Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Überblick

1.1 Einführung

Das klassiche Transaktionskonzept hat sich als zweckmäßiges Mittel zur Sicherung der Datenkonsistenz und der Synchronisation kurzer, parallel ablaufender Vorgänge bewährt. Das dem Konzept zugrundeliegende *ACID*–Prinzip erweist sich jedoch als ungeeignet für langandauernde, verteilte Abläufe. Dies liegt vor allem daran, daß eine Transaktion nach außen als atomar erscheint. Viele Bearbeitungsvorgänge lassen sich jedoch in der realen Welt so nicht modellieren. Die Ergebnisse einzelner Teilschritte können oftmals nicht durch Restaurieren eines *Before-Images* einfach 'zurückgesetzt' werden.

In einem ConTract wird ein langer Vorgang in einzelne Teilschritte, den Aktionen oder Steps, zerlegt. Diese besitzen die *ACID*-Eigenschaften und garantieren eine konsistente Manipulation der Objekte. Das ConTract-System übernimmt dabei die Aufgabe der Koordination und Überwachung der Ausführung der Teilschritte. Die Abfolge wird in einer Beschreibung des Kontrollflusses, dem Skript, festgelegt.

Um im Fehlerfall eine korrekte Fortführung garantieren zu können, muß der vollständige Zustand der Verarbeitung erfaßt und gesichert werden. Die einzelnen Zustände und die zugehörigen Übergänge werden durch den *Kontext* beschrieben. Mit Hilfe des Kontextes können im Fehlerfall geeignete Maßnahmen zur Fortsetzung des Vorgangs eingeleitet werden. Im Gegensatz zu Transaktionen, bei denen nach einem Fehler der Vorgang komplett zurückgesetzt wird, ist bei ConTracts eine Weiterführung durch 'logische Kompensationen' möglich. Dies beinhaltet sowohl rückwärts– wie auch vorwärtsgerichtete Kompensationsaktionen. Zur Kompensation abgeschlossener Einzelaktionen kann auf den im Kontext gesicherten früheren Ausführungszustand zurückgegriffen werden.

Durch die Kontrolle der Ausführung und die korrekte Fortführung im Fehlerfall mit Hilfe des Kontextes besitzt der ConTract die Eigenschaft der *Unverwundbarkeit* gegenüber Systemfehlern. Diese besagt, daß ein einmal begonnener Vorgang in endlicher Zeit garantiert in einen korrekten, definierten Endzustand gelangt.

1.2 Zielsetzung

Die Studienarbeit beschäftigt sich mit der Entwicklung eines Daten– und Verarbeitungsmodells für den Kontext im Rahmen des ConTract-Projekts. Die Aufgaben lassen sich wie folgt aufteilen:

1. Entwurf eines Datenmodells zur Speicherung von Kontextelementen und den zugehörigen Mechanismen zur Erzeugung der relationalen Schemata für einen ConTract.

   Von den denkbaren Kontexttypen sollen im Prototyp zunächst nur einfach strukturierte behandelt werden. Dies sind u.a. Programmvariablen einfachen Typs, z.B. *int*, *float*, *string* usw[1]. Genauer betrachtet werden soll zudem der **SQL**-Cursor zur Verarbeitung eines mengenwertigen Anfrageergebnisses. Weitergehende Kontexttypen, z.B. Dateien unter der Verwaltung des Betriebssystems oder Kommunikationsverbindungen, bleiben zunächst ausgeklammert.

2. Für die Kontextverarbeitung sind (generische) Funktionen zu erstellen, die zu Beginn und Ende eines ConTract-Verarbeitungsschrittes (Aktion, Step) Kontextelemente vom und zum Kontextspeicher transferieren. Hierzu sind entsprechende Schnittstellen zu entwerfen und zu implementieren. Zu diesem Aufgabengebiet gehört ebenfalls die Bereitstellung eines Bindemechanismus, welcher die Zuordnung einer Variablen in einer Aktion zu einem Kontextelement leistet, sowie eine Untersuchung der Möglichkeiten der Kontextreferenzierung.

3. Für die ConTract-Verarbeitung sind entsprechende Verwaltungs– und Anzeigeprozeduren zu entwickeln und zu realisieren, welche die Zusammenarbeit mit dem ConTract-Manager vervollständigen.

Die Arbeit wurde so aufgeteilt, daß der Schwerpunkt des Bearbeiters Holger Burkert auf der Entwicklung der relationalen Schemata und der Abbildung der verschiedenen Kontexttypen lag. Hinzu kam die Implementierung der notwendigen Verwaltungs– und Abfragefunktionen. Bernd Raichle war für den Entwurf und die Realisierung der Schnittstelle zwischen den Aktionen und der Kontextverwaltung, der Kontextbindung und der Referenzierung zuständig.

[1] Als Basis, auch für die Implementierung der benötigten Funktionen, dient, u.a. aus Gründen der Portierbarkeit (insbesondere zwischen den **SUN**-Workstations und der **TANDEM** des IPVR), die Programmiersprache **C**.

1.3 Zuordnung der einzelnen Kapitel

Die vorliegende Arbeit ist eine Doppelstudienarbeit. Untenstehende Tabelle gibt einen Überlick, welche Kapitel den einzelnen Autoren zuzuordnen sind.

Kapitel	Titel	Autor
1	Überblick	Holger Burkert und Bernd Raichle
2	Programmiermodell	Bernd Raichle
3	Kontext und seine Klassifikationen	Holger Burkert
4	Schemata zur Kontextverwaltung	Holger Burkert
5	Kontextreferenzierung	Bernd Raichle
6	Maßnahmen zur Erhöhung der Ausfallsicherheit	Holger Burkert
7	Vergleich mit anderen Ansätzen	Bernd Raichle
Anhang	Titel	Autor
A	Kontextschnittstelle für das Laufzeitsystem	Bernd Raichle
B	Kontextverwaltung und Abfrage im Prototypen	Holger Burkert
C	Bemerkungen zum **ANSI/ISO–SQL**–Standard	Holger Burkert

Kapitel 2

Das Programmiermodell

In den folgenden Abschnitten sollen anhand eines einfachen Beispiel-ConTracts die einzelnen Schritte zur Erstellung eines ConTracts und die dazu notwendigen Werkzeuge näher betrachtet werden. Die dabei beschriebenen Werkzeuge sind nicht Teil der Arbeit und werden deshalb nur umrißhaft skizziert.

2.1 Ein Beispiel

Als Beispiel wird die Zinsberechnung verwendet, die man auch mit dem Konzept des Mini-Batches realisieren kann. Diese Realisierung soll nun als ConTract nachgebildet werden.

Zu einem gewissen Zeitpunkt sollen bei einer Menge an Konten die Zinsen gutgeschrieben (bzw. belastet) werden. Jedes Konto ist genau einmal zu modifizieren. Der Zinssatz sowie die Kontennummern werden zu Beginn der Bearbeitung eingelesen, dabei werden die Kontonummern als Bereich mit Anfangs- und Endnummer angegeben.

Die Bearbeitung aller Konten kann man in einzelne Schritte – den sogenannten *Steps* – zerlegen, die jeweils die Änderungen an einer relativ kleinen Anzahl an Konten (in unserem Fall 1000) in einer Transaktion durchführen. Das ConTract-System garantiert die Abarbeitung aller Schritte eines ConTracts und im Fehlerfall die Fortsetzung an der abgebrochenen Stelle ohne daß man dies, im Gegensatz zum Mini-Batch, explizit angeben muß.

2.2 Die einzelnen Teile eines ConTracts

Ein ConTract besteht aus zwei Teilen:

1. einer Menge von Steps, die die einzelnen Verarbeitungsschritte durchführen und
2. dem Skript, das den Kontroll- und Datenfluß zwischen den einzelnen Steps festlegt.

Beide Teile eines ConTracts können unabhängig voneinander erstellt werden. Insbesondere kann man Steps, die bereits existieren, wiederverwenden.

Die Erstellung eines ConTracts kann durch diese Entkopplung von Verarbeitung (in den Steps) und Kontrollfluß (im Skript) in zwei Schritten erfolgen. Im ersten Schritt entwirft man das Skript. Anschließend wird die Stepbibliothek nach verwendbaren Steps durchsucht, die fehlenden Steps codiert und zur Stepbibliothek hinzugefügt. Im letzten Schritt wird das Skript vollständig ausprogrammiert und mit den vorher fehlenden Informationen über die verwendeten Steps ergänzt. Diese Vorgehensweise entspricht einer Mischung aus *bottom-up* und *top-down* Programmierung. Alternativ kann man einen ConTract streng *bottom-up* oder streng *top-down* entwerfen.

2.2.1 Das ConTract-Skript

Abbildung 2.1 zeigt das Skript eines ConTracts, das die in Abschnitt 2.1 vorgestellte Aufgabe realisiert. Die dabei verwendete Sprache ist blockstrukturiert und wurde für den Prototypen entworfen[1] [Mas90]. Für spätere ConTract-Prototypen sollen auch Netz-Notationen implementiert werden.

Jeder Block trägt eine eindeutige Nummer S_i, die sogenannte *Step-Nummer*. Da man einen Step in einem Skript mehrfach verwenden kann, kann durch diese zusätzliche Numerierung zwischen den unterschiedlichen Verwendungen desselben Steps unterschieden werden. Neben den einzelnen Steps erhalten alle Kontrollflußanweisungen eine eindeutige Step-Nummer, da Kontrollflußanweisungen meist implizit Steps ausführen, die Kontextelemente lesen und erzeugen könnten. Die Numerierung dieser Anweisungen dient damit der eindeutigen Referenzierung dieser Kontextelemente. Auf die Problematik der Referenzierung wird in Abschnitt 5.3 und folgende näher eingegangen.

