf die Komponentenvergleiche wird in Kapitel 5 näher betrachtet.

4.7 Record

Komponentenzugriff: [ ]

Mit diesem Operator wird auf eine einzelne Komponente eines Records zugegriffen. Die gewünschte Komponente wird mit Hilfe ihres Namens ausgewählt. Ergebnis der Operation ist ein Objekt vom Typ der referenzierten Komponente.

Vergleichsoperatoren: < , <= , > , >= , == , !=

Wie bei den Vergleichen für Arrays muß auch hier zuerst die Semantik für die Vergleichsoperatoren festgelegt werden. Es bieten sich verschiedene Möglichkeiten an:

- Der Vergleich kann sich auf die Struktur der Records beziehen, d.h. in welcher Relation stehen Anzahl und Typen der Komponenten zueinander.
- Vergleiche beziehen sich auf die Komponenten selbst.

Auch hier sollen die Vergleichsoperatoren mit der zweiten Semantik belegt sein. Das Vergleichsergebnis setzt sich also aus den Ergebnissen von Einzelvergleichen der Komponenten zusammen. Voraussetzungen dafür, daß ein Vergleich überhaupt durchführbar ist, sind:

- Die Struktur der zu vergleichenden Records ist identisch, d.h. sie weisen sowohl dem Namen als auch dem Typ nach dieselben Komponenten auf.
- Der gewählte Vergleichsoperator exsistiert für jeden Komponententyp.

Sind diese Voraussetzungen erfüllt, so kann die Semantik für sämtliche Vergleiche wie folgt angegeben werden:

==

Gleichheit ist gegeben, wenn sie für sämtliche Komponentenpaare gilt.

!=

Ungleichheit ist gegeben, wenn sie für mindestens ein Komponentenpaar gilt.

<, >

Die Kleinerbeziehung bzw. die Größerbeziehung gilt, wenn sie für alle Komponentenpaare gilt.

<=, =>

Diese Beziehungen lassen sich wie bei Strings auf die einfachen Beziehungen ==, < und > zurückführen.

Wie bei den Vergleichsoperatoren für Arrays ist auch hier zu beachten, daß es Paare von Records gibt, für die weder die Gleichheit noch eine Größer- oder Kleinerbeziehung gilt. Für alle diese Paare gilt aber in jedem Fall die Ungleichheit. Somit ist sichergestellt, daß zwischen jedem Record-Paar mindestens einer der Vergleichsoperatoren zu einer wahren Aussage führt. Die Abbildung des Record-Vergleichs auf Vergleiche der einzelnen Komponenten wird in Kapitel 5 näher betrachtet.

4.8 Set

Komponentenzugriff

Die Komponenten eines Sets weisen keine Ordnung auf. Sie sind auch nicht über Namen zu unterscheiden. Dies macht einen gezielten Zugriff auf einzelne Komponenten unmöglich. Ein Zugriff auf Komponenten wäre in der Form einer mengenorientierten Anfrageschnittstelle - ähnlich wie SQL-Anfragen - denkbar. Diese Ausdrucksmächtigkeit ist für Invarianten nicht vorgesehen. Vielmehr wird davon ausgegangen, daß eine Abbildung z.B. auf SQL-Anfragen in tieferen Schichten des Systems realisiert wird. Ein Objekt vom Typ Record, an das eine Invariante bestimmte Bedingungen stellt, kann vom Resource-Manager, der das Objekt verwaltet, in eine entsprechende SQL-Anfrage übersetzt werden, falls dies erforderlich ist.

Enthaltenseinsoperator: in

Dieser Operator hat als Operanden einmal ein Set-Objekt und einmal ein Objekt vom Komponententyp des Sets. Der Ausdruck stellt genau dann eine wahre Aussage dar, wenn das Objekt als Komponente in der Menge enthalten ist.

Anzahloperator: count

Dieser Operator hat nur einen Operanden, nämlich ein Set-Objekt. Das Ergebnis ist vom Typ Integer und gibt die Anzahl der Komponenten im Set-Objekt an.

Aggregatsfunktionen: max, min, avg, sum

Diese Operatoren sind sinnvoll, wenn über alle Komponenten einer Menge hinweg aggregierte Werte bestimmt werden sollen. Diese Aggregatsfunktionen führen beim hier betrachteten Typ Set zu Problemen bei der Festlegung der Syntax. Die Operatoren lehnen sich zwar an entsprechende Aggregate von SQL an, die Syntax von SQL läßt sich aber nicht direkt übernehmen. Das Problem soll kurz skizziert werden:

Bei Anfragen in SQL ist klar, daß die Aggregate über Attribute der der Abfrage zugrunde liegenden Relationen berechnet werden. Durch die verschiedenen Möglichkeiten bei der Wahl des Komponententyps für Set-Objekte ist diese Zuordnung hier nicht mehr so eindeutig. Verschiedene Beispiele sollen die daraus resultierende Komplexität verdeutlichen:

- Das Set-Objekt enthält Komponenten eines Basistyps, z.B. Integer. In diesen Fällen gilt das Aggregat direkt für die Komponenten der Menge.
- Das Set-Objekt enthält Komponenten eines zusammengesetzten Typs, z.B. Array. In diesem Fall kann sich das Aggregat auf verschiedene Ebenen beziehen. Die erste Ebene stellen die Komponenten selbst dar. Auf der zweiten Ebene liegen die Komponenten der Komponenten. Die Zahl der hier möglichen Ebenen ist wie die Tiefe der Schachtelung von zusammengesetzten Datentypen prinzipiell nicht beschränkt.
- Als Ergebnis eines Aggregats kann einmal der ermittelte Wert selbst erwartet werden. Darüber hinaus kann bei der Maximumbildung und der Minimumbildung auch von Interesse sein, welche Komponente des Set-Objekts diesen aggregierten Wert aufweist.

Die Berücksichtigung der unterschiedlichen Möglichkeiten Aggregate zu bilden würde die Grammatik für Invarianten wesentlich komplizierter gestalten. Darüber hinaus haben die Aggregatsfunktionen für die weitere Arbeit keine entscheidende Bedeutung, so daß auf ihre Einführung hier verzichtet wird.

Vergleichsoperatoren: ==, !=, < , <=, >, >=

Bei Vergleichen von Mengen ist die Semantik nicht von vornherein klar. Die Betrachtung soll hier in zwei Schritten erfolgen.

==, !=

Für die Gleichheit kommen zwei mögliche Interpretationen in Frage:

- Gleichheit bezieht sich auf die Zahl der Elemente in der Menge. Da dies auch mit Hilfe des count-Operators dargestellt werden kann, ist die zweite Interpretation an dieser Stelle sinnvoller.
- Gleichheit zweier Set-Objekte bedeutet, daß beide exakt dieselben Komponenten aufweisen. Diese Gleichheit kann festgestellt werden, da für alle anderen Typen die Gleichheit ebenfalls definiert ist.

<, >

Für die Größer- und Kleinerrelationen sind ebenfalls mehrere Möglichkeiten vorhanden, den Vergleich zu interpretieren:

- Der Vergleich bezieht sich auf die Zahl der Elemente in einer Menge.
- Der Vergleich besagt, daß die Relation für jedes beliebige Paar von Komponenten der beiden Set-Objekte gelten muß.
- Der Vergleich wird durchgeführt, indem die Komponenten der beiden Mengen sortiert werden. Es findet quasi eine Abbildung auf ein Array statt. In diesem Array kann dann paarweise der Vergleich für die Komponenten mit demselben Index durchgeführt werden. Es wird also der weiter oben beschriebene Array-Vergleich durchgeführt.

Am ehesten stellt von diesen drei Möglichkeiten wohl der Größenvergleich über die Zahl der Komponenten eine „natürliche“ Lösung dar. Da sich dieser Vergleich aber auch mit Hilfe des count-Operators darstellen läßt wird auf eine weitere Betrachtung der Relationen <, <=, >, >= für Set-Objekte verzichtet.

4.9 Typunabhängige Operatoren

exist

Dieses Prädikat ist für Objekte sämtlicher Typen möglich. Es kann überall dort eingesetzt werden, wo nicht ein bestimmter Wert eines Objekts sondern lediglich die Existenz des Objekts selbst bis zum Ende eines ConTracts gesichert werden muß.

Für jeden Step eines ConTracts gibt es einen zugeordneten Compensation-Step[18]. Dieser macht die Aktionen des Steps rückgängig. Hierzu werden verschiedene Objekte so verändert, daß sich ein logisches Undo des Steps ergibt. Die Kompensation kann nur dann korrekt ablaufen, wenn alle diese Objekte auch tatsächlich vorhanden sind. Es müssen also Sperren für Objekte vorhanden sein, die lediglich deren Existenz sichern. Bei Objekten, für die in einer Invariante Eigenschaften festgelegt werden, muß eine solche Existenzsperre automatisch gesetzt werden. Für andere Objekte, bei denen lediglich die Existenz sichergestellt werden soll, ist dieser Operator *exist* gedacht.

!, &&, ||

Diese drei logischen Operatoren sind ebenfalls unabhängig von den Datentypen für Objekte. Ihre Aufgabe ist es, einzelne Teilprädikate zu verknüpfen. Als Operanden der drei Operatoren sind sämtliche, korrekt gebildete Prädikate zugelassen.

[18] Vgl. [WäRe91], S.25-29

4.10 Operatoren

In der folgenden Tabelle 1 werden sämtliche Operatoren und die jeweils zugelassenen Operanden in der Übersicht dargestellt. In der weiteren Arbeit werden nur die hier dargestellten Möglichkeiten berücksichtigt. Einige Erläuterungen zu dieser Tabelle:

- Zu jedem Operator sind in separaten Zeilen die zugelassenen Kombinationen der Operanden aufgelistet. Über die aufgeführten hinaus sind keine Operanden möglich. Bei einigen Operatoren ist als ein Operand *predicate* eingetragen. Dies bedeutet, daß an der Stelle der Operanden beliebige korrekt gebildete Prädikate stehen können.
- Bei jeder Kombination von Operanden ist der Typ des Ergebnisses angegeben. Ist als Ergebnistyp *predicate* eingetragen, so soll hiermit ausgesagt werden, daß durch die Anwendung der Operation ein Prädikat entsteht. Dem sich ergebenden Ausdruck kann also ein Wahrheitswert zugeordnet werden.
- Zusätzlich ist die Wertigkeit und die Assoziativität der Operatoren angegeben. Steht ein Operand zwischen zwei Operatoren, so wird er zuerst dem mit der höheren Wertigkeit zugeordnet. Besitzen beide Operatoren dieselbe Wertigkeit, so entscheidet die Assoziativität, zu welchem Operator der Operand gehört.

Operator	**1. Operand**	**2. Operand**	**Ergebnis**	**Semantik**	**Wertigkeit, Assoziativität**
+ , - unär	Integer oder Floating	-	wie 1. Operand	liefert den Wert des 1. Operanden bzw. dessen Negation	13, rechts
+ binär	Integer oder Floating	Integer oder Floating	Floating, falls ein Operand vom Typ Floating ist Integer, sonst	gewöhnliche Addition	10, links
	String	String oder Character	String	Konkatenation	10, links
	Time	Time	Time	Addition von Zeitpunkten	...
- binär	Integer oder Floating	Integer oder Floating	Floating, falls ein Operand vom Typ Floating ist Integer, sonst	gewöhnliche Subtraktion	10, links
	Time	Time	Time	Subtraktion von Zeitpunkten	...
*	Integer oder Floating	Integer oder Floating	Floating, falls ein Operand vom Typ Floating ist Integer, sonst	Multiplikation	11, links
/	Integer oder Floating	Integer oder Floating	Floating, falls ein Operand vom Typ Floating ist Integer, sonst	Division	11, links
%	Integer	Integer	Integer	Restbildung	11, links
< , <= , > , >=	Integer oder Floating	Integer oder Floating	predicate	kleiner, kleinergleich, größer, größergleich	8, links
	Character	Character	predicate	Vergleich entsprechend des Zeichencodes	...

Operator	1. Operand	2. Operand	Ergebnis	Semantik	Wertigkeit, Assoziativität
	String	String	predicate	Semantik des Vergleichs siehe Abschnitt 4.5	...
	Array	Array	predicate	Semantik des Vergleichs siehe Abschnitt 4.6	...
	Record	Record	predicate	Semantik des Vergleichs siehe Abschnitt 4.7	
==, !=	Integer oder Floating	Integer oder Floating	predicate	gleich, ungleich	7, links
	Character	Character	predicate	Vergleich entsprechend des Zeichencodes	...
	String	String	predicate	Semantik des Vergleichs siehe Abschnitt 4.5	...
	Array	Array	predicate	Semantik des Vergleichs siehe Abschnitt 4.6	...
	Record	Record	predicate	Semantik des Vergleichs siehe Abschnitt 4.7	...
~	Integer	Integer	Integer	bitorientierte Negation	15, rechts
&	Integer	Integer	Integer	bitorientiertes AND	6, links
|	Integer	Integer	Integer	bitorientiertes OR	4, links
^	Integer	Integer	Integer	bitorientiertes XOR	5, links
<<	Integer	Integer	Integer	a << b: Operand a um b Bit nach links schieben	9, links
>>	Integer	Integer	Integer	a >> b: Operand a um b Bit nach rechts schieben	9, links
!	predicate	-	predicate	logisches NOT	14, rechts
&&	predicate	predicate	predicate	logisches AND	2, links
||	predicate	predicate	predicate	logisches OR	1, links
AT BEFORE AFTER	predicate	Time	predicate	Gibt die Zeitpunkte an, zu denen Prädikate gelten sollen.	3, links
IN	beliebig	Set	predicate	Gibt an, ob ein Objekt Element der Menge ist.	12,
COUNT	Set	-	Integer	Liefert die Anzahl der Elemente einer Menge	-
EXIST	beliebig	-	predicate	Stellt sicher, daß ein Objekt weiterhin existiert.	-
LENGTH	String	-	Integer	Liefert die Länge eines Strings	-

Tabelle 1: Operatoren und zugelassene Operanden

4.11 Grammatik für Invarianten

Die folgende Grammatik in Abbildung 4 stellt als Ergebnis dieses und des vorangegangenen Kapitels dar, welche Ausdrücke als Invarianten zugelassen werden sollen. Die Ausdrucksmächtigkeit der Grammatik wird durch die für einzelne Datentypen zugelassenen Operationen gemäß Tabelle 1 weiter eingeschränkt. Zunächst einige Anmerkungen zur Darstellung der Grammatik:

- Terminalsymbole sind fett gedruckt, während Nichtterminale normal ausgeführt sind.
- Alternative Produktionen sind mit | voneinander getrennt.
- Teile die optional stehen können sind in eckigen Klammern [ ] eingefaßt.
- Teile die beliebig oft stehen können sind in geschweiften Klammern { } zusammengefaßt.

predicate	=	expression relop expression | predicate timeop time-expression | **!** predicate | predicate **&&** predicate | predicate **||** predicate | primary **in** name **(** predicate **)** | **exist(** name **)** | **length(** name **)**
expression	=	primary | **-** primary | **+** primary | ~ primary | time-expression | primary binop expression
primary	=	name | constant | primary . name | primary [expression] | **coun**t(name **)** | **(** expression **)**
time-expression	=	time-primary | time-primary time-add time-expression
time-primary	=	name | time-constant
relop	=	**==** | **!=** | **<** | **<=** | **>** | **>=**
timeop	=	**at** | **before**| **after**
time-add	=	**+** | **-**
binop	=	**+** | **-** | ***** | **/** | **%** | **<<** | **>>** | **&** | **|** | **^**

name	=	Bezeichner für Objekte und Kontextvariablen
constant	=	integer constant | floating constant | character constant | string constant
integer constant	=	[sign] (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9) { digit }
floating constant	=	[sign] digit { digit } [.{ digit }] [E [sign] digit { digit }] | [sign] { digit } . digit { digit } [E [sign] digit { digit }]
digit	=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
sign	=	+ | -
character constant	=	**‘** character **‘** | **‘** escape **‘**
string constant	=	**″** { character | escape } **″**
escape	=	\ [n | t | b | r | f | v | \ | ‘ | „ | a | ?] | \ octal | \ hex
octal	=	{ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 }
hex	=	**x** { 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F }
character	=	beliebiges Zeichen

Abbildung 4: Grammatik für Invarianten

5 Abbildung auf Basistypen

Nachdem in den vorangegangenen Kapiteln Datentypen für Invarianten ausgewählt, und die Reihe auf Objekte solcher Datentypen zugelassenen Operationen bestimmt wurde, soll jetzt die Frage bearbeitet werden, wie Prädikate über zusammengesetzte Typen auf Prädikate über Basistypen abgebildet werden können. Gelingt es, für möglichst viele Invarianten über zusammengesetzte Datentypen solche Abbildungen zu finden, so kann daraus ein Vorteil für die Verwaltungskomponente der Invarianten resultieren. Hier müßten dann nur noch Invarianten über Basistypen berücksichtigt werden.

Dieses Kapitel gliedert sich in 4 Abschnitte für die vier zu betrachtenden zusammengesetzten Typen String, Array, Record und Set.

5.1 String

Ausgegangen wird hier von den beiden Strings A und B. Deren Längen lassen sich in einer Invarianten mit $length(A)$ und $length(B)$ bestimmen. Entsprechend werden hier $L_A := length(A)-1$ und $L_B := length(B)-1$ angenommen. L_A und L_B geben den Index des letzten Zeichens eines Strings an. Werden die von einer Invariante betroffenen Strings für die Zeit, in der die Invariante gilt, als in ihrer Länge fixiert angenommen, so kann bei den hier dargestellten Abbildungen jeweils L_A und L_B durch den aktuellen konstanten Wert ersetzt werden. Die Komponentenvergleiche können jeweils nur bis zu der Position durchgeführt werden, an der noch beide Strings Komponenten aufweisen. Diese Position ist $L:=MIN(L_A, L_B)$. Die Abbildung von Prädikaten über Strings auf Prädikate über einzelne Zeichen erfolgt gemäß der in Abschnitt 4.5 festgelegten Semantik für Stringvergleiche.

A==B

Die Gleichheit von Strings wird abgebildet auf die Gleichheit ihrer Länge und die Gleichheit aller Zeichenpaare mit demselben Index. Das Prädikat A==B läßt sich auch ausdrücken als:

L_A==L_B && (Ausdruck1)
A[0]==B[0] &&
A[1]==B[1] &&
...
A[L]==B[L]

A!=B

Da $(a==b) \Leftrightarrow !(a!=b)$ gilt kann $A!=B$ abgebildet werden auf $!(Ausdruck1)$. Dies wiederum kann geschrieben werden als:

L_A!=L_B ||
A[0]!=B[0] ||
A[1]!=B[1] ||
...
A[L]!=B[L]

A<B , A>B

Laut der Semantik für den Stringvergleich hängt auch diese Relation zwischen zwei Strings einerseits von der Länge der Strings und andererseits von dem Vergleich der Einzelzeichen ab. Eine Abbildung auf Prädikate über Basistypen wird gegeben durch:

```
A[0]<B[0] ||
(A[0]==B[0] && A[1]<B[1]) ||
(A[0]==B[0] && A[1]==B[1] && A[2]<B[2]) ||
...
(A[0]==B[0] && A[1]==B[1] && ... && A[L-1]==B[L-1] && A[L]<B[L] ) ||
(A[0]==B[0] && A[1]==B[1] && ... && A[L]==B[L] && L_A<L_B )... )
```

Durch Ausklammern von Teilausdrücken kann dieser Ausdruck so vereinfacht werden, daß keine doppelten Teilausdrücke vorkommen.

```
A[0]<B[0] || (                                                    (Ausdruck2)
A[0]==B[0] && ( A[1]<B[1] || (
A[1]==B[1] && ( A[2]<B[2] || (
...
A[L-1]==B[L-1] && ( A[L]<B[L] ) || (
A[L]==B[L] && L_A<L_B ) ... )
```

Die symmetrische Operation $A > B$ wird hier nicht gesondert betrachtet, da sie durch $B < A$ ersetzt werden kann.

A<=B , A>=B

Für diese zusammengesetzte Operatoren kann die Abbildung aus den entsprechenden Abbildungen für die einfachen Operatoren zusammengesetzt werden. So ist $A \leq B$ gleichbedeutend mit $Ausdruck1$ || $Ausdruck2$. Werden hier die beiden Ausdrücke von oben eingesetzt, so ergibt sich ein Ausdruck äquivalent zu:

```
A[0]<B[0] || (
A[0]==B[0] && ( A[1]<B[1] || (
A[1]==B[1] && ( A[2]<B[2] || (
...
A[L-1]==B[L-1] && ( A[L]<B[L] || (
A[L]==B[L] && ( L_A==L_B || L_A<L_B ) ... )
```

Die symmetrische Operation A>=B kann als B<=A geschrieben werden und ist mit dieser Abbildung damit ebenfalls abgedeckt.

5.2 Array

Bei Arrays sind ähnliche Abbildungen wie bei Strings möglich. Durch sie können Invarianten über Arrays auf Invarianten über Arraykomponenten abgebildet werden. Gegenüber der Betrachtung bei Strings gibt es einige wichtige Unterschiede. Auf diese soll aber erst im Anschluß an die Darstellung möglicher Abbildungen eingegangen werden.

Für das Folgende wird angenommen, daß es sich bei A und B um zwei Arrays handelt. Die Anzahl ihrer Komponenten sei mit L_A bzw. L_B gegeben. Da als Voraussetzung für die Durchführbarkeit der Vergleiche eine identische Komponentenanzahl bei beiden Arrays vorliegen muß kann $L := L_A \ (= L_B)$ gesetzt werden. Die Komponenten der beiden Arrays haben denselben Datentyp. Die Abbildungen von Prädikaten über Arrays auf Prädikate über Arraykomponenten gemäß der in Abschnitt 4.6 festgelegten Semantik für Vergleichsoperatoren sehen dann folgendermaßen aus:

A==B , A!=B

Die Gleichheit läßt sich abbilden auf den Ausdruck:

```
A[0]==B[0] &&                                    (Ausdruck3)
A[1]==B[1] &&
...
A[L]==B[L]
```

Für die Ungleichheit kann der Ausdruck !(Ausdruck3) angegeben werden. Eine äquivalente Form hierfür ist:

```
A[0]!=B[0] ||
A[1]!=B[1] ||
...
A[L]!=B[L]
```

A<B , A>B

Hier genügt es die Kleinerrelation zu betrachten, da $A > B$ äquivalent ist zu $B < A$. Die Gültigkeit von $A < B$ kann auch mit Hilfe des folgenden Ausdrucks überprüft werden:

```
A[0]<B[0] &&                                     (Ausdruck4)
A[1]<B[1] &&
...
A[L]<B[L]
```

A<=B , A>=B

Diese beiden Relationen können einfach auf die Kleinerrelation und die Gleichheit zurückgeführt werden. Aus $A <= B$ wird dann: $(A < B) \;||\; (A == B)$ und aus $A >= B$ wird $(B < A) \;||\; (A == B)$. Für die jeweiligen Teilprädikate können die entsprechenden Abbildungen von oben eingesetzt werden. Eine Vereinfachung der so entstandenen Ausdrücke ist allerdings nicht möglich.

Unterschiede zur Betrachtung bei Strings

Abgesehen davon, daß $A < B$ eine unterschiedliche Semantik aufweist, je nachdem, ob es sich bei A und B um String-Objekte oder um Array-Objekte handelt, gibt es einen weiteren Unterschied: Mit dem Zugriff auf die Komponenten eines Strings wird in jedem Fall ein Objekt eines Basistyps erreicht, nämlich ein Character. Bei Arrays ist das nicht so. Die Komponenten eines Arrays können wieder einen zusammengesetzten Typ aufweisen. Damit aus einem Prädikat über Arrays ein Prädikat über Basistypen werden kann ist darum oftmals ein mehrstufiger Abbildungsprozeß notwendig. Dieser Prozeß kann nur dann erfolgreich sein, wenn für die Komponenten eines Arrays und deren sämtliche Unterkomponenten die nach der Abbildung notwendigen Operationen auch definiert sind. So benötigt die Abbildung von A==B auf den Ausdruck3 z.B. die Operation Gleichheit für den Komponententyp der Arrays A und B.

Betrachtet man als Komponententyp die Typen String, Array und Record, so zeigt sich, daß hier die Abbildung problemlos ist, da alle Vergleichsoperatoren für die drei Typen definiert sind. Anders sieht dies für den Komponententyp Set aus. Hier fehlen die Relationen <, <=, >, >=. Das hat zur Folge, daß die oben dargestellten Abbildungen auf Prädikate über Basistypen hier keine Anwendung finden können. Folgende Wege mit diesem Problem umzugehen sind denkbar:

1. Objekte vom Typ Set werden als Komponenten von Arrays nicht weiter zugelassen. Damit werden die Möglichkeiten, komplexe zusammengesetzte Objekte zu bilden, eingeschränkt.
2. Die Vergleichsoperatoren <, <=, >, >= entfallen für Arrays.

3. Die Vergleichsoperatoren <, <=, >, >= können auf Arrays, die auf irgendeiner Stufe Sets als Komponenten aufweisen, keine Anwendung finden. Hierdurch werden die Ausdrucksmöglichkeiten für Invarianten eingeschränkt.

4. Die Vergleichsoperatoren <, <=, >, >= können zwar auch auf Arrays mit Set-Komponenten angewandt werden, jedoch ist eine Abbildung solcher Ausdrücke auf Prädikate über Basistypen dann nicht mehr möglich. Dies führt zu einer strukturabhängigen Verwaltung von Ausdrücken, die Arrays enthalten.

5.3 Record

Prädikate über Records lassen sich sehr einfach auf Prädikate über Basistypen abbilden. Den Abbildungen seien hier die beiden Records A und B zugrunde gelegt. Die Komponenten der beiden Records sind jeweils K_1 ... K_M. Beide Records müssen diese Komponenten aufweisen, da sonst die Vergleichsoperatoren nicht zugelassen sind[19]. Für Records sind nur wenige Operatoren vorgesehen, was die Zahl der möglichen Abbildungen überschaubar macht.

A==B , A!=B

Bei der Gleichheitsrelation kann die Abbildung auf folgenden Ausdruck erfolgen:

$A.K_1==B.K_1$ && (Ausdruck5)
$A.K_2==B.K_2$ &&
...
$A.K_M==B.K_M$

Für die Ungleichheit steht der Ausdruck !(Ausdruck5). Eine ausführliche äquivalente Form hierfür ist:

$A.K_1!=B.K_1$ ||
$A.K_2!=B.K_2$ ||
...
$A.K_M!=B.K_M$

A<B , A>B

Bei der Kleiner- und Größerbeziehung gibt es die Möglichkeit auf folgende Ausdrücke abzubilden:

$A.K_1<B.K_1$ && (Ausdruck6)
$A.K_2<B.K_2$ &&
...
$A,K_M<B.K_M$

$A.K_1>B.K_1$ && (Ausdruck7)
$A.K_2>B.K_2$ &&
...
$A.K_M>B.K_M$

A<=B , A>=B

Diese Ausdrücke können ersetzt werden durch solche, die sich aus Ausdruck5 und Ausdruck6 bzw. Ausdruck7 zusammensetzen. Also $Ausdruck5$ || $Ausdruck6$ für $A <= B$ und $Ausdruck5$ || $Ausdruck7$ für $A >= B$.