Jeder Step liest und schreibt Kontextelemente, den Eingabe- und den Ausgabekontext des Steps. Da man Steps in mehreren Skripten bzw. in einem Skript an mehreren Stellen verwenden kann, kann ein Kontextelement aus diesen beiden Mengen nur durch einen symbolischen Name bezeichnet werden, der bei der Verwendung des Steps im Skript durch den tatsächlichen Kontextelementnamen ersetzt werden muß. Diese Namensbindung ist in dem in Abbildung 2.1 gezeigten Skript hinter jedem Stepnamen als Argument zu jedem

[1] Für das Beispiel wurde das Skript syntaktisch vereinfacht, daher unterscheidet es sich geringfügig von einem Skript des Prototypen.

```
ConTract MiniBatch;
context
   low_acc_no    : integer;        loop_count   : integer;
   hi_acc_no     : integer;        i            : integer;
   interest_rate : real;
begin
   S0: Init_MiniBatch (In:   –;
                       Out: first : low_acc_no, last : hi_acc_no,
                            rate : interest_rate)
   S1: Compute_LoopCount (In:   anfang : low_acc_no, ende : hi_acc_no;
                          Out: anzahl : loop_count)
   S2: Set_InitValues (In:   low : low_acc_no, high : hi_acc_no;
                       Out: low : low_acc_no, high : hi_acc_no,
                            i : i)
   S3: while (i <= loop_count)
          S4: Compute_Interests (In:   first : low_acc_no, last : hi_acc_no,
                                       rate : interest_rate;
                                 Out: –)
          S5: Set_NewValues (In:   low : low_acc_no, high : hi_acc_no,
                                   maxhigh : hi_acc_no from S0,
                                   i : i;
                             Out: low : low_acc_no, high : hi_acc_no,
                                   i : i)
       endwhile
   S6: Receipt (In:   low : low_acc_no from S0,
                      high : hi_acc_no from S0,
                      rate : interest_rate;
                Out: –)
end.
```

Abbildung 2.1: ConTract-Skript

Step angegeben. Dabei gibt man für jedes Kontextelement ein Namenspaar an, dessen erste Komponente der symbolische Name im Stepcode und dessen zweite Komponente der im Skript verwendete Name des Kontextelementes ist. Zusätzlich kann man hier weitere Referenzierungsmechanismen für ein Kontextelement angeben. Im Step S_5 wird z.B. die `from`-Klausel benutzt, mit der genau die von einem bestimmten Vorgänger (hier: Step S_0) erzeugte Versionen des Kontextelementes `hi_acc_no` addressiert werden. Dadurch erhält man nicht den letzten erzeugten Wert des Kontextelementes, den der Step S_5 (bzw. S_2 beim ersten Durchlauf der Schleife) geschrieben hat, sondern den Wert, der vom Step S_0 geschrieben wurde.

2.2.2 Die Step-Bibliothek

Die Step-Bibliothek ist eine Sammlung von Steps, die das ConTract-System verwaltet. Zu jedem Step gehört neben dem ausführbaren Code noch eine Beschreibung dieses Steps. Diese Beschreibung besteht aus einer Erläuterung des Stepcodes und aus einer Auflistung der Kontextelemente, die dieser Step als Ein- und Ausgabekontext benötigt. Ein Eintrag in der Step-Bibliothek kann die in Abbildung 2.2 gezeigten Attribute besitzen.

Stepname
- Beschreibung des Stepcodes
- ausführbarer Stepcode
- Eingabekontext:
  Variablennamen der Kontextelemente
- Ausgabekontext:
  Variablennamen der Kontextelemente
- Invarianten
- weitere Attribute, wie z.B.
  Programmierer des Steps
  Historie des Steps (Änderungen, etc.)
  Schutzrechte
  Randbedingungen an die Ausführung, wie z.B.
  Ressourcenanforderungen, ...

Abbildung 2.2: Ein Eintrag der Step-Bibliothek

Zum Durchsuchen der Step-Bibliothek kann ein Browser dienen, der neben dem Suchen des Step-Namens auch ein Suchen mit Hilfe der Step-Beschreibung erlaubt. Dazu muß diese Beschreibung oder zumindest ein Teil davon in formalisierter Form vorliegen bzw. aus der Codebeschreibung hergeleitet werden können. Eine sehr kleine und unvollständige Auswahl von Ordnungskriterien zeigt die folgende Aufstellung:

- Finden nicht nur Kontextzugriffe statt ?
  - Bei Zugriffen auf eine Datenbank:
    Finden auch Änderungsoperationen oder nur Leseoperationen statt ?
    Welche Relationen werden berührt ?
  - Bei Zugriffen auf andere Ein-/Ausgabemedien, wie Dateien oder Terminals:
    Auf welche Medien und wie (lesend/schreibend) wird zugegriffen ?
- Welche Kontextelemente werden vom Step gelesen/geschrieben ?
  - Typen der Kontextelemente
  - Operationen auf Kontextelementen
    (arithmetische, aggregierende, etc.)

Beim Neueintragen, Ändern oder Löschen eines Eintrags in der Step-Bibliothek stellen sich einige Fragen, die hier nur kurz angerissen werden sollen. Das Eintragen eines neuen

Steps bereitet keine weiteren Probleme, es muß nur ein neuer eindeutiger Name für diesen Step vergeben werden, was vom System leicht unterstützt werden kann. Ändert bzw. löscht man dagegen einen bestehenden Step-Eintrag, so sind hiervon alle ConTracts betroffen, die diesen Step benutzen. Ist z.B. zum Zeitpunkt der Änderung eines Steps ein ConTract aktiv, der diesen Step benötigt, so sollte das System u.a. Änderungen an den Ein- und Ausgabekontextmengen und das Löschen dieses Steps unterbinden.

2.2.3 Die Steps

Nachfolgend sollen die im Skript verwendeten Steps näher erläutert werden. Dabei werden zwei Steps anhand ihres Programmcodes näher betrachtet. Die restlichen Steps des Beispiel sollen nur anhand ihrer Einträge in der Step-Bibliothek vorgestellt werden.

Step Init_MiniBatch

Description:	Liest vom Terminal zwei Integer-Werte `first` und `last` und einen Real-Wert `rate` ein, mit `first` $\leq$ `last` und $0 <$ `rate` < 1.
InContext:	–
OutContext:	`first:integer` (* erste Kontonummer *) `last:integer` (* letzte Kontonummer *) `rate:real` (* Zinssatz *)

Abbildung 2.3: Der Eintrag für den Step `Init_MiniBatch`

Der erste Step im Skript aus Abbildung 2.1 liest vom Terminal drei Werte: die erste Nummer `first` und letzte Nummer `last` des Bereichs der Kontonummern, denen der Zins `rate` gutgeschrieben werden soll. Abbildung 2.3 zeigt den zugehörigen Eintrag in die Stepbibliothek[2]. Als erster Step im Skript kann dieser Step noch keine Kontextelemente referenzieren; er schreibt die ersten Elemente des Skripts in den Kontext. Dies sind die vom Terminal gelesenen Werte `first`, `last` und `rate`, die unter den im Skript verwendeten globalen Namen `low_acc_no`, `hi_acc_no` und `interest_rate` im Kontext abgelegt werden.

Step Compute_LoopCount

Description:	Errechnet aus dem Bereich, der als erster Wert `anfang` und letzter Wert `ende` angegeben wird, den Integer-Wert $anzahl = \lceil (ende - anfang + 1)/1000 \rceil$. Falls $anfang > ende$, ist $anzahl = 0$.
InContext:	`anfang:integer` (* erster Wert des Bereichs *) `ende:integer` (* letzter Wert *)
OutContext:	`anzahl:integer` (* Ergebnis *)

Abbildung 2.4: Der Eintrag für den Step `Compute_LoopCount`

Abbildung 2.4 zeigt den Step-Bibliothekseintrag des zweiten Steps `Compute_LoopCount`, der aus den im ersten Step eingegebenen Bereichsgrenzen der Kontonummern errechnet, wie

[2] Im folgenden werden nur die Attribute betrachtet, die für die Kontextreferenzierung relevant sind.

oft die nachfolgende **while**-Schleife im Skript zu durchlaufen ist. Den vereinfachten Code dieses Steps zeigt Abbildung 2.5. Wie aus der Beschreibung und aus dem Code ersichtlich, kann dieser Step weiter verallgemeinert werden: Durch Einführung eines weiteren Eingabe-Kontextelementes, das die Anzahl der abzuarbeitenden Kontonummern angibt, im Beispiel als 1000 fest vorgegeben, könnte auch die Bereichsgröße variabel gehalten werden.

```
step()                      /* Compute_LoopCount */
{  int anfang, ende, anzahl;

   get_context(&anfang, "anfang", "INTEGER");
   get_context(&ende,   "ende",   "INTEGER");

   if (anfang <= ende)
     anzahl = (ende-anfang)/1000 + 1;
   else
     anzahl = 0;

   save_context(&anzahl, "anzahl", "INTEGER");
}
```

Abbildung 2.5: Der Programmcode des Steps `Compute_LoopCount`

Im Step `Set_InitValues` werden die Schleifenvariable `i` mit 0 initialisiert und die Bereichsgrenzen `low_acc_no` und `high_acc_no` für die ersten 1000 Kontonummern bestimmt (Abbildung 2.6).