---

[19] Vgl. hierzu auch 4.7

Probleme der Abbildungen

Wie bei Arrays haben die hier dargestellten Möglichkeiten, Prädikate über Records auf Prädikate über Recordkomponenten abzubilden, den Nachteil, daß Prädikate über Basistypen eventuell erst nach mehreren Stufen der Abbildung erreicht werden. Das Ziel läßt sich überhaupt nur dann erreichen, wenn die Vergleichsoperatoren auch für die Komponententypen eines Records definiert sind. Probleme kann es wie bei den Arrays geben, wenn als Komponententyp der Typ Set auftaucht, da für diesen nicht alle Vergleichsoperatoren vorgesehen sind. Für dieses Problem gelten die bei der Abbildung von Prädikaten über Arrays in Abschnitt 5.2 aufgelisteten Entscheidungsmöglichkeiten. Eine erneute Aufzählung kann hier entfallen.

5.4 Set

Für Objekte vom Typ Set sind in 4.8 lediglich die Operatoren in, == und != für Prädikate eingeführt worden. Bei der Betrachtung von Abbildungen auf Basistypen müssen darum nur diese berücksichtigt werden. Abbildungen, wie sie oben für Strings, Arrays und Records aufgeführt wurden, sind aber davon abhängig, daß gezielt auf einzelne Komponenten zugegriffen werden kann. Dies ermöglicht auch ein systematisches Erreichen aller Komponenten. Hierzu gibt es aber bei dem hier eingeführten Typ Set keine Möglichkeit. Eine Folge hiervon ist, daß innerhalb der in dieser Arbeit eingeführten Grammatik keine Möglichkeit besteht einen Ausdruck anzugeben, der die gewünschte Abbildung darstellt. Prädikate über Objekte vom Typ Set lassen sich also nicht auf Prädikate von Basistypen abbilden.

6 Vereinfachung von Invarianten

Invarianten können nach der vorliegenden Grammatik sehr komplexe Ausdrücke sein. Für jedes Objekt können beliebig viele Invarianten vorliegen. Eine Aufgabe bei der Verwaltung besteht darin, bei jeder Veränderung an Objekten, für die Aussagen in einer oder mehreren Invarianten gemacht werden, sicherzustellen, daß durch die Änderung keine der Invarianten verletzt werden. Diese Überprüfung muß bei sich oft ändernden Objekten eventuell recht häufig ausgeführt werden. Hier ist es dann wichtig, daß eine möglichst kleine Anzahl von Invarianten bezüglich eines Objekts ausgewertet werden muß. An dieser Stelle setzt dieses Kapitel an. Es soll in möglichst vielen Fällen aufzeigen, wie einzelne Invarianten vereinfacht und mit anderen zusammengefaßt werden können.

Innerhalb einer Invariante tauchen abgesehen von den Operatoren unterschiedliche Elemente auf. Hierunter gibt es einmal globale Objekte, über die mit Hilfe der Invariante eine Aussage gemacht werden soll. Daneben gibt es für ein ConTract lokale Kontextvariablen. Da diese für ein ConTract allein zugänglich sind gibt es keine zu synchronisierenden Zugriffe. Für Kontextvariablen muß es darum keine Invarianten geben. Darüber hinaus tauchen auch Konstanten auf. Diese können soweit als möglich mit den aktuellen Werten von Kontextvariablen zusammengefaßt werden. Dies hat zur Folge, daß in den zu verwaltenden Invarianten nur noch globale Objekte mit ihrem Namen auftauchen. Ansonsten sind alle anderen Namen auf den aktuellen, konstanten Wert abgebildet.

Die Vereinfachungen sind zunächst nach den logischen Operatoren, die einzelne, einfache Invarianten verbinden geordnet. Nach der Grammatik kommen hier die Operatoren $NOT\ (!)$, $AND\ (\&\&)$ und $OR\ (||)$ in Frage. Unter einfachen Invarianten seien hier alle diejenigen verstanden, in denen keiner der drei logischen Operatoren auftaucht. Nach dieser Definition handelt es sich bei $A < 0$ um eine solche einfache Invariante, nicht aber bei $A < 0 || A > 1$. Diese einfachen Invarianten lassen sich in drei Komplexitätsstufen einteilen:

1. Die einfachste Form sind Invarianten mit einem Objektbezeichner auf nur einer Seite eines Vergleichsoperators. Auf der anderen Seite steht ein Ausdruck, der Kontextvariablen und Konstanten enthält.

2. Eine etwas komplexere Form stellen Invarianten dar, die auf einer Seite eines Vergleichsoperators einen Objektbezeichner aufweisen, der in einen Ausdruck mit Kontextvariablen und Konstanten eingebettet ist. Auf der anderen Seite steht ein Ausdruck nur mit Konstanten und Kontextvariablen.

3. Auf beiden Seiten stehen sowohl Objektbezeichner, Konstanten und Kontextvariablen. Hierbei handelt es sich um die komplexeste Form der Invarianten.

Die folgenden Betrachtungen konzentrieren sich auf einfache Invarianten der ersten Stufe. Diese sind in der Praxis am häufigsten anzutreffen, da hiermit die meisten Einschränkungen für ein Objekt angegeben werden können. Eine umfassende Bearbeitung der beiden komplexeren Stufen würde den Rahmen dieser Arbeit bei weitem übersteigen. Hier soll nur an einigen Beispielen gezeigt werden, welche Möglichkeiten prinzipiell bestehen.

Die folgende Untersuchung ist nach Datentypen geordnet, da die möglichen Zusammenfassungen vom Datentyp der Objekte abhängen können. Berücksichtigt werden nur Basistypen. Dies ist ausreichend, da vor den Zusammenfassungen ein Abbildungsprozeß, wie in Kapitel 5 beschrieben, durchgeführt werden sollte. Hiermit können die meisten Invarianten über zusammengesetzte Typen auf Invarianten über Basistypen abgebildet werden.

6.1 Vereinfachungen bei NOT

Eine Möglichkeit, einfachere Invarianten zu erhalten besteht darin, bei bestimmten Typen eine negierte einfache Invariante in die äquivalente nicht-negierte Form umzuwandeln. Dies ist möglich für die Typen: Integer, Floating, Character und Time. Die möglichen Vereinfachungen sind in Tabelle 2 aufgeführt. A soll hierbei ein Objekt eines der vier Basistypen sein. Bei x handelt es sich um einen Ausdruck desselben Typs, der aber keine Objektbezeichner, sondern lediglich Konstanten und Kontextvariablen enthält.

ursprüngliche Invariante	*neue Invariante*
! A < x	A >= x
!(A<x)	
! x > A	
!(x > A)	
! A <= x	A > x
!(A <= x)	
! x >= A	
!(x >= A)	
! A == x	A != x
!(A == x)	
! x == A	
!(x == A)	
! A != x	A == x
!(A != x)	
! x != A	
!(x != A)	
! A >= x	A < x
!(A >= x)	
! x <= A	
!(x <= A)	
! A > x	A <= x
!(A > x)	
! x < A	
!(x < A)	

Tabelle 2: Zusammenfassungen bei NOT-Operator

6.2 Vereinfachungen bei AND-Verknüpfung

Hier handelt es sich um die wohl am häufigsten anzutreffenden Vereinfachungen. Dies beruht auf der Tatsache, daß alle Invarianten von unterschiedlichen ConTracts als mit AND verknüpft zu betrachten sind, da die Invarianten aller ConTracts gleichzeitig gültig sein sollen.

6.2.1 Objekte vom Typ Integer

Integer-Objekte können unterschiedliche Dinge repräsentieren. Entsprechend weisen sie eine unterschiedliche Semantik auf. Hier sollen zwei verschiedene Arten von Integer-Objekten betrachtet werden:

1. Integer-Objekte, die ein Aggregat darstellen. Diese Objekte werden von der Anwendung als Zähler für reale Objekte benutzt. Entsprechend ist das Hinzufügen und Entnehmen von Objekten eine häufige Operation der Anwendung. Sie wird durch Addition oder Subtraktion auf das Integer-Objekt abgebildet. Da ein solches Integer-Objekt als Zähler dient ist sein Wertebereich eingeschränkt auf 0 und positive Zahlen zu betrachten.

   Beispiel: Integer-Objekt, das die Zahl freier Sitzplätze angibt.

2. Integer-Objekte, die kein Aggregat darstellen. Bei diesen ist für die Anwendung nur der aktuelle Wert des Objektes von Interesse. Es gibt keine Entsprechung für eine Anzahl realer Objekte.

   Beispiel: Integer-Objekt, das Flugkilometer angibt.

Entsprechend der unterschiedlichen Semantik von Integer-Objekten kann auch ein Unterschied in der Semantik für Invarianten über Integer-Objekte festgestellt werden. Dies soll an folgender Invariante näher betrachtet werden:

Beispiel: A >= 10

1. Handelt es sich bei A um ein Integer-Objekt vom Typ Aggregat, so besagt die Invariante, daß sichergestellt werden soll, daß solange die Invariante aktiv ist, mindestens 10 Objekte, die mit dem Integer-Objekt A gezählt werden, vorhanden sind. Anders ausgedrückt bedeutet die Invariante: Es sollen später noch 10 der mit A gezählten Objekte entnommen werden können.

2. Handelt es sich bei A um ein Integer-Objekt, das kein Aggregat repräsentiert, so besagt die Invariante lediglich, daß solange sie aktiv ist, der Wert von A nicht unter 10 absinken soll.

In den folgenden Abschnitten wird das Zusammenfassen von Invarianten bei Integer-Objekte für Aggregate und Nicht-Aggregate getrennt betrachtet. Es wird sich dabei zeigen, daß der Unterschied in der Semantik zu verschiedenen Möglichkeiten der Zusammenfassung führt.

6.2.1.1 Objekt repräsentiert kein Aggregat

Die folgende Tabelle 3 zeigt alle Kombinationen von Invarianten, die bei Integer-Objekten mit relationalen Operatoren möglich sind. Zu jeder Kombination ist angegeben, ob eine Zusammenfassung möglich ist oder ob die beiden Invarianten nicht vereinbar sind. Wie eine mögliche Zusammenfassung aussehen kann wird im Anschluß an die Tabelle für die einzelnen Kombinationen näher erläutert.

Folgendes wird für die Tabelle und die weitere Darstellung dieses Abschnitts vorausgesetzt:

- A stellt ein Integer-Objekt dar.
- x und y sind beliebige Integer-Ausdrücke, die aber kein Integer-Objekt enthalten, über das eine Aussage gemacht werden soll. Es handelt sich also um eine Zusammenfassung von Kontextvariablen und Konstanten. x und y decken hier sämtlichen Integerausdrücke ab, also z.B. auch *(Länge+25)*2-(Breite>>4)*. Voraussetzung ist hier lediglich, daß es sich bei *Länge* und *Breite* um Kontextvariablen handelt und nicht um Integer-Objekte, für die die Invariante gelten soll. Solche Integer-Objekte sollen bei den zunächst betrachteten Fällen schließlich nur auf einer Seite eines Vergleichsoperators stehen.

Die Einträge in der Tabelle 3 haben folgende Bedeutung:

ja — Eine Zusammenfassung der beiden Invarianten ist möglich. Wie diese aussehen kann wird in den folgenden Abschnitten beschrieben.

nein — Eine Zusammenfassung der beiden Invarianten ist nicht möglich. Sie müssen getrennt verwaltet und überwacht werden.

unvereinbar — Die beiden Invarianten sind unvereinbar, d.h. es kann nur die Einhaltung einer der beiden Invarianten garantiert werden. Die andere muß von der Verwaltungskomponente zurückgewiesen werden.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für*		
			x = y	*x ≺ y*	*y ≺ x*
1	A < x	A < y	ja	ja	ja
2	A < x	A <= y	ja	ja	ja
3	A < x	A == y	unvereinbar	unvereinbar	ja
4	A < x	A >= y	unvereinbar	unvereinbar	nein
5	A < x	A > y	unvereinbar	unvereinbar	nein
6	A <= x	A < y	ja	ja	ja
7	A <= x	A <= y	ja	ja	ja
8	A <= x	A == y	ja	nein	ja
9	A <= x	A >= y	ja	unvereinbar	nein
10	A <= x	A > y	unvereinbar	unvereinbar	nein
11	A == x	A < y	unvereinbar	ja	unvereinbar
12	A == x	A <= y	ja	ja	unvereinbar
13	A == x	A == y	ja	unvereinbar	unvereinbar
14	A == x	A >= y	ja	unvereinbar	ja
15	A == x	A > y	unvereinbar	unvereinbar	ja
16	A >= x	A < y	unvereinbar	nein	unvereinbar
17	A >= x	A <= y	ja	nein	unvereinbar
18	A >= x	A == y	ja	ja	unvereinbar
19	A >= x	A >= y	ja	ja	ja
20	A >= x	A > y	ja	ja	ja
21	A > x	A < y	unvereinbar	nein	unvereinbar
22	A > x	A <= y	unvereinbar	nein	nein
23	A > x	A == y	unvereinbar	ja	unvereinbar
24	A > x	A >= y	ja	ja	ja
25	A > x	A > y	ja	ja	ja

Tabelle 3: Mögliche Zusammenfassungen für Invarianten über Integerobjekte

Kombination 19: A >= x , A >= y

Beide Invarianten sollen gleichzeitig gelten. Dies führt zu folgender Aussage und Umformung:

$$
\begin{aligned}
& A \geq x \wedge A \geq y \\
\Leftrightarrow \quad & A = x + \Delta_1 \quad \wedge \quad A = y + \Delta_2 \quad \Delta_1 \geq 0 \quad \wedge \quad \Delta_2 \geq 0 \\
\Leftrightarrow \quad & x + \Delta_1 = y + \Delta_2 \quad \wedge \quad \Delta_1 \geq 0 \quad \wedge \quad \Delta_2 \geq 0 \\
\Leftrightarrow \quad & x + \Delta_1 = y + \Delta_2 \quad \wedge \quad \Delta_1 = y - x + \Delta_2 \quad \wedge \quad \Delta_1 \geq 0 \quad \wedge \quad \Delta_2 \geq 0 \\
\Leftrightarrow \quad & \Delta_2 = x - y + \Delta_1 \quad \wedge \quad \Delta_1 \geq y - x \quad \wedge \quad \Delta_1 \geq 0 \quad \wedge \quad \Delta_2 \geq 0 \qquad (1)
\end{aligned}
$$

Fall 1: $x > y$

$x > y$

$\Rightarrow y - x < 0 \wedge x - y > 0$

mit (1):

$\Rightarrow \Delta_1 \geq 0 \wedge \Delta_2 = x - y + \Delta_1$

$\Rightarrow \Delta_1 \geq 0 \wedge \Delta_2 \geq x - y > 0$

$\Rightarrow A \geq x$

Fall 2: $y > x$

$y > x$

$\Rightarrow y - x > 0 \wedge x - y < 0$

mit (1):

$\Rightarrow \Delta_1 \geq y - x \wedge \Delta_2 = x - y + \Delta_1$

$\Rightarrow \Delta_1 \geq y - x \wedge \Delta_2 \geq x - y + y - x = 0$

$\Rightarrow A \geq y$

Fall 3: $x = y$

$x = y \Rightarrow y - x = 0$

mit (1):

$\Rightarrow \Delta_1 \geq 0 \wedge \Delta_2 = x - y + \Delta_1 = 0 + \Delta_1 \geq 0$

$\Rightarrow A \geq x$

Zusammenfassung der Fälle 1, 2 und 3:

$\Rightarrow A \geq \max(x, y)$

Ohne Fallunterscheidung läßt sich aus (1) eine weitere Lösung ableiten:

$\Delta_1 \geq -x + |x| + |y|$

Nach (1) muß gelten: $\Delta_1 \geq y - x$:

$\Delta \geq -x + |x| + |y| \geq y - x$:

$\Leftrightarrow |x| + |y| \geq y$

$\Leftrightarrow |y| \geq y$

Nach (1) gilt: $\Delta_2 = x - y + \Delta_1$

$\Rightarrow \Delta_2 \geq x - y + (-x + |x| + |y|) = -y + |x| + |y|$

$\Rightarrow A \geq |x| + |y|$

Bei der Zusammenfassung der beiden Invarianten $A >= x$ und $A >= y$ gibt es beliebig viele Möglichkeiten. Zwei dieser Möglichkeiten sind hier herausgegriffen:

1. $A \geq \max(x, y)$

2. $A \geq |x| + |y|$

Ein Vergleich der beiden Möglichkeiten erfolgt nach dem Abschnitt, der sich mit Integer-Objekten beschäftigt, die Aggregate repräsentieren.

Kombination 24: A > x , A >= y

$A > x$ kann auch als $A >= x + 1$ dargestellt werden. Damit ergeben sich wie oben die beiden möglichen Zusammenfassungen:

1. $A \geq \max(x+1, y)$
2. $A \geq |x+1| + |y|$

Kombination 20: A >= x , A > y

$A > y$ kann auch als $A >= y + 1$ dargestellt werden. Damit ergeben sich wie oben die beiden möglichen Zusammenfassungen:

1. $A \geq \max(x, y+1)$
2. $A \geq |x| + |y+1|$

Kombination 25: A > x , A > y

Dieser Fall stellt die Kombination der beiden vorangegangenen dar. Als mögliche Zusammenfassungen ergeben sich damit:

1. $A \geq \max(x+1, y+1)$
2. $A \geq |x+1| + |y+1|$

Kombination 7: A <= x , A <= y

Für diesen Fall lassen sich die entsprechenden Berechnungen wie bei $A >= x, A >= y$ durchführen. Als mögliche Zusammenfassungen ergeben sich dann:

1. $A \leq \min(x, y)$
2. $A \leq -|x| - |y|$

Kombination 2: A < x , A <= y

Kombination 6: A <= , A < y

Kombination 1: A < x , A < y

Für diese drei Fälle gilt, daß $A < x$ auch als $A <= x - 1$ und $A < y$ als $A <= y - 1$ geschrieben werden kann. Die möglichen Zusammenfassungen ergeben sich dann wie bei den entsprechenden Fällen von > und >=.

Kombinationen 3, 8, 9, 11, 12, 13, 14, 15, 17, 18, 23

Diese Kombinationen sollen hier gemeinsam betrachtet werden. Ihnen allen ist gemeinsam, daß eine Zusammenfassung nur abhängig von den Werten x und y möglich ist. Die möglichen Zusammenfassungen sind in Tabelle 4 angegeben.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für* $x = y$	$x < y$	$y < x$
3	A < x	A == y	unvereinbar	unvereinbar	A==y
8	A <= x	A == y	A==x	nein	A==y
9	A <= x	A >= y	A==x	unvereinbar	nein
11	A == x	A < y	unvereinbar	A==x	unvereinbar
12	A == x	A <= y	A==x	A==x	unvereinbar
13	A == x	A == y	A==x	unvereinbar	unvereinbar
14	A == x	A >= y	A==x	unvereinbar	A==x
15	A == x	A > y	unvereinbar	unvereinbar	A==x
17	A >= x	A <= y	A==x	nein	unvereinbar
18	A >= x	A == y	A==x	A==y	unvereinbar
23	A > x	A == y	unvereinbar	A==y	unvereinbar

Tabelle 4: Zusammenfassungen für Nicht-Aggregate

6.2.1.2 Objekt repräsentiert ein Aggregat

In diesem Abschnitt sollen die entsprechenden Vereinfachungen für die Kombinationen aus Tabelle 3 dargestellt werden, wenn es sich beim Objekt A um ein Integer-Objekt handelt, das ein Aggregat repräsentiert.

Kombination 19: A >= x , A >= y

Bei der Betrachtung dieser Kombination muß eine mögliche Zusammenfassung ausgehend von der Semantik der einzelnen Invarianten gefunden werden. $A >= x$ besagt, daß es bis zum Ende des ConTracts möglich sein muß von A den Wert x abzuziehen. Danach hat A dann den Wert $A - x$ und das soll, da es sich bei A um ein Aggregat handelt größer als 0 sein. Entsprechendes gilt für $A >= y$. Darüber hinaus muß es möglich sein, daß sowohl x als auch y von A abgezogen wird, ohne daß dieses negativ wird. Berücksichtigt man als 4. Fall, daß weder x noch y von A subtrahiert werden, so ergibt sich als Bedingung für A:

$$
\begin{aligned}
& A \geq 0 \wedge A - x \geq 0 \wedge A - y \geq 0 \wedge A - x - y \geq 0 \\
\Leftrightarrow\ & A = \Delta_1 \wedge A = x + \Delta_2 \wedge A = y + \Delta_3 \wedge A = x + y + \Delta_4 \wedge \Delta_i \geq 0 \\
\Leftrightarrow\ & \Delta_1 = x + \Delta_2 \wedge \Delta_1 = y + \Delta_3 \wedge \Delta_1 = x + y + \Delta_4 \wedge x + \Delta_2 = y + \Delta_3 \wedge x + \Delta_2 = x + y + \Delta_4 \wedge y + \Delta_3 = x + y + \Delta_4 \wedge \Delta_i \geq 0 \\
\Leftrightarrow\ & \Delta_1 = x + \Delta_2 \wedge \Delta_1 = y + \Delta_3 \wedge \Delta_1 = x + y + \Delta_4 \wedge \Delta_2 = y - x + \Delta_3 \wedge \Delta_2 = y + \Delta_4 \wedge \Delta_3 = x + \Delta_4 \wedge \Delta_i \geq 0 \\
\Rightarrow\ & \Delta_2 < \Delta_1 \wedge \Delta_3 < \Delta_1 \wedge \Delta_4 < \Delta_1 \wedge \Delta_4 < \Delta_2 \wedge \Delta_4 < \Delta_3 \wedge \Delta_i \geq 0
\end{aligned}
$$

Damit ist Δ_4 als kleinster Wert aus Δ_1, Δ_2, Δ_3 und Δ_4 identifiziert. Setzt man für diesen den kleinstmöglichen Wert, nämlich 0 ein, so ergibt sich weiter:

$$
\begin{aligned}
\Delta_4 &= 0 \\
\Delta_3 &= x + \Delta_4 = x \\
\Delta_2 &= y + \Delta_4 = y \\
\Delta_1 &= x + y + \Delta_4 = x + y
\end{aligned}
$$

Hieraus folgt für A: $A = x + y$ als minimaler Wert. Den minimalen Wert für A zu suchen ist sinnvoll, da dieser die kleinstmögliche Einschränkung der auf A noch möglichen Operationen bedeutet. Als Zusammenfassung der Invarianten $A >= x$ und $A >= y$ ergibt sich: $A >= x + y$.

Kombination 24: A > x , A >= y

Kombination 20: A>=x , A>y

Kombination 25: A>x , A>y

Diese drei Fälle können auf die Kombination 19 abgebildet werden. Hierzu muß $A > x$ als $A >= x + 1$ und $A > y$ als $A >= y + 1$ geschrieben werden. Dies sind jeweils gleichbedeutende Aussagen, solange es sich bei A um ein Objekt vom Typ Integer handelt. Als Zusammenfassungen der beiden Invarianten ergeben sich dann: $A >= x + y + 1$, $A >= x + y + 1$ bzw. $A >= x + x + 2$.

Kombination 7: A<=x , A<=y

Hier muß zunächst einmal festgelegt werden, welche Semantik der Ausdruck $A <= x$ besitzt, wenn es sich bei A um ein Integer-Objekt vom Typ Aggregat handelt. Als Bedeutung soll hier folgendes angenommen werden: A<=x besagt, daß solange die Invariante wirksam ist, nicht mehr als x, der mit A gezählten Objekte, zu entnehmen sein sollen. Dies führt zu folgenden Bedingungen für A:

$$
\begin{array}{lccccccc}
 & A - x \le 0 & \wedge & A - y \le 0 & & & & \\
\Leftrightarrow & A - x + \Delta_1 = 0 & \wedge & A - y + \Delta_2 = 0 & \wedge & \Delta_1 \ge 0 & \wedge & \Delta_2 \ge 0 \\
\Leftrightarrow & A = x - \Delta_1 & \wedge & A = y - \Delta_2 & \wedge & \Delta_1 \ge 0 & \wedge & \Delta_2 \ge 0 \\
\Leftrightarrow & x - \Delta_1 = y - \Delta_2 & \wedge & \Delta_1 = x - y + \Delta_2 & \wedge & \Delta_1 \ge 0 & \wedge & \Delta_2 \ge 0 \\
\Leftrightarrow & \Delta_2 = y - x + \Delta_1 & \wedge & \Delta_1 \ge x - y & \wedge & \Delta_1 \ge 0 & \wedge & \Delta_2 \ge 0
\end{array}
$$

Mit einer Fallunterscheidung, wie bei Kombination 19 für Nichtaggregate ergibt sich dann $A <= min(x, y)$ als Zusammenfassung der beiden Invarianten $A <= x$ und $A <= y$.

Kombination 2: A < x , A <= y

Kombination 6: A <= , A < y

Kombination 1: A < x , A < y

Diese Fälle können auf die Kombinatione 7 abgebildet werden, indem $A < x$ als $A <= x - 1$ und $A < y$ als $A <= y - 1$ geschrieben werden. Als zusammengefaßte Invarianten ergeben sich dann: $A <= min(x - 1, y)$, $A <= min(x, y - 1)$ bzw. $A <= min(x - 1, y - 1)$.

Kombinationen 3,8,11,12,13,14,15,18,23

Diesen neun Kombinationen ist gemeinsam, daß mindestens eine der Invarianten die Gleichheit zu einem konstanten Wert fordert. Der Inhalt von Objekt A ist also nicht mehr veränderbar, ohne diese Invariante zu verletzen. Lautet nun die zweite Invariante $A <= y$ (Kombination 12) oder $A <= y$ (Kombination 14), so wird hiermit ausgesagt, daß einige der mit A gezählten Objekte vor Ende des ConTracts entnommen werden sollen. Dies ist aber nicht möglich, wenn A auf einen bestimmten Wert „eingefroren“ ist. Die Invarianten sind also unvereinbar. Es sind bei den hier betrachteten Kombinationen nur noch wenige Zusammenfassungen möglich, wie in Tabelle 5 dargestellt.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für*		
			x = y	*x < y*	*y < x*
3	A < x	A == y	unvereinbar	unvereinbar	unvereinbar
8	A <= x	A == y	unvereinbar	unvereinbar	unvereinbar
11	A == x	A < y	unvereinbar	unvereinbar	unvereinbar
12	A == x	A <= y	unvereinbar	unvereinbar	unvereinbar
13	A == x	A == y	A==x	unvereinbar	unvereinbar

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für* $x = y$	$x \prec y$	$y \prec x$
14	A == x	A >= y	unvereinbar	unvereinbar	unvereinbar
15	A == x	A > y	unvereinbar	unvereinbar	unvereinbar
18	A >= x	A == y	unvereinbar	unvereinbar	unvereinbar
23	A > x	A == y	unvereinbar	unvereinbar	unvereinbar

Tabelle 5: Zusammenfassungen für Aggregate

Kombination 9: A<=x , A>=y

Kombination 17: A<=x , A<=y

Bei diesen beiden symmetrischen Kombinationen gibt es nur dann eine Möglichkeit die Invarianten zusammenzufassen, wenn $x = y$ gilt. Damit können diese Kombinationen wie bei Nicht-Aggregaten behandelt werden. Hierzu sei auf Tabelle 4 verwiesen.