Step Set_InitValues		
Description:	low := low high := min(low + 999, high) i := 0 Der Ausgabebereich entspricht den ersten 1000 Elementen des Eingabebereichs. Besitzt der Eingabebereich weniger als 1000 Elemente, so ist der Ausgabebereich gleich dem Eingabebereich.	
InContext:	`low:integer`	`(* erster Wert des Bereichs *)`
	`high:integer`	`(* letzter Wert *)`
OutContext:	`low:integer`	`(* erster Wert des Teilbereichs *)`
	`high:integer`	`(* letzter Wert *)`
	`i:integer`	`(* initialisierter Zähler *)`

Abbildung 2.6: Der Eintrag für den Step `Set_InitValues`

Zur Auswertung der Abbruchbedingung der **while**-Schleife werden die beiden Kontextelemente `i` und `loop_count` benötigt.

Der Step `Compute_LoopCount` berechnet das Kontextelement `loop_count` einmal vor Eintritt in die Schleife, der Wert von `loop_count` bleibt innerhalb der Schleife konstant. Das Kontextelement `i` wird vor der Schleife durch den Step `Set_InitValues` initialisiert und bei jedem Schleifendurchgang durch den Step `Set_NewValues` erhöht.

Step Compute_Interests

Description:	[tb] Schreibt die Zinsen mit Zinssatz `rate` den Konten in der Relation `Konto` gut, deren Kontonummer zwischen `first` und `last` (einschließlich) liegen. Ändert Tupel in der Relation `Konto`.

InContext:	`first:integer (* erste Kontonummer *)`
	`last:integer (* letzte Kontonummer *)`
	`rate:real (* Zinssatz *)`

OutContext:	–

Abbildung 2.7: Der Eintrag für den Step `Compute_Interests`

```
step()                    /* Compute_Interests */
{  int anfang, ende;
   float zinssatz;

   get_context(&anfang,   "first", "INTEGER");
   get_context(&ende,     "last",  "INTEGER");
   get_context(&zinssatz, "rate",  "REAL");

   EXEC SQL
     UPDATE konto
      SET   balance = balance * (1 + :zinssatz)
      WHERE :anfang <= acc_no AND acc_no <= :ende;

   /* Step besitzt keinen Ausgabekontext */
}
```

Abbildung 2.8: Der Programmcode des Steps `Compute_Interests`

Innerhalb der Schleife schreibt der Step `Compute_Interests` jeweils für den aktuellen Kontonummernbereich die Zinsen gut. Die Abbildungen 2.7 und 2.8 zeigen den Eintrag und den Programmcode dieses Steps.

Für den nächsten Schleifendurchlauf erhöht der Step `Set_NewValues` die Schleifenvariable `i` um 1 und die Bereichsgrenzen der Kontonummern um 1000 (Abbildung 2.9). Da im letzten Durchlauf meist ein Bereich mit weniger als 1000 Kontonummern übrigbleibt, muß dies durch eine Abfrage, ob der Gesamtbereich überschritten wird, abgefangen werden. Dazu benötigt der Step den Wert des Kontextelementes `high_acc_no`, das der Step `Init_MiniBatch` erzeugt hat. Zum Startzeitpunkt des Steps wurde dem Kontextelement vom Step `Set_InitValues` bereits ein neuer Wert zugewiesen, somit muß zur Referenzierung des früheren Wertes zusätzlich zum Kontextelementnamen noch die Step-Nummer des Steps `Init_MiniBatch` angegeben werden.

Als letzter Step schreibt `Receipt` eine Nachricht als Quittierung der Zinsberechnung auf das Terminal (Abbildung 2.10). Diese Nachricht besteht aus dem gesamten abgearbeiteten Kontonummernbereich (`low_acc_no..high_acc_no`) und dem Zinssatz (`interest_rate`).

Step Set_NewValues

Description: $i := i + 1$
$low := low + 1000$
$high := \min(high + 1000, maxhigh)$
Inkrementiert `i`. Setzt den Ausgabebereich gleich dem um 1000 Elemente nach oben verschobenen Eingabebereich. Überschreitet dieser das angegebene oberste Element `maxhigh`, so wird der Ausgabebereich um die darüberliegenden Elemente verkleinert.

InContext:
```
low:integer          (* erster Wert des vorherigen Bereichs *)
high:integer         (* letzter Wert *)
maxhigh:integer      (* maximaler Wert von high *)
i:integer            (* Zähler *)
```

OutContext:
```
low:integer          (* erster Wert des nächsten Bereichs *)
high:integer         (* letzter Wert *)
i:integer            (* erhöhter Zähler *)
```

Abbildung 2.9: Der Eintrag für den Step `Set_NewValues`

Step Receipt

Description: Schreibt auf das Terminal zwei Integer-Werte `first`, `last` und einen Real-Wert `rate`.

InContext:
```
first:integer        (* erste Kontonummer *)
last:integer         (* letzte Kontonummer *)
rate:real            (* Zinssatz *)
```

OutContext: –

Abbildung 2.10: Der Eintrag für den Step `Receipt`

2.3 Parallelisierung der Zinsberechnung

Der Step `Compute_Interests` im Skript des vorherigen Abschnitts ändert jeweils eine Menge Tupel in der Relation `Konto`. Diese Tupelmengen, die bei jedem Schleifendurchgang bearbeitet werden, sind paarweise disjunkt, d.h. es könnte eine parallele Änderung der Tupel durch diese Steps erfolgen.

Im folgenden soll nun ein ConTract-Skript vorgestellt werden, das diese Idee realisiert. Dabei zeigt sich, daß man die meisten Steps des obigen Skripts auch für dieses Skript verwenden kann.

Abbildung 2.11 zeigt das Skript. Zur parallelen Ausführung stellt die Skriptsprache die Kontrollflußanweisung `parforeach` zur Verfügung. `parforeach` erhält als Parameter eine Menge und einen zwischen `parforeach` und `endeach` eingeschlossenen Block an Steps. Für jedes Element der Menge wird der angegebene Step-Block parallel ausgeführt. Zur Verdeutlichung der Parallelität ist in Abbildung 2.12 der zugehörige Ablaufgraph der Steps für `loop_count`= 3 dargestellt.

Im Skript ist nur ein einziger Step neu hinzugekommen: Der Step `Set_Bounds` errechnet aus dem übergebenen Kontextelement `i` den für diesen parallelen Zweig zu bearbeitenden Teilbereich `low_acc_no..high_acc_no`, der dann vom nachfolgenden Step `Compute_Interests` als Eingabekontext verwendet wird (Abbildung 2.13).

ConTract MiniBatch (∗ parallele Version ∗)

context
low_acc_no : integer; loop_count : integer;
hi_acc_no : integer; i : integer;
interest_rate : real;

begin
S_0: Init_MiniBatch (**In:** −;
Out: first : low_acc_no, last : hi_acc_no,
rate : interest_rate)
S_1: Compute_LoopCount (**In:** anfang : low_acc_no, ende : hi_acc_no,
Out: anzahl : loop_count)
S_2: **parforeach** (i **in** [1. . loop_count])
S_3: Set_Bounds (**In:** i : i,
low : low_acc_no, high : hi_acc_no;
Out: low : low_acc_no, high : hi_acc_no)
S_4: Compute_Interests (**In:** first : low_acc_no, last : hi_acc_no,
rate : interest_rate;
Out:−)
endeach
S_5: Receipt (**In:** low : low_acc_no **from** S_0,
high : hi_acc_no **from** S_0,
rate : interest_rate;
Out: −)
end.

Abbildung 2.11: ConTract-Skript (mit paralleler Abarbeitung)

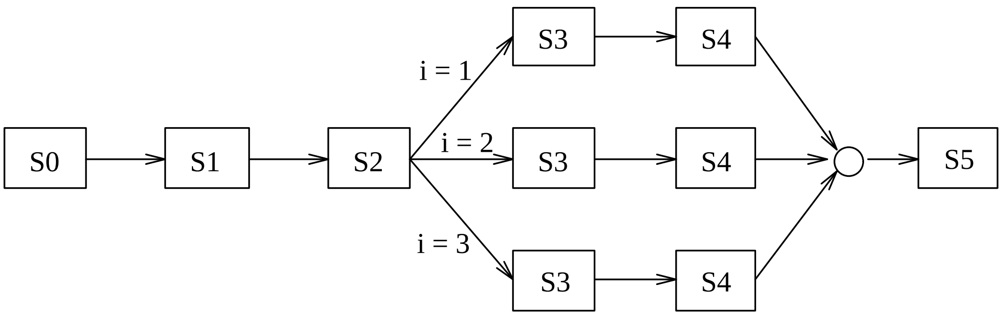

Abbildung 2.12: Ablaufgraph des Skripts (für `loop_count`= 3)

Die Referenzierung der beiden Kontextelemente `low_acc_no` und `high_acc_no` durch den Step `Compute_Interests` muß so realisiert werden, daß dieser Step nur die Werte liest, die von dem im gleichen parallelen Zweig abgelaufenen Step `Set_Bounds` errechnet wurden. Dabei ist zu beachten, daß weitere parallel dazu ablaufende Steps `Set_Bounds` neue Versionen dieser Kontextelemente schreiben, die nicht referenziert werden dürfen. Diese Problematik wird in Abschnitt 5.5.2 näher behandelt.

Step Set_Bounds

Description: $\mathrm{low} := (\mathrm{i}-1) * 1000 + \mathrm{low}$

$\mathrm{high} := \min(\mathrm{i} * 1000 - 1 + \mathrm{low}, \mathrm{high})$

Der Gesamtbereich `low..high` wird in einzelne Teilbereiche mit jeweils 1000 Elementen unterteilt. Der letzte Teilbereich enthält die restlichen Elemente. Als Ergebnis erhält man den *i.* Teilbereich.

InContext:
```
low:integer          (* erster Wert des Gesamtbereichs *)
high:integer         (* letzter Wert *)
i:integer            (* Index *)
```

OutContext:
```
low:integer          (* erster Wert des Teilbereichs *)
high:integer         (* letzter Wert *)
```

Abbildung 2.13: Der Eintrag für den Step `Set_Bounds`

Kapitel 3

Kontext und seine Klassifikationen

3.1 Der Kontext

3.1.1 Begriffsdefinition

Der Kontext dient der Beschreibung von Zuständen eines ConTracts. Genauer umschreibt der Begriff Kontext die Gesamtheit aller Ausprägungen eines bestimmten Datums, wobei das zuletzt gesicherte Kontextelement als aktuell gültig zu bezeichnen ist[1].

3.1.2 Aufgaben des Kontextes

Der Step eines ConTracts stellt nun einen Übergang von einem gültigen Zustand in einen neuen, ebenfalls gültigen Zustand dar. Kann letzterer aus noch zu erwähnenden Gründen nicht erreicht werden, kann anhand der gesicherten Kontextelemente eine Fortführung in Form einer logischen Kompensation durchgeführt werden. Konkret kann zum Beispiel bei einem Werkstück, dessen Bearbeitung durch einen Ablauffehler unterbrochen wurde, die Bearbeitung fertiggestellt werden; es findet somit kein Rücksetzen des Werkstücks (neuer Rohling) statt.

3.1.3 Aufgaben der Kontextverwaltung

Die Kontextverwaltung übernimmt — ihrem Namen entsprechend — die Aufsicht über die verschiedenen Kontexte. Sie gewährleistet die Erzeugung und gesicherte Speicherung der Kontextelemente und realisiert den Zugriff auf diese Elemente mit unterschiedlicher Funktionalität. So müssen die Elemente zum Beispiel unterschieden werden können und auf Anfrage ihre Werte bestimmt werden.

[1] Die Einschränkung der Zahl aktueller Kontextsicherungen muß nicht fix sein.

Darüber hinaus werden in der Kontextverwaltung komplexe Objekte auf Datentypen des zugrundeliegenden Datenhaltungssystems abgebildet, das im allgemeinen die Anforderungen des ACID–Konzepts erfüllt.

3.1.4 Abgrenzung zu rein datenbankorientierten Ansätzen

Allgemein liegt Datenbanken und ihren Transaktionen das ACID–Prinzip zugrunde. Dies impliziert, daß Operationen erfolgreich beendet oder zurückgesetzt werden. Bei längeren Operationen, und diese sind Gegenstand von ConTracts, führt aber gerade dies zu erheblichen Problemen und Performance–Verlusten. Offensichtlich ist in diesem Fall eine feinere Untergliederung zu finden, hier durch Kontexte realisiert. Die Konsequenz lautet, daß der Zeitpunkt einer Sicherung von Datenbeständen nicht der Datenbank autonom überlassen werden kann (traditionelles Logging), sondern explizit angestoßen werden muß. Die hier üblichen 'Checkpoints' sind jedoch nach jeder Sicherung eines Kontextelements gegeben, so daß im Prinzip auf jeden vorigen Zustand aufgesetzt werden kann, ein 'Nachfahren' mittels 'Before–/After–Images' o.ä. entfällt laut Definition ebenfalls. Eine weitere Motivation, Kontext einzuführen, ist die benutzertransparente Verwaltung komplexer, nicht von der Datenbank propagierter Datentypen. Die Kontextverwaltung kann durch Verwendung des DBMS das ACID–Prinzip bezüglich einzelner Werte nutzen, gleichzeitig jedoch diese als komplexe Objekte interpretiert zusammensetzen. Dem ConTract kann somit die reine Steuerung des Kontrollflusses überlassen werden, die Verwaltung der Daten unterliegt dem Prinzip des Information–Hiding[2].

3.1.5 Realisierung verschiedener Zustände durch Kontexte

Grundsätzlich dient der Kontext, wie bereits geschildert, der „Gewährleistung der Unverwundbarkeit durch sichere Verwaltung des gesamten Verarbeitungszustandes“. Das bedeutet im Normalfall die reine Sicherung von Daten. Dieser Kontext wird als **Aktionskontext** bezeichnet:

> *Ein Aktionskontext steht in der Verantwortung und Nutzung einzelner Steps. Er umfaßt lokale Variablen und Ein–/Ausgabeparameter ebenso wie die Position eines zur Zeit geöffneten Cursors einer mengenwertigen OBS[3]–Anfrage. Er entspricht somit im wesentlichen den von herkömmlichen Programmiersprachen zur Verfügung gestellten Methoden der Verwaltung von Daten.*

Tritt beim Ablauf einer Aktion ein Fehler auf, kommt der **Kompensationskontext** zum Tragen:

[2] Eine absolute Trennung von Daten– und Kontrollfluß ist nicht möglich oder sinnvoll. Die aktuelle Implementierung kennt denn auch gewisse Konstrukte zur Linearisierung von Mengen.

[3] Objektbanksystem

Ein Kompensationskontext charakterisiert Daten, deren Vorhandensein nicht essentiell für den normalen Ablauf ist, sondern nur im Fall eines Fehlers von Bedeutung sind. Deshalb kann dieser Kontext, im Gegensatz zum vorwärtsgerichteten Aktionskontext, als rückwärtsgerichtet bezeichnet werden. Die Notwendigkeit ist durch den jeweiligen ConTract bzw. das ConTract–System zu entscheiden.

Da die Verwaltung der Steps eines ConTracts ebenfalls verschiedene Zustände durchläuft, bietet sich eine Ausweitung des Konzepts auch auf diesen Bereich an. Der hierfür verwendete Kontext wird mit **Verarbeitungskontext** bezeichnet:

Die für die Abarbeitung von Steps eines ConTracts notwendigen Informationen, wie die derzeitige Ausführungseinheit (Rechnerknoten, CPU, etc.) oder zuständige Kompensationsaktion, sind Inhalt des Verarbeitungskontextes. Je nach dem, wo die Grenze zwischen System– und Benutzerumgebung gezogen werden kann, könnte man zum Beispiel Position und Inhalt eines Fensters auf dem Bildschirm als systeminterne Variable betrachten, die dem Benutzer als 'Umgebung' dienen.

Schließlich muß noch der Forderung nach verteilter Verarbeitung Rechnung getragen werden. Hierfür ist der **Kommunikations–/Ausführungskontext** verantwortlich:

Der Kommunikations–/Ausführungskontext zeichnet verantwortlich für die Überwachung der Kommunikationsverbindungen. Im Falle einer Störung ermöglicht er die Akquirierung und Verwendung von Ersatzverbindungen. Ebenso kann die Lastbalancierung mit seinen Informationen gespeist und durch Komponentenüberwachung ein fehlertolerantes System realisiert werden.

Im weiteren wird vorrangig der Aktionskontext behandelt, dem Stand der Prototyp–Entwicklung entsprechend.

Im folgenden sollen nun Grenzen und Möglichkeiten der Kontextverwaltung aufgezeigt werden, wobei der Aktionskontext, also der Kontext, der von den einzelnen Aktionen benötigt und erzeugt wird, im Mittelpunkt stehen soll. Aktionen eines Kontextes sind Einheiten, die auf üblichen Rechnersystemen zum Ablauf kommen. Offensichtlich ist also der Kontext ein Konstrukt, das in seiner endgültigen Realisierung Speicher repräsentiert. Er entspricht im wesentlichen den Datentypen einer Programmiersprache, insbesondere denen der 3. Generation. So sei auch die Prämisse zu setzen erlaubt, daß das ConTract–System vornehmlich in einer solcher Umgebung zum Einsatz kommen soll, so zum Beispiel im Bereich der Fertigungstechnik, Prozeßdatenverarbeitung oder Office–Service–Systeme.

Aktionen sind Programme bzw. Programmsysteme, die in eben jenen Programmiersprachen, wie **COBOL**, **FORTRAN**, **C** oder **PASCAL**, erstellt wurden. Die Abbildung der Datenkonstrukte dieser Programmiersprachen in die Darstellung der untergeordneten Instanz der Datenhaltung — konkret, der relationalen Datenbank — steht denn auch im Vordergrund der Ausführungen.

3.2 Klassifikation von Kontextobjekten

Die hier aufgeführte Klassifikation ist ein erster Ansatz, die große Anzahl verschiedener Datentypen denkbarer Kontextobjekte zu erfassen. Sie beginnt mit der Beschreibung einfacher Datentypen und reicht bis hin zu zusammengesetzten oder mengenartigen Konstrukten. Die jeweils gewählten Beispiele sind auch als solche zu betrachten nicht etwa als die einzig richtige Lösung.

Zunächst seien die verschiedenen Kontextobjekte grob anhand ihrer Komplexität unterschieden:

- **Einfache Kontextobjekte**.
  Diese Kontextobjekte zeichnen sich dadurch aus, daß der Programmierer für ihre Realisation keinen Aufwand betreiben muß. Sie sind in der jeweiligen Programmiersprache durch lediglich eine Deklaration zu definieren. Zu nennen sind hierfür z.B. `real` oder `integer` aber auch Felder oder SQL–Cursor.