6.2.1.3 Vergleich möglicher Zusammenfassungen bei Aggregaten und Nicht-Aggregaten

Ein Vergleich der möglichen Zusammenfassungen für aggregierte und nicht-aggregierte Integer-Objekte zeigt, daß drei Fälle unterschieden werden können:

- Die Überlegungen führen sowohl bei Aggregaten als auch bei Nicht-Aggregaten zu demselben Ergebnis. Eine Zusammenfassung kann also ohne Kenntnisse darüber erfolgen, ob A ein aggregiertes Integer-Objekt ist oder nicht. Hierunter fallen die Kombinationen 9,13 und 17.

- Es gibt unterschiedliche Ergebnisse in den Abschnitten 6.2.1.1 und 6.2.1.2, aber es existiert jeweils mindestens eine Zusammenfassung, die für beide Fälle möglich ist. Diese kann dann Anwendung finden unabhängig davon, ob ein aggregiertes Integer-Objekt vorliegt oder nicht. In diesen Bereich fallen die Kombinationen 19, 20, 24 und 25, sowie 1, 2, 6 und 7. Bei der Kombination 19 ist beispielsweise die Zusammenfassung $A \geq |x| + |y|$ in jedem Fall möglich.

- Die unterschiedlichen Ergebnisse für aggregierte und nicht-aggregierte Integer-Objekte schließen sich gegenseitig aus. Dies bedeutet für die Verwaltungskomponente, daß ihr in diesen Fällen bekannt sein muß um welche Art von Integer-Objekten es sich handelt. Alternativ können in all diesen Fällen die beiden Invarianten als unvereinbar erklärt werden. Somit muß mindestens eine von beiden abgelehnt werden. Dieses Problem tritt bei den Kombinationen 3, 8, 11, 12, 14, 15, 18 und 23 auf.

Unter Berücksichtigung dieser drei Fälle läßt sich dann als Ergebnis der Abschnitte 6.2.1.1 und 6.2.1.2 die Tabelle 6 mit Zusammenfassungen für Invarianten über Integer-Objekte angeben. Die beschriebenen Konflikte zwischen den Ergebnissen der beiden Abschnitte sind mit dem Eintrag *Konflikt* gekennzeichnet.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für* $x = y$	$x \prec y$	$y \prec x$
1	A < x	A < y		A<=min(x-1,y-1)	
2	A < x	A <= y		A<=min(x-1,y)	
3	A < x	A == y	unvereinbar	unvereinbar	Konflikt
4	A < x	A >= y	unvereinbar	unvereinbar	nein
5	A < x	A > y	unvereinbar	unvereinbar	nein
6	A <= x	A < y		A<=min(x,y-1)	
7	A <= x	A <= y		A<=min(x,y)	
8	A <= x	A == y	Konflikt	nein	Konflikt

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für* $x = y$	$x \prec y$	$y \prec x$
9	A <= x	A >= y	A==x	unvereinbar	nein
10	A <= x	A > y	unvereinbar	unvereinbar	nein
11	A == x	A < y	unvereinbar	Konflikt	unvereinbar
12	A == x	A <= y	Konflikt	Konflikt	unvereinbar
13	A == x	A == y	A==x	unvereinbar	unvereinbar
14	A == x	A >= y	Konflikt	unvereinbar	Konflikt
15	A == x	A > y	unvereinbar	unvereinbar	Konflikt
16	A >= x	A < y	unvereinbar	nein	unvereinbar
17	A >= x	A <= y	A==x	nein	unvereinbar
18	A >= x	A = y	Konflikt	Konflikt	unvereinbar
19	A >= x	A >= y		$A \geq \lvert x \rvert + \lvert y \rvert$	
20	A >= x	A > y		$A \geq \lvert x \rvert + \lvert y + 1 \rvert$	
21	A > x	A < y	unvereinbar	nein	unvereinbar
22	A > x	A <= y	unvereinbar	nein	nein
23	A > x	A == y	unvereinbar	Konflikt	unvereinbar
24	A > x	A >= y		$A \geq \lvert x + 1 \rvert + \lvert y \rvert$	
25	A > x	A > y		$A \geq \lvert x + 1 \rvert + \lvert y + 1 \rvert$	

Tabelle 6: Allgemeine Zusammenfassungen für Integerobjekte

6.2.1.4 Zusätzliche Vergleichsoperatoren

Die Tabelle 6 stellt zusammen, wie Integer-Objekte mit unterschiedlicher Semantik bei der Zusammenfassung von Invarianten möglichst gleich behandelt werden können. Neben den dort dargestellten Möglichkeiten kann ein grundsätzlich anderer Weg gegangen werden. Dies ist die Einführung zusätzlicher Vergleichsoperatoren. Als solche seien hier ´< und ´> betrachtet. Ihre Semantik ist:

A´>x : A soll um x größer sein, als dies durch bisherige Invarianten gefordert ist.

A´<x : A soll um x kleiner sein, als dies durch bisherige Invarianten gefordert ist.

Beispiel: Eine erste Invariante könnte z.B. *A >= 10* lauten. Kommt als zweite Invariante *A ´> 5* hinzu, so wird die bisherige Untergrenze für A um 5 erhöht. Als Invariante genügt es dann *A > 15* zu verwalten.

Die Einführung solcher relativen Operatoren bietet den Vorteil, daß für alle Integer-Objekte die Zusammenfassungen, wie oben für Nicht-Aggregate aufgeführt, verwendet werden können. Die Tatsache, daß es sich bei einem Objekt um ein Aggregat handelt wird hier durch den Operator ausgedrückt. Dieser sorgt dafür, daß z.B. bei A´>x die benötigte Anzahl x der mit A gezählten Objekte reserviert wird.

6.2.2 Objekte vom Typ Floating

Für Objekte vom Typ Floating können keine sich in der Semantik unterscheidenden Arten gefunden werden. Die Betrachtung von Aggregaten macht nur für Ganzzahlwerte Sinn. Mögliche Zusammenfassungen lassen sich wie bei Integer-Objekten, die keine Aggregate darstellen, herleiten. Für die Kombination 19 z.B. wird hier $A >= max(x, y)$ als Zusammenfassung genutzt. Die entsprechenden Zusammenfassungen für die anderen Kombinationen sind in Tabelle 7 aufgeführt. Hierbei handelt es sich nicht um speziell für den Typ Floating bestehende Möglichkeiten zur Zusammenfassung. Die Tabelle 7 hat ihre Gültigkeit genauso für Integer-Objekte, die keine Aggregate darstellen, für Character-Objekte und für Time-Objekte, also für alle hier betrachteten skalaren Typen.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für* $x = y$	$x < y$	$y < x$
1	A < x	A < y	A<x	A<x	A<y
2	A < x	A <= y	A<x	A<x	A<=y
3	A < x	A == y	unvereinbar	unvereinbar	A==y
4	A < x	A >= y	unvereinbar	unvereinbar	nein
5	A < x	A > y	unvereinbar	unvereinbar	nein
6	A <= x	A < y	A<y	A<=x	A<y
7	A <= x	A <= y	A<=x	A<=x	A<=y
8	A <= x	A == y	A==x	nein	A==y
9	A <= x	A >= y	A==x	unvereinbar	nein
10	A <= x	A > y	unvereinbar	unvereinbar	nein
11	A == x	A < y	unvereinbar	A==x	unvereinbar
12	A == x	A <= y	A==x	A==x	unvereinbar
13	A == x	A == y	A==x	unvereinbar	unvereinbar
14	A == x	A >= y	A==x	unvereinbar	A==x
15	A == x	A > y	unvereinbar	unvereinbar	A==x
16	A >= x	A < y	unvereinbar	nein	unvereinbar
17	A >= x	A <= y	A==x	nein	unvereinbar
18	A >= x	A = y	A==x	A==y	unvereinbar
19	A >= x	A >= y	A>=x	A>=y	A>=x
20	A >= x	A > y	A>x	A>y	A>=x
21	A > x	A < y	unvereinbar	nein	unvereinbar
22	A > x	A <= y	unvereinbar	nein	nein
23	A > x	A == y	unvereinbar	A==y	unvereinbar
24	A > x	A >= y	A>x	A>x	A>=y
25	A > x	A > y	A>x	A>x	A>y

Tabelle 7: Zusammenfassungen für skalare Typen

6.2.3 Objekte vom Typ Character

Für Invarianten über Objekte dieses Typs gelten die Möglichkeiten der Zusammenfassung bei skalaren Typen. Diese sind in Tabelle 7 dargestellt.

6.2.4 Objekte vom Typ Time

Hier gelten zunächst die möglichen Zusammenfassungen für beliebige skalare Typen, wie sie in Tabelle 7 aufgeführt sind. Zusätzlich muß betrachtet werden, ob auch zwei Invarianten mit zeitlich beschränkter Gültigkeit zusammengefaßt werden können. Dies soll am Beispiel der folgenden Invarianten geschehen:

A>0 after 1.1.1995:14.00.00 und A>0 after 1.1.1995:13.00.00

Werden diese beiden Invarianten der Verwaltungskomponente übergeben, so ist es sicherlich ausreichend, wenn die Einhaltung der ersten Invariante ständig überprüft wird. Da die zweite Invariante lediglich eine abgeschwächte Version der ersten darstellt, ist ihre Einhaltung mit der Einhaltung der ersten automatisch garantiert. Solange beide Invarianten gelten müssen, kann also die erste als eine Zusammenfassung beider gesehen werden. Weitere Zusammenfassungen in diesem Stil sind in Tabelle 8 aufgeführt. Hierbei ist zu beachten, daß immer vorausgesetzt wird, daß vor dem Operator, der die zeitliche Beschränkung angibt dasselbe Prädikat A steht. x und y seien beliebige Ausdrücke, wie sie in der Grammatik als *time-expression* angegeben sind. Diese Ausdrücke müssen zu dem Zeitpunkt, zu dem die Invarianten an die Verwaltungskomponente übergeben werden zu konstanten Ausdrücken ausgewertet werden können.

1. einfache Invariante	*2. einfache Invariante*	*neue Invariante* $x = y$	$x < y$	$x > y$
A before x	A before y	A before x	A before x	A before y
A before x	A at y	unvereinbar	unvereinbar	A at y
A before x	A after y	unvereinbar	unvereinbar	nein
A at x	A before y	unvereinbar	A at x	unvereinbar
A at x	A at y	A at x	unvereinbar	unvereinbar
A at x	A after y	unvereinbar	unvereinbar	A at x
A after x	A before y	unvereinbar	nein	unvereinbar
A after x	A at y	unvereinbar	A at y	unvereinbar
A after x	A after y	A after x	A after y	A after x

Tabelle 8: Zusammenfassungen bei zeitlich beschränkten Invarianten (AND)

6.3 Vereinfachungen bei OR-Verknüpfung

In Abschnitt 6.2 wurden einfache durch AND verknüpfte Invarianten betrachtet. Hier sollen nun die entsprechenden Überlegungen bei Verknüpfung mit OR folgen. Der Aufbau dieses Abschnitts orientiert sich an Abschnitt 6.2.

6.3.1 Objekte vom Typ Integer

Für Integer-Objekte muß wieder die unterschiedliche Semantik für aggregierte und nicht-aggregierte Objekte beachtet werden. Die für diese beiden Fälle möglichen Zusammenfassungen sollen beispielhaft für die Kombination 19 hergeleitet werden. Für alle anderen Kombinationen kann aus Tabelle 9 abgelesen werden, wie Invarianten zusammengefaßt werden können.

A sei ein Objekt vom Typ Integer. x und y stehen wieder für Integerausdrücke, wie unter 6.2.1.1 beschrieben. Zunächst wird A nicht als Aggregat betrachtet. Bei Kombination 19 sind die beiden Invarianten $A >= x$, sowie $A >= y$ zusammenzufassen. Als Bedingung für A ergibt sich hier:

$$
\begin{aligned}
& A \geq x \quad \vee \quad A \geq y \\
\Leftrightarrow \quad & (\quad x = y \quad \wedge \quad A \geq x \quad) \\
\vee \quad & (\quad x < y \quad \wedge \quad A \geq x \quad) \\
\vee \quad & (\quad x > y \quad \wedge \quad A \geq y \quad) \\
\Leftrightarrow \; & A \geq \min(x, y)
\end{aligned}
$$

Diese Zusammenfassung schränkt den Wertebereich für A so wenig wie möglich ein. Sie ist darum allen anderen Zusammenfassungen mit $A \geq z$ und $z > \min(x, y)$ vorzuziehen.