- **Schwach strukturierte Kontextobjekte**.
  Die diesen Objekten zugrundeliegenden Datentypen hingegen verändern zur Laufzeit explizit Form und Größe. Vorzustellen hat man sich z.B. Listen und Bäume, für deren Richtigkeit und Generierung der Programmierer mit eigenem Programm–Code sorgt. Dies impliziert offensichtlich eine gewisse Kommunikation zwischen System und Benutzer über Art und Weise, wie bestimmte Daten zu interpretieren sind, z.B. bei einer Verkettung mittels Zeigern.

- **Komplexe Kontextobjekte**.
  Der Grad der Komplexität ist hier am höchsten. Die Daten in ihrer Form und Ausprägung können im vorneherein nicht bekannt sein. Dem Datentyp liegt auch keine konkrete Beschreibung zugrunde, vielmehr eine Art von Regelsystem über die Wohlgeformtheit seines Aufbaus. Man kann sich hier vorstellen, daß die Arbeit an einem Programmsystem — vielmehr den jeweiligen Quellprogrammen — in eine Umgebung gefaßt werden soll. In diesem Beispiel hat ein Kontext den Code eines Programmes vor und nach einer Veränderung aufzunehmen und zu verwalten. Gleiches gilt für den Entwurf von Zeichnungen, CAD–Entwürfen oder das Gebiet der Wissensverarbeitung.

Offensichtliches Merkmal der Komplexität ist Grad und Umfang der jeweils auf die Daten angewendeten Interpretation. Eine Zeichnung als Feld von Punkten darzustellen, drückt etwas anderes aus, als die Beschreibung der Linien, Kreise etc.. Während im zweiten Fall bei geeigneter Erfassung die Bilder nacheinander in den vorherigen Zustand rückversetzt werden können, ist solch ein Vorgehen im ersten Fall nicht mehr möglich. Je nach Bedarf ist auf Unterschiede dieser Art bei der Modellierung der Anwendung Rücksicht zu nehmen.

3.2.1 Einfache Kontextobjekte

Zahlen und einzelne Zeichen

Ganze Zahlen, Gleitkomma- und Festpunktzahlen sowie einzelne Zeichen (CHARACTER) als Kontextelemente stellen für die Abbildung auf ein relationales Schema keine Schwierigkeit dar. Sie können direkt auf Attribute abgebildet werden, da laut **ANSI/ISO–SQL**–Standard u.a. die Attributtypen `INTEGER`, `DOUBLE`, `DECIMAL` und `CHARACTER` zur Verfügung stehen[4]. Geachtet werden muß nur auf die unterschiedlichen Wertebereiche der einzelnen Attribute, sei es im Vergleich zwischen Programm und Datenbank oder zwischen verschiedenen Datenbanksystemen.

Dies ist insbesondere dann erforderlich, wenn es darum geht, einen ConTract in einer heterogenen Rechnerumgebung zu migrieren oder den Vorgang prinzipiell auf unterschiedlicher Hardware auszuführen. Am **IPVR** wurde der Prototyp zur ConTract–Verwaltung so ausgelegt, daß Aktionen von **SUN**–Workstations zur **TANDEM** und umgekehrt migriert werden können. Dies machte eine Beschränkung auf die Schnittmenge der DBMS–Wertebereiche von Gleitkommazahlen erforderlich.

Viele Datenbanksysteme stellen dem Benutzer weitere Attributtypen wie `TIME`, `DATE`, `TIMESPAN` und `BOOLEAN` zur Verfügung. Ihnen liegt i.allg. aber nur eine andere, einfache Interpretation der Bits zugrunde, die keine Neuklassifikation notwendig macht[5].

Eindimensionale Felder

Zeichenketten

Für den Bereich der eindimensionalen Felder existiert ein Sonderfall, bei dem Felder von Zeichen in der Grundmenge der Attributtypen vorhanden sind und somit direkt abgebildet werden können.

Felder über Non–Character Attribute

Non–Character Attribute sind im **ANSI/ISO**–Standard ganze Zahlen, Gleitkomma- und Festpunktzahlen. Darüber hinaus gehende Typen, wie `TIME` u.ä. , können in gleicher Weise behandelt werden, sie gehören jedoch nicht dem **ANSI/ISO**–Standard an.

a.) **Statische Felddimension:**
Wenn die Dimension des Feldes bekannt ist, besteht die Möglichkeit, die einzelnen

[4] Eine vollständige Liste der ANSI–Datentypen befindet sich im Anhang C, S. 83.

[5] Soweit erforderlich, werden Funktionen für den Umgang mit diesen Attributtypen in Programmiersprachen von Seiten der DBMS–Hersteller angeboten.

Werte jeweils als ein Attribut eines Tupels abzuspeichern. Dies bedeutet eine Aufteilung der Feldwerte auf Attribute durch die Kontextverwaltung.

Beispiel: Ein Feld reeller Zahlen sei in PASCAL wie folgt deklariert:

```
VAR Feld : ARRAY [1..5] OF REAL;
```

Die Relation, in der das Feld gespeichert wird, hat den Aufbau:

```
R_ARRAY1(ConTract_Id, Feldname, Wert1, Wert2, Wert3 ,Wert4, Wert5)
```

Die Funktion zum Sichern des Kontextelementes 'Feld' benützt das folgende **INSERT**–Statement:

```
INSERT INTO R_ARRAY
    VALUES (:Feld[1], :Feld[2], :Feld[3], :Feld[4], :Feld[5] );
```

Vorteile:

- Das gesamte Feld wird als ein Tupel abgespeichert. D.h., es ist zum Lesen bzw. Schreiben des Feldes nur ein Funktionsaufruf und maximal ein Plattenzugriff erforderlich (Kosten für Disk–I/O, die beim Zugriff über den Index enstehen, werden nicht betrachtet).

Nachteile:

- Die Anzahl der Attribute, also die Dimension des Feldes, ist beschränkt. Maßgebend für die Anzahl ist die Größe einer Seite bzw. eine eventuelle Beschränkung der Attributanzahl von Seiten des DBMS. Bei den für den Prototypen verwendeten Datenbanksystemen **NONSTOP–SQL** und **TRANSBASE** wird die Attributanzahl durch die Seitengröße eingeschränkt, bei **ORACLE** hingegen darf die Zahl der Attribute nicht größer als 254 sein.
- Es muß entweder für jede verwendete Feldgröße eine Relation erzeugt werden oder es werden bei kleinen Feldern die zuviel erzeugten Attribute ungenutzt gelassen und somit Speicherkapazität verschwendet.
- Der Programmcode zum Lesen und Schreiben eines Kontextelementes ist nicht statisch, sondern vom Aufbau der Relation und somit von der Felddimension abhängig. Dies impliziert auch, daß die Feldgröße zur Laufzeit nicht verändert werden kann.

b.) **Dynamische Felddimension:**
Die Abspeicherung von Feldern, deren Dimension von vorneherein nicht bekannt ist oder deren Felder nicht immer genutzt werden („dünn besetzte Matrizen"), kann dadurch erfolgen, daß zur Identifikation des Feldwertes Feldname und Feldindex gespeichert werden. Diese drei Werte zusammen sind ein Tupel innerhalb der Relation.

Für obiges Beispiel:

```
R_ARRAY2(ConTract_Id, Feldname, Feldindex, Wert)
```

Die Anweisungen zum Sichern eines Kontextelementes vom Typ 'ARRAY' kann nun allerdings nicht mehr mit nur einem einzigen **SQL**–Statement erfolgen, es ist i.allg. eine Schleife erforderlich. Dies hat den Nachteil, daß sehr viele DB–Funktionsaufrufe vonnöten sind[6]. Andererseits muß nicht mehr für jede Felddimension eine eigene Prozedur zur Kontextreferenzierung erstellt werden, es genügt die Dimension der Prozedur als Parameter zu übergeben.

Folgendes Fragment stellt eine Schleife zum Einfügen eines Feldes in eine Kontextrelation dar:

```
FOR i := lower_bound TO upper_bound
BEGIN
   INSERT INTO R_ARRAY2 (ConTract_Id, Feldname, Feldindex, Wert)
          VALUES (:cid, :arrayname, :arrayindex: :Feld[i] );
END
```

Vorteile:

- Das Feld kann eine beliebige (insbesondere auch dynamische) Dimension haben. Es genügt, für jeden Datentyp eine Relation zu erzeugen anstatt für jeden Feldtyp.
- Die Funktionen der Kontextverwaltung zur Umwandlung zwischen der Sicht des Programmierers und des logischen Schemas der Datenbank sind nicht von der Felddimension anhängig. Sie können mit Schleifen und einfachen **SQL**–Statements innerhalb nur einer Funktion programmiert werden.
- Dünnbesetzte Felder können in effizienter Weise gespeichert werden, indem Feldelemente ohne Wert nicht in die Datenbank übernommen werden. Beim Lesen solcher Felder werden physisch nicht vorhandene Feldelemente durch die Kontextverwaltung mit einem ausgezeichneten Wert initialisiert.

```
EXEC SQL DECLARE C1 CURSOR FOR
         SELECT Feldindex, Wert
         FROM   R_ARRAY2
         WHERE  ConTract_Id = :cid
         AND    Feldname    = :arrayname
         ORDER BY Feldindex;

EXEC SQL OPEN C1;

Ende := FALSE;
last_arrayindex := lower_bound;

EXEC SQL WHENEVER NOT FOUND Ende := TRUE;
```

[6] Eine Verminderung der Zugriffskosten beim Lesen könnte hier eine Mengenschnittstelle bringen. Dies ist in den heutigen Datenbanksystemen und im **ANSI/ISO–SQL**–Standard nicht vorgesehen. Manche DBMS, z.B. **ORACLE** erlauben es aber, beim Lesen mit einem CURSOR (in einem Programm mit *Embedded SQL*) statt einzelner Variablen ganze Felder zu verwenden. Dies bedeutet, daß mit einem DB–Aufruf, und somit ohne fortlaufende Prozeßwechsel, z.B. 20 Tupel verarbeitet werden können.

```
WHILE (NOT Ende) DO
 BEGIN
   EXEC SQL FETCH C1 INTO :index, :value;

   IF (index >= upper_bound) THEN
       Ende := TRUE
   ELSE
    BEGIN
       FOR i := last_arrayindex TO index-1
           Feld[i] := Default_Value;

       Feld[index] := value;
    END {else}
 END {while}

EXEC SQL CLOSE C1;
```

- Bei einfachen, direkt auf Attributtypen abbildbaren Feldtypen können die **SQL**–Aggregatfunktionen **MIN**, **MAX**, **AVG** und **SUM** verwendet werden. Des weiteren kann mit der **ORDER BY**–Klausel eine Sortierung der Feldelemente erreicht werden. Solche Feldoperationen können funktional dem ConTract–Programmierer zur Verfügung gestellt werden.

Nachteil:

- Als Nachteil gegenüber den 'Statischen Feldern' ist die signifikant größere Anzahl der Zugriffe auf die Datenbank zu nennen. Dies führt zu vermehrten Prozeßwechselkosten und Plattenzugriffen, setzt man die heute üblichen Architekturen der verteilten Verarbeitung voraus[7].

Bemerkungen:

- Komplexe Feldelemente, die beispielsweise bei 'ARRAY OF RECORD' gegeben sind, müssen auf verschiedene, jeweils für einen Datentyp vorgesehene Relationen aufgeteilt werden.
- Allgemein sind die Kosten für die Bearbeitung eines Kontextobjekts im Vergleich zu statischen Feldern höher jedoch ist die Speicherung effizienter und mächtiger. Somit sollten statische Methoden nur eingesetzt werden, wenn sämtliche Feldgrößen vorab bekannt und konstant sind.

[7] Im allgemeinen existieren designierte Datenbank–Prozesse. Entweder müssen diese aktiviert ($\rightsquigarrow$Prozeßwechsel) oder bei Mehrprozessor–Systemen mittels Nachrichten beauftragt werden.

Strukturen

Analog zur Verwaltung von Feldern, bietet sich auch bei Strukturen (Verbunden) die Möglichkeit an, die Daten in einem Tupel einer Relation zu speichern, oder aber eine Aufteilung auf Tupel in unterschiedlichen Relationen vorzunehmen. Die Wahl hängt wiederum von der Komplexität der Kontextobjekte ab.

a.) **Strukturen aus ordinalen Datentypen:**
Im wesentlichen kann dieses Problem auf die Speicherung einzelner Werte zurückgeführt werden. Sollen variante Strukturen verwaltet werden, führt die Speicherung in mehreren Relationen ebenso zum Ziel wie die Speicherung in einer Relation, in der für jeden möglichen Datentyp Attribute in ausreichender Zahl vorhanden sind. Verschachtelungen werden linearisiert, sprich: sämtliche Attribute auf eine Ebene (oder Struktur) abgebildet, Namenskonflikte müssen hierbei eventuell aufgelöst werden.

Vorteil:

- Wird die Struktur als ein Tupel abgespeichert, ist zum Lesen bzw. Schreiben des Datensatzes nur ein Datenbankzugriff erforderlich. Wird sie über mehrere Relationen verteilt gespeichert, 'verschenkt' man keinen Speicherplatz, außerdem ist das Konzept für laufzeitdynamische Strukturen geeignet.

Nachteile:

- Die Anzahl der Attribute ist physisch beschränkt.
- Des weiteren muß für jede verwendete Struktur bzw. für jeden verwendeten Attributtyp eine eigene Relation und die nötigen Zugriffsroutinen erzeugt werden.

Bemerkungen:

- Bei der Auflösung verschachtelter Strukturen können Namenskonflikte auftreten, die jedoch durch Einbeziehung der Verschachtelungstiefe in den Attributnamen vermieden werden können. Selbiges gilt für Namenskollisionen in varianten Strukturen.
- Die generelle Aufteilung der Attribute auf verschiedene Relationen und deren Gestaltung kann durch Verwendung einer zusätzlichen Compilierphase (Precompiler) bewältigt werden. Ebenso ist die Generierung der Aufrufe vorab durch den Precompiler möglich und spart gegenüber generellen Funktionen Ausführungszeit. In ersten Fall werden direkt DB–Operationen in die Wirtssprache eingesetzt, im anderen Fall werden Funktionen parametrisiert aufgerufen, die die Interpretation vollständig zur Laufzeit übernehmen.

  Prinzipiell muß solch eine generelle Funktion in der Lage sein, beliebig beschriebene Strukturen selbst zu verarbeiten oder eine beliebige Zahl verschiedenster Typen von Attributen mittels SQL erfragen zu können. Hier ist ein Ansatz interessant, der es erlaubt, mittels Deskriptoren dynamische SQL–Anfragen bezüglich Typ und Zahl von Attributen zu generieren. Für **INFORMIX**, **TRANSBASE** und **ORACLE** ist dies bekannt, ob und in welchem Ausmaß das Verfahren der `CALL`–Schnittstelle für andere Datenbanksysteme Gültigkeit hat, ist fraglich.

b.) **Beliebige Strukturen:**
Strukturen, die Datenelemente komplexen Typs aufweisen, können offensichtlich nicht immer in einem Tupel einer Relation dargestellt werden. Daraus ergibt sich, daß solche Strukturen auf diverse Tabellen verteilt werden *müssen*. Hierbei erscheint es sinnvoll, für jede auftretende Strukturkomponente eine eigene Relation anzulegen, wobei eventuell Relationen verschiedener Strukturen zusammengefaßt werden können, soweit es der jeweilige Datentyp erlaubt. Dieses Verfahren stellt eine Linearisierung im weiteren Sinne dar, so daß nun auch Mengen o.ä. als Komponenten einer Struktur auftreten können.

Aufteilung und Zusammensetzung der einzelnen Strukturkomponenten erfolgt anhand der oben beschriebenen Methode der Nutzung eines Deskriptors bzw. eines Precompilers.

Vorteile:

- Die Darstellung ist von allgemeinerer Form als der obige Ansatz.
- Die Verwendung eines Precompilers wird vereinfacht. Mußte im vorherigen Fall die Möglichkeit bestehen, laufzeit–dynamische SQL–Statements bezüglich Zahl und Typ von Attributen zu generieren, können die Zugriffe auf die verteilte Darstellung vorab festgelegt und eingesetzt werden. Berücksichtigt werden muß auch, daß die von den Datenbank–Herstellern verwendeten Verfahren dynamischer Anfragen sich unterscheiden, da eine Normierung fehlt.
- Bei geeigneter Datenmodellierung und entsprechender Größe der Anwendung kann die Zahl der nötigen Relationen klein gehalten werden. Außerdem stellt diese Darstellung laut Definition eine vertikale Zerlegung eines Objekttyps dar, was entscheidende Vorteile im Bereich verteilter Speicherung bringt.

Nachteile:

- Die Zahl der Zugriffe für das Lesen/Schreiben einer Struktur ist mindestens gleich der Zahl der Komponenten.
- Für die eindeutige Referenzierung einzelner Komponenten müssen über den Namen der Struktur hinaus weitere Informationen abgelegt werden.
- Die Verwaltung von Zugriffsrechten gewinnt an Komplexität, sollten Komponenten verschiedener Strukturen zusammengefaßt werden. Zu jedem Eintrag einer Tabelle ist eine Benutzerkennung mit abzulegen, um Zugriffsrechte verifizieren zu können, da in SQL nur die Differenzierung von Rechten bezüglich Spalten, nicht aber von Zeilen, möglich ist. Die Tatsache aufwendigerer Sperrvorgänge kann vernachlässigt werden, da eine Modifikation von Kontextelementen nicht stattfindet.

Mehrdimensionale Felder

Grundsätzlich ist die Dimension eines Feldes fortgesetzt um 1 zu erniedrigen, wenn man die jeweilige Dimension in den Datenbereich abbildet. Konkret geschieht dies, indem die

Dimensionswerte in Attributen der Tabelle verwaltet werden. Somit sind letztendlich Felder beliebiger Dimension auf bisher behandelte Objekttypen zurückführbar. Dies entspricht offensichtlich der zusammengefaßten Definition eines Feldes in PASCAL, im Beispiel

```
TYPE Feld2 = ARRAY [1..2][1..4] OF
    RECORD
        reell  :  REAL;
        imag   :  REAL;
    END;
```

Bei der Realisierung mittels einer relationalen Datenbank treten jedoch folgende Probleme auf:

- Die Referenzierung der 'Dimensionsattribute' ist statisch. Das bedeutet, daß in SQL die Namen der selektierten Attribute nicht indiziert angegeben werden können und somit eine Umsetzung der Dimension in das zugehörige Attribut innerhalb der Kontextverwaltung stattfinden muß.