Betrachtet man A als ein Aggregat, so bedeutet die Invariante $A >= x \;||\; A >= y$, daß entweder x oder y von A abgezogen werden soll. Dies muß möglich sein, ohne daß A kleiner als 0 wird. Hier ergibt sich als Bedingung:

$$\begin{aligned} & A - x \geq 0 \quad \vee \quad A - y \geq 0 \\ \Leftrightarrow \quad & A \geq x \quad \vee \quad A \geq y \\ \Leftrightarrow \quad & A \geq \min(x, y) \end{aligned}$$

Damit führen die Überlegungen für Aggregate zu denselben Ergebnissen wie für Nicht-Aggregate. Dies kann entsprechend für alle anderen Kombinationen nachgewiesen werden. Die Ergebnisse sind in Tabelle 9 zusammengestellt. Hier tritt als Eintragung zusätzlich zu den bisherigen *Aufhebung* auf. Dies kennzeichnet Fälle, in denen die beiden durch OR verknüpften Invarianten zusammen keine Einschränkung für das Objekt A darstellen. Beispielsweise ist bei $A >= x \;||\; A < x$ jeder beliebige Wert für A zugelassen. Da bei der OR-Verknüpfung immer nur mindestens eine der Invarianten gelten muß, gibt es keine Fälle, in denen eine Unvereinbarkeit der beiden Invarianten festgestellt werden kann.

Kombination	*Invariante 1*	*Invariante 2*	*Zusammenfassung für*		
			x = y	*x < y*	*y < x*
1	A < x	A < y		A<max(x,y)	
2	A < x	A <= y	A<=x	A<=y	A<x
3	A < x	A == y	A<=x	nein	A<x
4	A < x	A >= y	Aufhebung	nein	Aufhebung
5	A < x	A > y	A!=x	nein	Aufhebung
6	A <= x	A < y	A<=x	A<y	A<=x
7	A <= x	A <= y		A<=max(x,y)	
8	A <= x	A == y	A<=x	nein	A<=x
9	A <= x	A >= y	Aufhebung	nein	Aufhebung
10	A <= x	A > y	Aufhebung	nein	Aufhebung
11	A == x	A < y	A<=x	A<y	nein
12	A == x	A <= y	A<=x	A<=y	nein
13	A == x	A == y	A==x	nein	nein
14	A == x	A >= y	A>=x	nein	A>=y
15	A == x	A > y	A>=x	nein	A>y
16	A >= x	A < y	Aufhebung	Aufhebung	nein
17	A >= x	A <= y	Aufhebung	Aufhebung	nein
18	A >= x	A = y	A>=x	A>=x	nein
19	A >= x	A >= y		A>=min(x,y)	
20	A >= x	A > y	A>=x	A>=x	A>y
21	A > x	A < y	A!=x	Aufhebung	nein
22	A > x	A <= y	Aufhebung	Aufhebung	nein
23	A > x	A == y	A>=x	A>x	nein
24	A > x	A >= y	A>=x	A>x	A>=y
25	A > x	A > y		A>min(x,y)	

Tabelle 9: Zusammenfassungen bei OR-Verknüpfung

6.3.2 Objekte vom Typ Floating oder Character

Wie bei der Verknüpfung mit dem AND-Operator sind auch hier für beide Datentypen dieselben Zusammenfassungen möglich. Hierbei handelt es sich um Zusammenfassungen, die für jeden skalaren Typ zulässig sind. Hierunter fällt auch der Typ Integer. Für diesen wurde im vorangegangenen Abschnitt dargestellt, daß hier nicht zwischen aggregierten und nicht-aggregierten Objekten unterschieden werden muß. Es gibt für Integer-Objekte also eindeutige Ergebnisse. Diese können darum auch für Invarianten über Objekte der Typen Floating und Character Anwendung finden. Hierzu sei auf Tabelle 9 verwiesen.

6.3.3 Objekte vom Typ Time

Für Invarianten über Objekte dieses Typs gelten die Zusammenfassungen, wie sie in Tabelle 9 für einen beliebigen skalaren Typ aufgeführt sind. Zusätzlich soll hier noch betrachtet werden, wie zeitlich beschränkte Invarianten zusammengefaßt werden können. Die Möglichkeiten, die sich hier ergeben, sind in Tabelle 10 zusammengefaßt.

1. einfache Invariante	*2. einfache Invariante*	*neue Invariante*		
		$x = y$	$x < y$	$y<x$
A before x	A before y	A before x	A before y	A before x
A before x	A at y	nein	nein	A before x
A before x	A after y	!(A at x)	nein	nein
A at x	A before y	nein	A before y	nein
A at x	A at y	A at x	nein	nein
A at x	A after y	nein	nein	A after y
A after x	A before y	!(A at x)	nein	nein
A after x	A at y	nein	A after x	nein
A after x	A after y	A after x	A after x	A after y

Tabelle 10: Zusammenfassungen bei zeitlich beschränkten Invarianten (OR)

Damit sind alle möglichen Zusammenfassungen für Invarianten der einfachen Form aufgeführt. Wie bereits erwähnt sollen zulässige Komprimierungen für komplexe Invarianten nicht ausführlich betrachtet, sondern nur kurz skizziert werden.

6.4 Komplexere Invarianten

Die zusätzliche Komplexität resultiert hier aus der Tatsache, daß die Invariante auf jeder Seite eines Vergleichsoperators mehr als einen Objektbezeichner aufweist. Oder aber die Invariante betrifft gleich mehrere Objekte. Als Beispiel seien hier die Invarianten $A + x >= y$ und $A >= z$ über das Integer-Objekt A betrachtet. Sie sollen mit AND verknüpft sein. Hier läßt sich eine Zusammenfassung leicht erreichen, wenn die erste Invariante zunächst in $A >= y - x$ umgeformt wird. Dann sind die Möglichkeiten aus Abschnitt 6.2 wieder anwendbar. Die sich ergebende Invariante lautet $A >= y - x + z$. Für andere komplexe Invarianten sind ähnliche Zusammenfassungen, die auf arithmetischen Umformungen basieren denkbar. Die Betrachtung von Vereinfachungsmöglichkeiten für Invarianten soll hier beendet werden und zu anderen Fragen der Invariantenverwaltung übergegangen werden.

7 Invariantenverwaltung

In diesem Kapitel werden verschiedene Fragen erörtert, die mit einer sinnvollen Verwaltung von Invarianten zusammenhängen. Hierzu gehört die Frage, in welchem APRICOTS-Modul die Verwaltungskomponente untergebracht werden kann. Darüber hinaus sollen Auswirkungen von Veränderungen an Prädikaten auf die Verwaltungskomponente betrachtet werden. Anschließend wird geklärt, welche Daten zusammen mit den einzelnen Invarianten gespeichert werden müssen und wie geeignete Datenstrukturen aussehen können.

7.1 Position der Verwaltungskomponente

Zwei der in Abbildung 3 dargestellten APRICOTS-Module kommen als Position der Verwaltungskomponente für Invarianten in Frage. Dies ist auf der einen Seite der ConTract-Manager und auf der anderen Seite sind es die Resource-Manager. Hierdurch ergibt sich eine zentrale und eine dezentrale Anordnung. Ein Vergleich der beiden Lösungen soll Vor- und Nachteile beleuchten.

7.1.1 Zentrale Verwaltungskomponente

Hier erfolgt das Invarianten-Management direkt im ConTract-Manager. Dieser übergibt bei der Abarbeitung des Skripts die dort enthaltenen Invarianten an den Invarianten-Manager. Dort wird für die Speicherung der Invarianten gesorgt. Außerdem ist diese Komponente dafür verantwortlich, bei jeder Änderung eines Objekts, die für dieses Objekt vorhandenen Invarianten zu überprüfen. Wird durch den Zugriff auf das Objekt die Invariante verletzt, so muß dieser verhindert werden[20]. Hierzu dient die Kommunikation zwischen Invarianten-Manager und den Resource-Managern. Der Aufbau ist schematisch in Abbildung 5 dargestellt.

Vor- und Nachteile dieser Lösung:

+ Alle Invarianten werden zentral von einer Komponente verwaltet. Mechanismen zur Vereinfachung und Zusammenfassung von Invarianten können dann direkt auf alle Invarianten zugreifen. Bei der Veränderung von Invarianten ist keine Kommunikation mit den Resource-Managern notwendig.

+ Es gibt keine Probleme bei der Verwaltung von Invarianten, die gleichzeitig mehrere Objekte betreffen. Siehe hierzu auch die dezentrale Lösung.

– Die Invarianten müssen bei jeder Änderung eines Objekts erneut überprüft werden. Mit der Durchführung der Zugriffe auf Objekte wird der zuständige Resource-Manager beauftragt. Dieser muß hier zunächst dem Invarianten-Manager mitteilen, welche Änderung des Objekts beabsichtigt ist. Der Invarianten-Manager stößt dann die Überprüfung an und stellt fest, ob eine Invariantenverletzung vorliegt. Entsprechend teilt er dem Resource-Manager mit, ob die Änderung zulässig ist oder nicht. Änderungsoperationen können nun für viele auf verschiedene Resource-Manager verteilte Objekte gleichzeitig ausgeführt werden. Hierdurch kann eine enorme

[20] Vgl. andere Strategien in Abschnitt 2.2.4

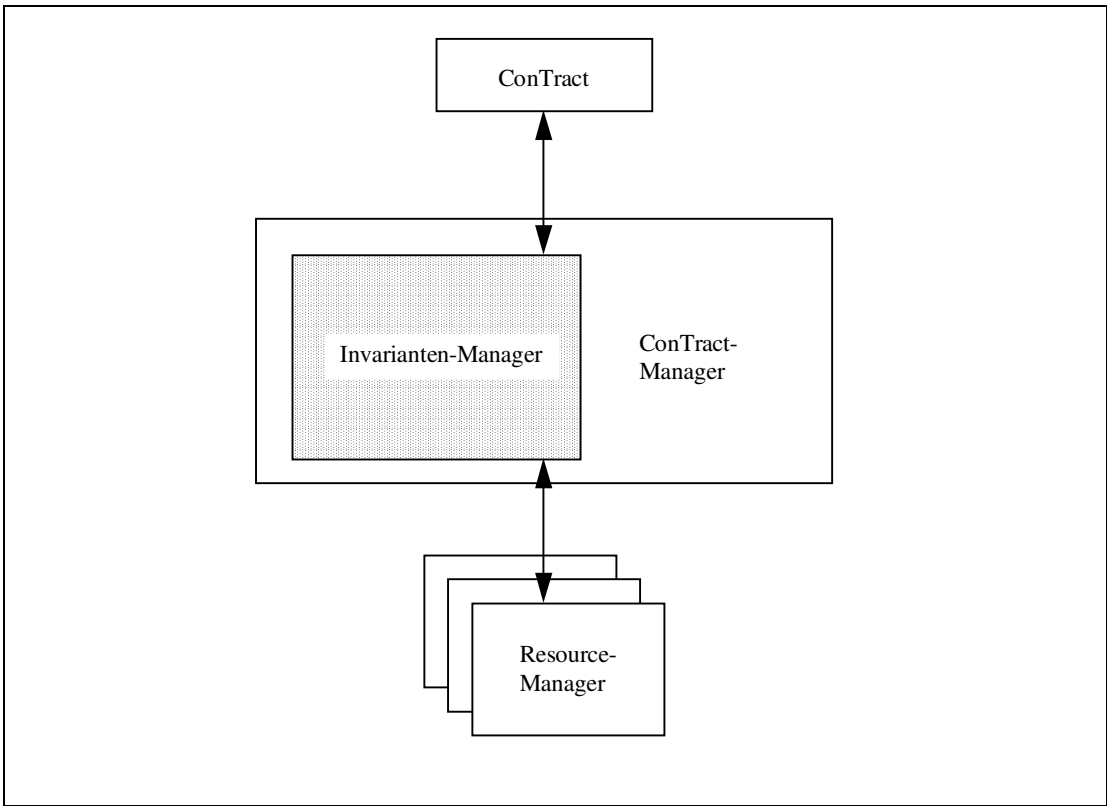

Abbildung 5: Zentrale Verwaltungskomponente

Anzahl von Nachrichten zwischen Invarianten-Manager und Resource-Managern notwendig werden. Der Invarianten-Manager wird zum Flaschenhals.

7.1.2 Dezentrale Verwaltungskomponente

Diese Version positioniert den Invarianten-Manager direkt in jedem Resource-Manager, wie in Abbildung 6 gezeigt. Eine neu hinzukommende Invariante wird hier im ConTract-Manager nur kurz verarbeitet. Hier können z.B. erste Vereinfachungen erfolgen. Ansonsten ist der ConTract-Manager nur dafür verantwortlich, die Invariante dem richtigen Resource-Manager zu übergeben. Dort wird sie gespeichert und die Überprüfung ihrer Gültigkeit bei Bedarf durchgeführt.

Vor- und Nachteile dieser Lösung:

+ Die Unterbringung des Invarianten-Managers in einer Komponente, die registrieren kann, wann ein Objekt geändert wird, hat den Vorteil, daß die Feststellung einer Änderung und die Überprüfung der Invarianten in demselben Modul erfolgen kann. Hierzu ist kein Nachrichtenaustausch mit dem ConTract-Manager notwendig.