  Ein anderer Ansatz besteht darin, sämtliche Dimensionsinformationen in einem einzigen Attribut zu kodieren. Dabei muß aber auf die physische Grenze des Attributs Rücksicht genommen werden, was die Anzahl der zur Verfügung stehenden Dimensionen eingrenzt. Die Kodierung hat zudem Einfluß auf die Verwendbarkeit der Aggregatsfunktionen (siehe 2, S. 22) und muß von der Kontextverwaltung durchgeführt werden.

- Ist der dem Feld zugrundeliegende Datentyp von derart komplexer Struktur, daß er verteilt gespeichert werden muß, so bietet es sich an, nicht den Datenwert selbst, sondern seinen Primärschlüssel als Fremdschlüssel innerhalb der Felddarstellung abzulegen. Somit ist die Adressierung der Feldwerte weitgehend von der Verwaltung der Objekttypen entkoppelt.

- Für Felder von jeweils unterschiedlicher Dimension sind eventuell auch verschiedene Relationen zu erzeugen, die die nötige Anzahl von 'Dimensionsattributen' aufnehmen können. Es kommt zu Rückwirkungen auf die Zahl der Zugriffsfunktionen etc. , wie bereits erwähnt.

Bemerkungen:

- Kleine Felder können auch als ein Tupel einer Relation gespeichert werden, wobei die Interpretation durch die Kontextverwaltung vorgenommen wird. Eine statische Größe des Feldes ist Voraussetzung.

- Ebenso können zu Optimierungszwecken die Werte verschiedener Dimensionen innerhalb einer Zeile einer Tabelle abgelegt werden. Anzumerken ist jedoch, daß auch dies eine spezielle Interpretation durch die Kontextverwaltung voraussetzt.

Mengen

Mengen zu speichern ist die grundlegende Idee des relationalen Datenmodells. Der allgemeine Ansatz für die Speicherung von Mengen ist die Aufteilung der Mengenelemente auf Zeilen der jeweiligen Relation (unter Angabe einer Mengenidentifikation). So kann die Sicherung von Mengen und die typischen Mengenoperationen, wie z.B. Vereinigung oder Schnitt, direkt auf die *Data Manipulation Language DML* (in diesem Fall SQL) abgebildet werden.

Aufwendigere Mechanismen sind nötig, wenn die Mengenelementen nicht aus ordinalen Datentypen bestehen. Für diese Fälle bietet es sich an, eine gewisse Indirektion in Kauf zu nehmen, insofern als die Menge durch Verweise auf die eigentlichen Mengenelemente in Form von Fremdschlüsseln dargestellt wird. Somit ist die Möglichkeit geschaffen, Mengen beliebig komplexer Elemente zu verwalten. Anforderungen bzgl. Konsistenz müssen jedoch — entsprechend verteilt gespeicherten Konstrukten — den veränderten Rahmenbedingungen angepaßt werden.

Erweiterung einer Programmiersprache um das Mengenkonstrukt

Stellt die jeweilige Wirtssprache den Datentyp Menge nicht zur Verfügung, sollen jedoch trotzdem Mengen gesichert werden können, kann die komplette Verwaltung von Mengen durch die Kontextverwaltung erörtert werden. Wie schon erwähnt, könnten neben dem Datentyp Menge auch Mengenoperationen für die Wirtssprache bereitgestellt werden. Zu beachten bleibt jedoch, daß das Konzept allgemein gehalten werden muß, um nicht eine Diskrepanz gegenüber Sprachen, die Mengenkonstrukte besitzen, zu erzeugen.

Sicherung sprachimmanenter Mengenkonstrukte

Hierbei entfällt die Abbildung von Mengenoperationen auf die DML, so daß die reine Speicherung von Mengen (eventuell 'indirekt') übrig bleibt. Sind jedoch gewisse Prämissen bezüglich der zu speichernden Menge bekannt, kann obiger allgemeiner Ansatz in bezug auf performanten Zugriff optimiert werden. Prämissen können die maximale Anzahl der Mengenelemente oder die Festlegung der Definitionswerte der Elemente sein.

a.) **Definitionsmenge bekannt:** Ist die Definitionsmenge der Elemente endlich und vorab bekannt[8], ist gleichfalls die maximale Anzahl der Elemente in der Menge bekannt. So kann eine Menge als genau ein Tupel einer Relation gespeichert werden. Das jeweilige Attribut besitzt als Namen den möglichen Elementwert und als Inhalt die Kennzeichnung, ob der betreffende Wert in der Menge enthalten ist.

[8] Im Falle der Programmiersprache PASCAL wird eine Menge durch eine endliche Aufzählung spezifiziert.

Beispiel:

Menge	a	b	c	d	e	f
$\emptyset$						
A	×			×		×
B		×	×	×	×	×

bezeichnet die Mengen

$$\begin{aligned} \emptyset &= \{\} \\ A &= \{a, d, f\} \\ B &= \{b, c, d, e, f\} \end{aligned}$$

b.) **Zahl der Mengenelemente bekannt:** Hier kann die Zahl der notwendigen Attribute einer Relation festgelegt sein, die dann die Werte der Elemente aufnehmen. Wird als Attributtyp eine Zeichenkette verwendet, können in begrenztem Maße auch Elemente verschiedenen Typs in einer Menge abgelegt werden.

Beispiel:

Sei folgende Relationdefinition gegeben

```
CREATE TABLE mengen
 (
  ...
  mengen_name CHAR(20),
  elem1       CHAR(10),
  elem2       CHAR(10),
  elem3       CHAR(10),
  elem4       CHAR(10),
  elem5       CHAR(10)
 );
```

Dann charakterisiert das Statement

```
INSERT INTO mengen
    VALUES (..., "MENGE_A", "5", "Hello", "773", "0.4", NULL);
```

die Menge MENGE_A $= \{5, \text{Hello}, 773, 0.4\}$.

Bemerkungen:

- Offensichtlich muß bei der zweiten Variante Rücksicht auf die Ordnung innerhalb der Relation genommen werden. Die jeweils 'freien' Attribute müssen markiert und aufgesucht werden.
- Die Optimierungsansätze sind nur dann zu vertreten, wenn innerhalb einer Anwendung mehrere Mengen gleicher Ausprägung bezüglich Definitionsmenge bzw. Zahl der Elemente auftreten, um diese innerhalb einer gemeinsamen Relation speichern zu können. Auch sind die Referenzierungsfunktionen den jeweiligen Mengen und dem damit verbundenen Relationenaufbau anzupassen, so daß im Normalfall die allgemeine Darstellung zu bevorzugen ist.

3.2.2 Das Kontextobjekt *Cursor*

Im Zuge der fortschreitenden Verwendung von Applikationen mit Zugriff auf Datenbanksysteme findet das SQL–Cursor–Konzept immer weitere Verbreitung. Sie dienen generell der Sequentialisierung des Zugriffs auf eine Menge von Werten, wobei jedoch nicht jeder einzelne Wert explizit benannt, sondern die Menge der Werte mittels Gemeinsamkeiten spezifiziert wird. Die Applikation geht also davon aus, daß sie nicht auf bestimmte Daten zugreifen wird, vor und nach dem Zugriff auch nicht bekannt ist, welche Tupel betroffen waren.

Zunächst ist die Frage zu klären, welche Interpretation dem Begriff des *Cursor* als Kontextobjekt zugrunde liegt. Folgende Möglichkeiten existieren:

- Volle Sperrverwaltung und –weitergabe
  Hierbei wird grundsätzlich vereinbart, daß die Sperrverwaltung von der Aktion der Skriptsprache — und somit letzendlich von der Transaktion des Datenbanksystems — entkoppelt und an die Lebensdauer des Cursor–Kontextelementes gebunden wird. Im Beispiel des *Mini–Batch–Problem* (siehe dazu Abschnitt 2.1 auf Seite 5) wird die Abarbeitung einer Menge von Buchungen in Teil–Transaktionen unterteilt, um die zwischenzeitlich für die erfolgten Buchungen gesperrten Konten–Datensätze wieder frei zu geben[9], um den Parallelitätsgrad[10] zu erhöhen und gleichzeitig die Sperrverwaltung zu entlasten.

- Statische Evaluierung des Cursors
  Wird die Transaktion zeitweise 'suspendiert', um Sperren untergeordneter Strukturen (*nested transaction*) freizugeben, so ist selbstverständlich, daß sie danach auf unveränderten Daten fortfahren will. Für die Gewährleistung sind unter anderem folgenden Mechanismen denkbar:
  - Explizite Sperranforderung
    Für die relevanten Datensätze einer Aktion werden vorab Sperren angefordert, sei es in Form einer Relationensperre oder in Form simulierter Tupelsperren. Bei letzerem Verfahren differenziert die Kontextverwaltung Sperren verschiedener Applikationen bezüglich ganzer Relationen, was natürlich den Grad der Parallelität erheblich steigert. Beide Methoden haben zur Folge, daß die zu verarbeitenden Daten nicht von anderen Transaktionen verändert werden können, solange der Cursor bzw. sein Kontext aktiv ist. Die Sperren werden automatisch an Folgeaktionen weitergegeben und erst nach der Nutzung[11] des zugehörigen Kontextobjekts freigegeben.
  - Temporäre Relation
    Eine weitere Möglichkeit besteht darin, die Ergebnismenge des Cursors in einer temporären Relation zu speichern, für die nur die jeweilige Transaktion das Zugriffsrecht besitzt, und so Änderungen von außen ausgeschlossen sind.

[9] X–Sperren werden sonst bis zum Ende der Transaktion gehalten.
[10] *Vice–President–Problem*
[11] Der Cursor wird in der Kontextbindung einer Aktion nicht mehr als Ausgabekontext geführt.