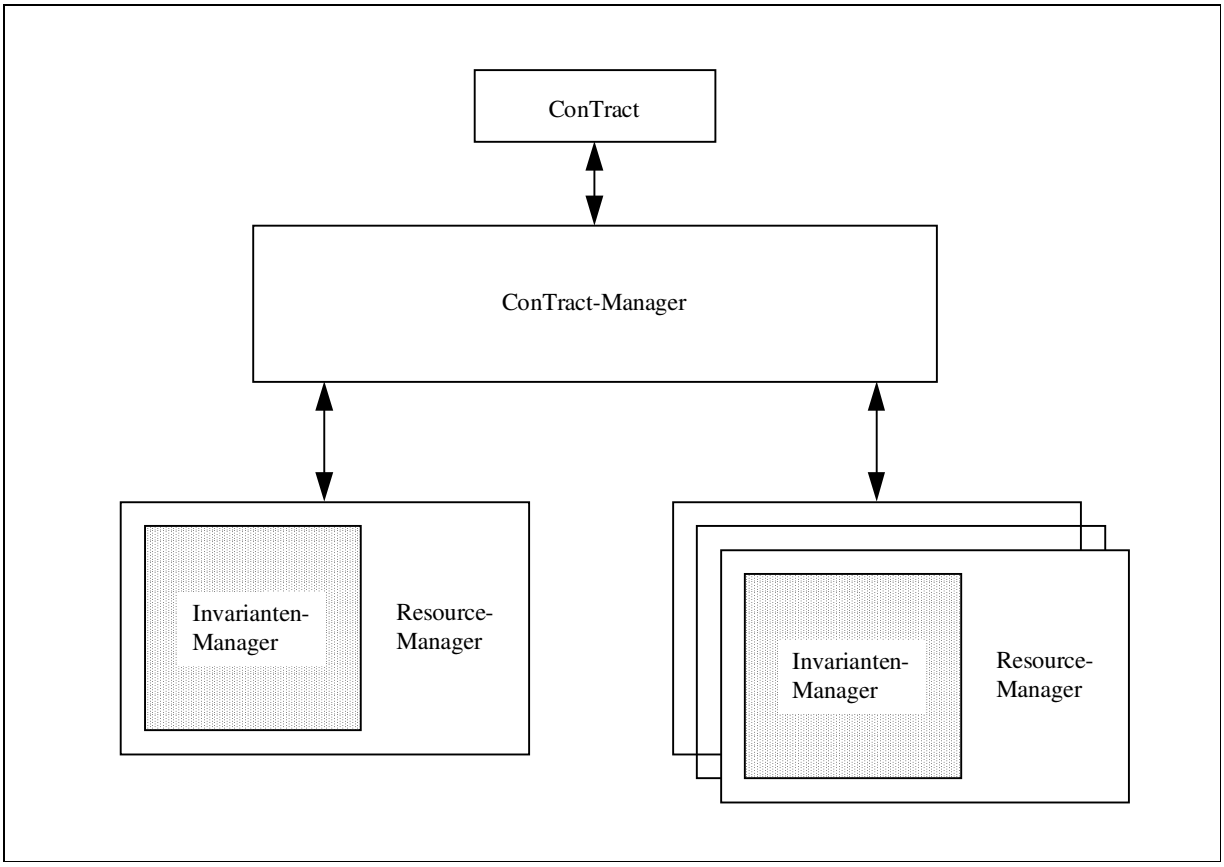

Abbildung 6: Dezentrale Verwaltungskomponente

- Die Lösung ist nicht vollständig dezentral, da der ConTract-Manager eine gewisse Vorarbeit leisten muß, um die Invarianten auf die Resource-Manager zu verteilen.
- Es können Invarianten existieren, die gleichzeitig verschiedene Objekte betreffen. Es ist möglich, daß diese durch verschiedene Resource-Manager verwaltet werden. Solche Invarianten müssen dann in sämtlichen betroffenen Resource-Managern vorhanden sein, was eventuell zu einer großen Anzahl von Duplikaten führt. Zur Auswertung einer Invariante muß dann auf andere Resource-Manager zugegriffen werden.
- Sollen sämtliche Invarianten gefunden werden, die von einem ConTract stammen, so können diese zunächst bei allen Resource-Managern vorliegen. Das gezielte Auffinden dieser Invarianten ist also mit hohem Aufwand verbunden.

7.2 Verwaltung vereinfachter Invarianten

Wie in Kapitel 6 beschrieben, ist es möglich, verschiedene Invarianten zu vereinfachen, oder sie in einer Invariante zusammenzufassen. Dies führt zu einer geringeren Anzahl von Invarianten, die bei Änderung eines Objektes überprüft werden müssen. Für die Verwaltungskomponente existieren nun mehrere Invarianten. Auf der einen Seite die Ursprünglichen und auf der anderen Seite die Zusammenfassungen

derselben. Die Frage ist, ob es für einen korrekten Ablauf ausreichend ist, nur die zusammengefaßte Version zu speichern, oder ob auch die ursprünglichen Versionen weiter vorhanden sein müssen. Dies soll in einem Beispiel betrachtet werden.

Beispiel:

- Zwei verschiedene ConTracts liefern Invarianten für das Integer-Objekt A, wobei A kein Aggregat darstellt. Von ConTract 1 kommt $A >= 3$ und von ConTract 2 kommt $A >= 5$.
- Diese beiden Invarianten können nach Kapitel 6 zu $A >= 5$ zusammengefaßt werden.
- ConTract 2 ersetzt nun die Invariante $A >= 5$ durch $A == 4$.
- Hat die Verwaltungskomponente nur die zusammengefaßte Version verwaltet, so existiert nun die Invariante $A >= 5$ und $A == 4$ ist hinzugekommen. Diese beiden sind aber unvereinbar.
- Verwaltet der Invarianten-Manager nicht nur Zusammenfassungen, sondern zusätzlich die ursprünglichen Invarianten, so kann hier zunächst die Zusammenfassung rückgängig gemacht werden. Es gelten wieder $A >= 3$ und $A >= 5$, wobei diese sofort durch $A == 4$ ersetzt wird. Nun kann erneut zusammengefaßt werden, mit dem Ergebnis $A == 4$. Dies ist das Ergebnis, das eigentlich nach der Ersetzung der Invariante durch ConTract 2 erwartet wird.

Das Beispiel zeigt deutlich, daß im Invarianten-Manager sowohl die ursprünglichen Invarianten, als auch mögliche Zusammenfassungen derselben gespeichert sein müssen. Bei Änderungen an Objekten muß allerdings nur die zusammengefaßte Version zur Überprüfung einer Invariantenverletzung herangezogen werden.

7.3 Änderung von Invarianten

Der Invarianten-Manager speichert unabhängig davon, in welchem Modul von APRICOTS er untergebracht ist, den kompletten Satz aktueller Invarianten. Deren Anzahl wechselt ständig. Hier folgt eine kurze Zusammenstellung der Ereignisse, die eine Veränderung des Satzes an Invarianten in der Verwaltungskomponente zur Folge haben. In jedem Fall sollen die möglichen Reaktionen der Verwaltungskomponente skizziert werden. Hieraus ergeben sich weitere Anforderungen an die weiter unten betrachteten Datenstrukturen zur Invariantenverwaltung.

Ende eines ConTracts

Mit dem Ende eines ConTracts werden sämtliche Invarianten des ConTracts ungültig. Sie können vom Invarianten-Manager gelöscht werden. Zusätzlich sind alle aus diesen Invarianten hervorgegangenen Zusammenfassungen ungültig. Eventuell ergeben sich mit den Invarianten anderer ConTracts neue Zusammenfassungen. Als Anforderung an die Datenstrukturen, in denen Invarianten gespeichert sind ergibt sich zweierlei. Einmal müssen alle Invarianten eines ConTracts gefunden werden können. Zum zweiten müssen zu jeder Invariante die Zusammenfassungen gefunden werden können, in die sie eingegangen ist.

Eine Invariante kommt hinzu

Dieser Fall tritt z.B. ein, wenn ein Step beendet wird und er seine Ausgangsinvariante an die Verwaltungskomponente übergibt. Handelt es sich um die erste Invariante des ConTracts zu einem bestimmten Objekt, so ist die Aufgabe des Invarianten-Managers klar. Die Invariante wird in die Liste gültiger Invarianten aufgenommen. Zusätzlich kann nach möglichen Zusammenfassungen mit anderen Invarianten gesucht werden.

Anders stellt sich die Situation dar, wenn zu einem Objekt schon eine Invariante des ConTracts existiert. Der neuen Invariante kann dann eine unterschiedliche Semantik zugeordnet werden.

- Die Invariante ist zusätzlich

  In diesem Fall wird die neue Invariante zusätzlich zu den bestehenden Invarianten desselben ConTracts verwaltet.

- Die Invariante ersetzt bestehende

  Die neue Invariante ersetzt hier alle Invarianten desselben ConTracts, die sich auf dasselbe Objekt beziehen. Eine Folge hiervon ist, daß zu jedem Objekt von jedem ConTract genau eine Invariante gültig sein kann.

Zeitlich beschränkte Invarianten

Die Besonderheit dieser Invarianten ist es, daß deren Einhaltung mit dem Zeitpunkt, zu dem sie dem Invarianten-Manager übergeben werden, nicht unbedingt sofort überprüft werden muß. Die Invariante $A >= after :13.00.00$ muß zu allen Zeitpunkten vor $:13.00.00$ nicht überprüft werden. Trotzdem muß sie verwaltet werden. Die Datenstrukturen zur Invariantenverwaltung müssen also eine Unterscheidung zwischen Invarianten, die momentan zutreffen müssen, und solchen, für die das nicht gilt, ermöglichen.

7.4 Daten für die Invariantenverwaltung

Der Invarianten-Manager kann sich nicht damit begnügen nur die Invarianten selbst zu speichern. Wie der vorangegangene Abschnitt gezeigt hat sind für jede Invariante zusätzliche Informationen notwendig. Diese sind:

- Identifikator des ConTracts

  Ein Identifikator, der jeder Invariante den ConTract, von dem sie stammt, eindeutig zuordnen läßt, ermöglicht es der Verwaltungskomponente alle Invarianten eines ConTracts schnell aufzufinden.

- Objekt

  Zu jeder Invariante soll der Identifikator für alle von der Invariante betroffenen Objekte gespeichert sein. Zur Adressierung von Objekten siehe Näheres in Abschnitt 7.6.

- Statusangabe

  Diese Angabe dient dazu, Invarianten deren Einhaltung momentan überprüft werden muß, von solchen zu unterscheiden, für die das nicht gilt. Beispielsweise müssen Invarianten, die mit anderen zusammengefaßt werden konnten zunächst nicht mehr überprüft werden. Es ist ausreichend, wenn die komprimierte Invariante bei Änderung der betroffenen Objekte geprüft wird.

- Verweis auf Zusammenfassungen

  Für jede Invariante müssen Verweise auf die Zusammenfassungen existieren, in die sie eingegangen ist. Dies erleichtert beim Löschen einer Invariante das Auffinden der daraus Hervorgegangenen.

7.5 Datenstrukturen

In diesem Abschnitt sollen nun einige für die Verwaltung von Invarianten geeignete Datenstrukturen vorgestellt werden. Hierbei wird davon ausgegangen, daß ein zentraler Invarianten-Manager vorliegt[21].

Zunächst einige Datenstrukturen, die alle Invarianten des Systems aufnehmen können. Wie eine einzelne Invariante gespeichert werden kann soll erst später untersucht werden. Als Vorstellung kann hier die Unterbringung in einem String dienen. Vorgestellt werden hier eine einfache Liste und eine zweistufige Listenstruktur. Bei jeder Datenstruktur wird der durchschnittliche Aufwand für folgende Operationen angegeben:

- Einfügen einer neuen Invariante
- Finden der Invarianten eines ConTracts
- Finden der Invarianten zu einem Objekt

Der Aufwand wird als Zahl der Zugriffe auf einzelne Elemente der Listen angegeben. Die Gesamtzahl der Listenelemente sei N. Die jeweiligen Strukturen sind im Beispiel mit den Invarianten $A > 10$ von ConTract 1, $B > 5$ von ConTract 2 und $A > 15$, sowie $B == 3$ von ConTract 3 dargestellt.

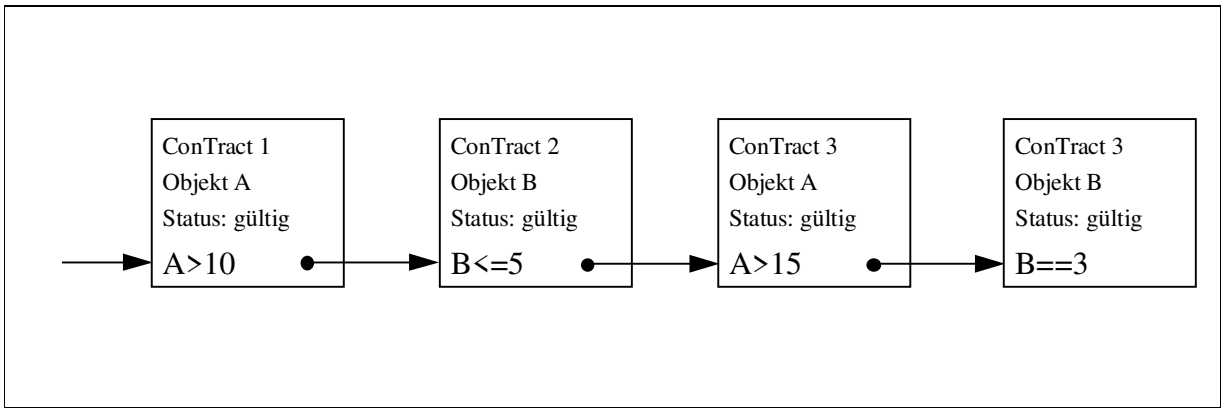

Abbildung 7: einfache Listenstruktur

Einfache Listenstruktur

Diese ist in Abbildung 7 dargestellt. Jedes Element der Liste verfügt über sämtliche Angaben zu einer bestimmten Invariante, gemäß Abschnitt 7.4. Ist die Liste unsortiert, so ist für das Einfügen einer neuen Invariante ein konstanter Aufwand notwendig. Für die Suche nach einer bestimmten Invariante sind durchschnittlich N/2 Zugriffe zu veranschlagen.

Unter der Voraussetzung, daß die Identifikatoren für ConTracts und für die Objekte geordnet werden können, ist es möglich die Liste danach zu sortieren. In der sortierten Liste benötigt das Einfügen N/2 Zugriffe auf Listenelemente. Die Suche über das Kriterium, nach dem sortiert wurde, benötigt dann wieder

[21] Vgl. Abschnitt 7.1

N/2 Zugriffe im Durchschnitt. Dies gilt hier auch für den Fall, daß das gesuchte Element gar nicht enthalten ist. Hier sind bei der unsortierten Liste N Zugriffe notwendig.

Zweistufige Listenstruktur

Diese ist in Abbildung 8 dargestellt. Hier existiert zunächst eine Liste mit Elementen für jeden ConTract bzw. für jedes Objekt. An diesen Elementen hängen dann weitere Listen, die die entsprechenden Invarianten zum ConTract bzw. Objekt enthalten. Diese Listen der zweiten Stufe weisen wesentlich weniger Elemente auf, als dies bei der einfachen Listenstruktur der Fall ist. Der Aufwand sie zu sortieren erscheint darum nicht als angemessen.

Die Liste der ersten Stufe sei mit N_1 Elementen angenommen, die der zweiten Stufe mit N_2 Elementen. Daraus folgt für das Einfügen ein Aufwand von $N_1/2+1$ Zugriffen. Die Suche nach den Invarianten zu einem ConTract benötigt $N_1/2+N_2/2$ für den Fall, daß die ConTracts in der Liste der ersten Stufe eingetragen sind. Gilt dies nicht, so muß die gesamte Struktur durchsucht werden mit einem Aufwand von N_1*N_2 Zugriffen. Entsprechendes gilt für die Suche nach Invarianten zu einem bestimmten Objekt.

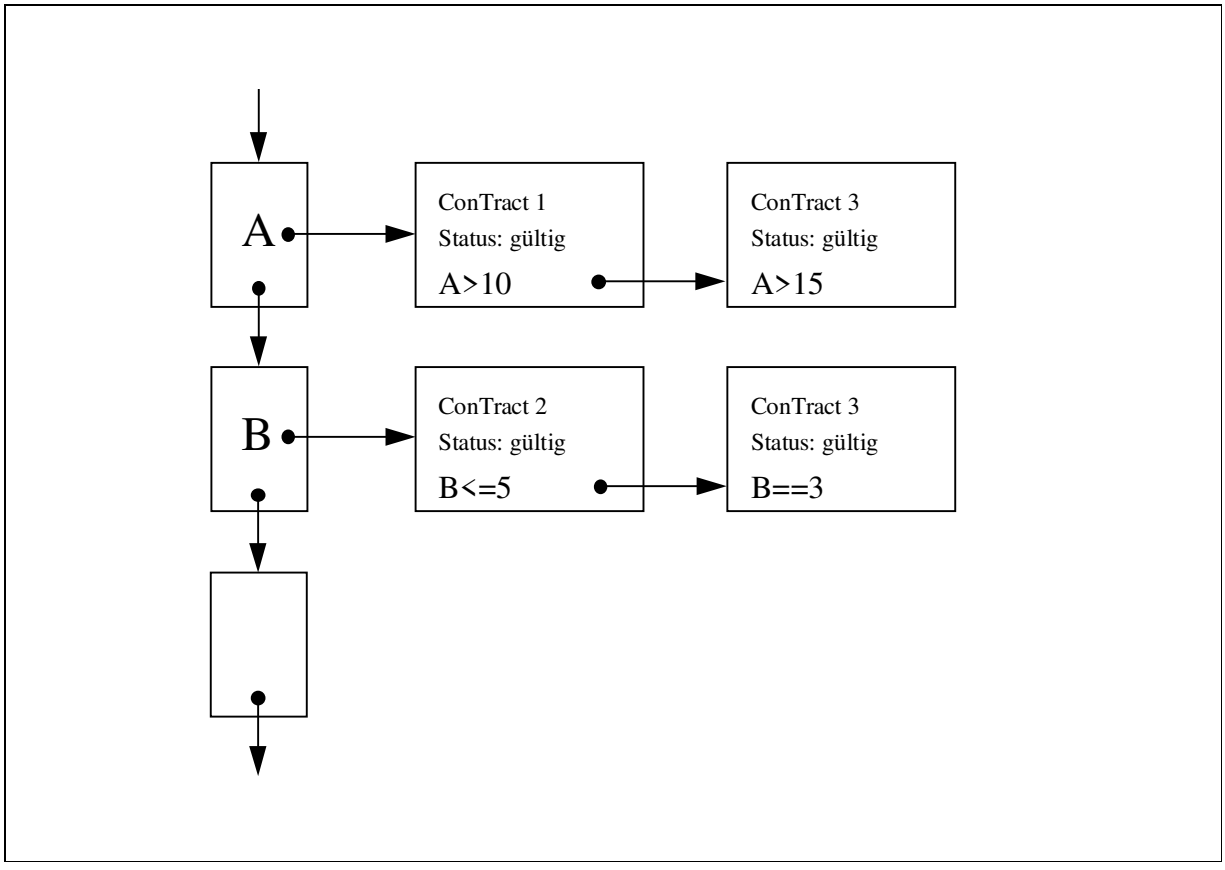

Abbildung 8: zweistufige Listenstruktur

Datenstruktur für einfache Invarianten

Die einfachen Invarianten, wie sie in einem Element der oben vorgestellten Listenstrukturen gespeichert werden sollen, können alternativ zur Darstellung als String auch in einem Operatorbaum untergebracht werden. Abbildung 9 zeigt eine entsprechende Baumstruktur für eine Invariante. In einer solchen Struktur

sind arithmetische Umformungen, wie sie eventuell bei der Zusammenfassung von Invarianten notwendig werden leichter durchzuführen, als in einem unstrukturierten String.

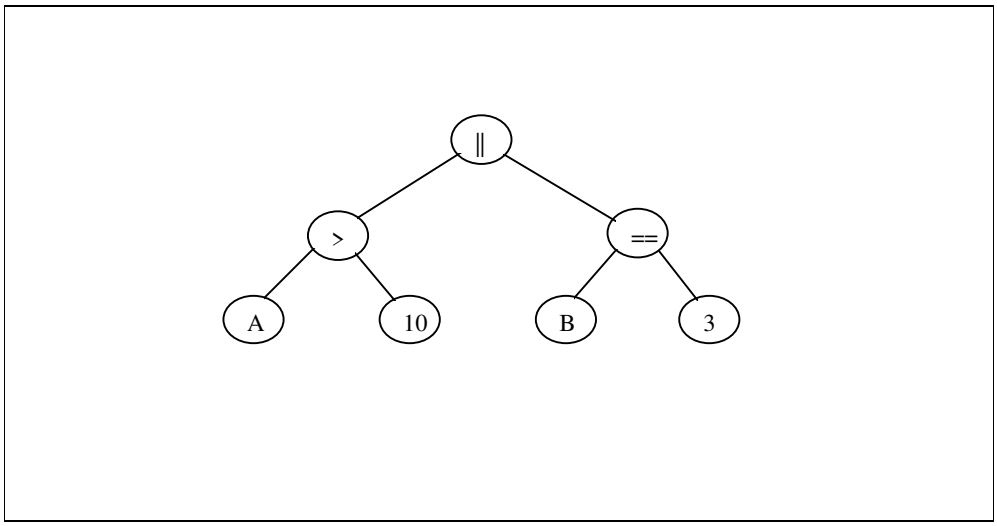

Abbildung 9: Datenstruktur für einfache Invarianten

7.6 Adressierungsverfahren

Der Invarianten-Manager ist dafür zuständig, die Invarianten zu speichern und ihre Einhaltung zu überwachen. Hierzu muß auf die von einer Invariante betroffenen Objekte zugegriffen werden. Für diese Zugriffe müssen die von Resource-Managern bereitgestellten Dienste in Anspruch genommen werden. Diese Dienste werden in der Anwendung mit anwendungsweit eindeutigen logischen Namen bezeichnet. Die Anwendung selbst ist dafür zuständig, ihre Struktur und Vergabemethode festzulegen. Es muß nun eine Abbildung von einem logischen Namen auf einen Prozeß möglich sein[22]. Der Name eines Objekts muß auf einen Prozeß abgebildet werden, der einen Zugriff auf das Objekt ermöglicht. Hierzu sind folgende Angaben notwendig[23]:

service_name : Dieser gibt den Namen des gewünschten Dienstes an.

resource_manager_class: Dies gibt die Klasse an, zu der der angesprochene Resource-Manager gehört.

component_name : Der Komponentenname gibt den Namen des Prozesses im Resource-Manager an.

service_adress : Gibt die dienstlokale Adresse für den Aufruf des Dienstes an.

Der Dienst für den Zugriff auf ein Objekt ist damit durch die vier Angaben

service_name , resource_manager_class , component_name , service_adress

eindeutig beschrieben. Der Name des referenzierten Objekts muß dann noch als Parameter bei dem Aufruf des Dienstes übergeben werden.

---

[22] Vgl. [WäRe90], S. 207-208

[23] Vgl. [Schü93], S. 41

8 Ausblick

Diese Arbeit hat sich mit den Invarianten, die zur Synchronisation einzelner Steps eines ConTracts genutzt werden beschäftigt. Die Syntax für solche Invarianten wurde mit Hilfe einer Grammatik fixiert. Hierbei wurde versucht, ein Gleichgewicht zwischen großer Flexibilität der Ausdrücke und der Einfachheit der Grammatik zu finden. Hierzu mußte die Zahl der betrachteten Datentypen und der zugelassenen Operationen eingeschränkt werden. Über die hier ausgewählten Typen und Operationen hinaus kann eine ganze Reihe anderer für verschiedene Anwendungen von Interesse sein. Überlegungen hierzu können in weiterführenden Arbeiten angestellt werden.

Ein weiterer Schwerpunkt der Arbeit lag in der Veränderung von Invarianten. Diese müssen so ausgeführt werden, daß keine inkonsistenten Zustände auftreten können. Hier ist sowohl die Abbildung von Prädikaten über zusammengesetzte Typen auf Prädikate über Basistypen, als auch die Zusammenfassung von Invarianten berücksichtigt worden. Bei diesen Zusammenfassungen konnte die aufgrund der Grammatik sehr großen Anzahl zu untersuchender Fälle nicht voll abgedeckt werden. Das Hauptaugenmerk wurde darum auf Invarianten mit einfachem Aufbau gelegt, da diese für viele Anwendungen ausreichend sind. Die Verarbeitung komplexerer Invarianten muß separat betrachtet werden.

In einem weiteren Kapitel sind verschiedene Aspekte der Verwaltung von Invarianten diskutiert worden. Hierzu gehört unter anderem die Positionierung der Verwaltungskomponente im Gesamtsystem APRICOTS, die Auflistung möglicher Datenstrukturen zur Invariantenverwaltung und das Verfahren zur Adressierung von Datenobjekten. Diese Überlegungen können als Ausgangspunkt für die Implementierung eines prototypischen Invarianten-Managers genutzt werden.

9 Literaturverzeichnis

[BuRa90] Burkert, Holger; Raichle, Bernd:
Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext. Universität Stuttgart, Fakultät Informatik, IPVR, Studienarbeit Nr. 859, 1990.

[DaWh89] Date, C.J.; White, Colin J. :
A guide to SQL/DS. Reading u.a.: Addison-Wesley, 1989.

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

[Gray81] Gray, Jim:
The Transaction Concept: Virtues and Limitations. IN: Proc. VLDB, Cannes, 1981.

[GrRe93] Gray, Jim; Reuter, Andreas:
Transaction Processing: Concepts and Techniques. San Mateo: Morgan Kaufmann, 1993.

[HaSt91] Harbison, Samuel P.; Steele, Guy L.:
C: Ein Referenzhandbuch.Wolfram´s, 1991.

[LaDu90] Laube, Dieter; Dung, Tran Van:
Entwurf und Spezifikation eines prototypischen ConTract-Managers. Universität Stuttgart, Fakultät Informatik, Studienarbeit Nr. 875, 1990.

[Laub92] Laube, Dieter:
Transaktionsorientierte Kommunikationsmechanismen für das verteilte Transaktionssystem APRICOTS. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 865, 1992.

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

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

[Rens90] Renschler, Ursula:
Entwurf und Implementierung einer experimetellen Komponente zur ConTract-Verarbeitung. Universität Stuttgart, Fakultät Informatik, Diplomarbeit Nr. 666, 1990.

[Reut89] Reuter, Andreas:
ConTracts: A Means for Extending Control beyond Transaction Boundaries. IN: Proc. 2nd Int. Workshop on High Performance Transaction Systems, Asimolar, 1989.

[RoWa93] Rothermel, Kurt; Wahl, Thomas:
Representing Time in Multimedia Systems. Universität Stuttgart, Fakultät Informatik, Fakultätsbericht 12/1993.

[Schü93] Schützle, Alexander:
Portierung / Reimplementierung eines transaktionsorientierten Kommunikationssystems. Universität Stuttgart, IPVR, Dimplomarbeit, 1993.

[Wäch91] Wächter, Helmut:
ConTracts: A Means for Improving Reliability in Distributed Computing. In: IEEE Spring CompCon, 1991.

[WäRe90] Wächter, Helmut; Reuter, Andreas: Grundkonzepte und Realisierungsstrategien des ConTract-Modells. In: Informatik Forschung und Entwicklung. (1990) Nr. 5, S. 202-212.

[WäRe91] Wächter, Helmut; Reuter, Andreas: The ConTract Model. Universität Stuttgart, Fakultät Informatik, IPVR, Computer Science Report Nr. 8/91.

[Zell92] Zellmer, Oliver:
Eine graphische Oberfläche für das verteilte Transaktionssystem APRICOTS. Universität Stuttgart, IPVR, Studienarbeit 1111, 1992.

Erklärung

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