- Keine Sperrverwaltung
  Die der Ergebnismenge des Cursors zugrundeliegenden Tupel und Relationen werden von der Kontextverwaltung nicht gesperrt. Somit sind nur die Sperren im Rahmen der normalen Transaktionen vorhanden. Die Kontextverwaltung übernimmt lediglich die Sicherung von der Beschreibung der zugrundeliegenden Anfrage und Position des Cursors für die Lebensdauer des zugehörigen Kontextelementes. Eine Garantie für die Konsitenz des Anfrageergebnisses wird nicht gegeben. Änderungs-, Einfüge- und Löschoperationen sind nach der Beendigung einer Aktion sichtbar und somit i.allg. vor Ende der Nutzung des Cursors.

Offensichtlich ist jedoch eine enge Kopplung von Kontextverwaltung und Datenbanksystem bzw. dessen Transaktions–Managers notwendig, um einer der oben geforderten Funktionalitäten zu erreichen. Der Grad der Kopplung ist abhängig von der Wahl der Interpretation und ist Gegenstand der Realisierung. Dabei kann eine Erweiterung der jeweiligen Datenbank als nicht wahrscheinlich betrachtet werden, worauf bei den nun ausführlicher betrachteten Implementierungsansätzen Rücksicht genommen wird.

Cursor mit voller Sperrverwaltung und –weitergabe

Die volle Verwaltung bzw. Weitergabe von Sperren auf Datenbankobjekten ist, unter der Prämisse, daß im ConTract-Prototypen ein existierendes Datenbanksystem verwendet werden soll, unabdingbar mit der Konsequenz verbunden, daß eine Transaktion nicht an eine Aktion (oder einen Prozeß) gekoppelt sein darf. Dies bedeutet wiederum, daß alle Zugriffe auf Objekte des Datenbanksystems über eine Zwischeninstanz erfolgen <u>müßen</u>, die Träger der Transaktion ist. Alle Anfragen und Operationen mit dem DBMS erfolgen dann über diese Instanz, da sonst die Gefahr einer Endlos–Wartesituation auftreten kann[12].

Vorteil:

- Sperren können über einzelne Aktionen hinweg aufrechterhalten werden, da sie automatisch an Folgeaktionen weitergegeben werden.

Nachteile:

- Die Anfragen an die Datenbank müssen zur Vermeidung von Wartesituationen über die Zwischeninstanz erfolgen. Dazu werden sämtliche Zugriffe auf einen Cursor in der Wirtssprache durch Kommandos für diese Zwischeninstanz ersetzt. Die Kontrolle erfolgt entweder zur Laufzeit oder bei der Vorübersetzung. Dabei muß allerdings gewährleistet sein, daß eine Umgehung der Kontrollinstanz nicht erfolgt.

[12] Es wird eine Relation mit Hilfe eines Cursor über die Zwischeninstanz gelesen und gleichzeitig werden die gelesenen Daten direkt oder indirekt, unter Umgehung der Zwischeninstanz, geändert. Dies führt zu einer Wartesituation, da die Zwischeninstanz in ihrer Transaktion Lesesperren auf Objekte hält, auf die die Aktion Schreibsperren erwerben möchte.

- Die Zwischeninstanz als Träger der Transaktion muß sicher ablaufen. Tritt hier ein Fehler auf oder 'stürzt' die Zwischeninstanz ab, so wird die Transaktion zurückgesetzt und alle Operationen, die seit Beginn durchgeführt wurden, sind verloren[13].

- Da die Aktionen eines ConTracts nicht zwangsweise auf ein und demselben Rechner ablaufen müssen (ein ConTract kann migrieren), ist die Zwischeninstanz nicht an die Aktion, sondern an die verwendeten Datenbanken gebunden. Die Zwischeninstanz migriert i.allg. also nicht auf einen anderen Rechner (in einer vernetzten Umgebung), da sonst eventuell zu haltende Sperren verloren gingen. Diese Tatsache erfordert aufwendige Maßnahmen, um für die Zwischeninstanz ausreichende Kommunikationsmittel bereitzustellen.

- Die komplette Sperrweitergabe führt zu einer Zentralisierung der Verarbeitung der Datenbankoperationen auf wenige Knoten. Des weiteren können ConTracts sehr lange Ausführungszeiten haben. Dies hat zur Folge, daß Sperren sehr lange gehalten werden müssen und die Maßnahmen für den Ausfall der Datenbank so entworfen werden müssen, daß sie effizient und weitgehend automatisiert ablaufen, denn ein Ausfall, auf einen langen Zeitraum betrachtet, ist erheblich wahrscheinlicher.

Bemerkungen:

- Soll das Konzept der vollen Sperrverwaltung und -weitergabe realisiert werden, ist es erforderlich, die Datenbanksysteme zu erweitern. Für einen DBMS-Hersteller ist es sicherlich einfacher als für die Kontextverwaltung, eine Möglichkeit zu schaffen, Transaktionen nicht an Prozesse zu koppeln. Dies würde eine erhebliche Erleicherung zur Folge haben.

  Wird eine Datenbank eingesetzt, die ein Sperrprotokoll der Konsistenzebene drei[14] einsetzt, (siehe dazu auch [LSch87], Kapitel über Maßnahmen zur Wahrung von Sicherheits- und Integritätsbedingungen) ist auch eine Erweiterung dahingehend möglich, die erworbenen Lesesperren auf Objekte der DB vor Ende einer Aktion (Step) freizugeben. Dadurch wird zwar der Fundamentalsatz des Sperrens[15] verletzt, entspricht aber im Prinzip dem Sperren der Konsistenzebene zwei[16]. Die Anzahl der Sperren wird dadurch erheblich reduziert und das Auftreten des Problems der *Inkonsistenten Analyse* kann für bestimmte Anwendungen sicherlich ausgeschlossen werden bzw. ist unerheblich (zum Beispiel, dann wenn keine Analyse erfoderlich ist).

[13] Verloren gehen alle Operationen, die nach dem Öffnen des ersten Cursors durchgeführt wurden. Dies gilt auch für Operationen, die nicht in direktem Zusammenhang mit einem Cursor stehen, jedoch nach der betreffenden Öffnung erfolgt sind. Wurde überhaupt kein Cursor verwendet, können sämtliche Operationen bestätigt (**COMMIT**) werden.

[14] Die Transaktion hält lange X–Sperren auf den Objekten, die sie verändert und kurze auf solche, die sie liest.

[15] siehe [LSch87] Abschnitt 4.4.2.4 Seite 426

[16] Die Transaktion hält lange X-Sperren, aber nur kurze S–Sperren

Statische Evaluierung des Cursors

Um eine Garantie geben zu können, daß ein Cursor bei erneuter Anwendung (`OPEN CURSOR`) die gleiche Ergebnismenge besitzt, bedarf es gewisser Einschränkungen der Funktionalität des Cursors. Verschiedene Möglichkeiten, in Form einer Implementierung als Erweiterung des DBMS, sind folgende:

1. **Explizite Sperranforderung durch Relationensperren.**
   Die einfachste Möglichkeit, eine Veränderung eines Anfragergebnisses zu verhindern, ist es, das Einfügen, Löschen und Ändern der Zeilen der im Cursor verwendeten Tabelle zu verhindern. Voraussetzungen hierfür sind:

   a.) Der Cursor wird als **READONLY**-Cursor eingesetzt. D.h. , es werden direkt über den Cursor keine Änderungen an Objekten vorgenommen.

   b.) Des weiteren ändert die Aktion auch keine Objekte aus Relationen, die im Cursor benützt werden, über andere Statements.

   c.) Es muß im laufenden (Multiuser–)Betrieb möglich sein, die Tabellen für Schreiboperationen zu sperren.

   d.) Schließlich muß das verwendete DBMS einen Befehl zur Verfügung stellen, der es erlaubt, die gesamte Relation zu sperren. Gesperrt werden soll dabei so, daß nur noch Lesetransaktionen ablaufen können. Diese Anweisung gehört nicht zum **ANSI-SQL**-Standard, wird aber von fast allen Datenbanksystemen angeboten (z.B. **ORACLE**, **TRANSBASE**, **NonStop-SQL** und **INFORMIX**). Die Anweisungen lauten in deren SQL–Dialekt meist `LOCK TABLE tablename in SHARE MODE`.

   Sind alle Vorrausetzungen erfüllt, dann kann vor der Benutzung des Cursor durch ein Aktionsprogramm ein Prozeß gestartet werden, der die im Cursor benötigten Tabellen mit dem oben erwähnten Befehl sperrt. Dieser Prozeß, im folgenden Cursorprozess (CP) genannt, hält die Sperren bis zur letzten Verwendung des Cursor durch ConTract–Steps. Ein Ablaufdiagramm soll dies verdeutlichen:

Aktion	**Cursorprozeß (CP)**
	`LOCK` Tabelle `IN SHARE MODE`
{Beginn Aktion 1}	
`OPEN CURSOR`	
⋮	
`CLOSE CURSOR`	
{Ende Aktion 1}	
⋮	
{Beginn Aktion n}	
`OPEN CURSOR`	
⋮	
`CLOSE CURSOR`	
{Ende Aktion n}	
	`UNLOCK` Tabellen

Vorteile:

- Es existiert eine einfache Möglichkeit, die identische Evaluierung eines Cursor zu gewährleisten.
- Die Implementierung ist einfach. Die einzige Schwierigkeit besteht darin, dafür zu sorgen, daß die Sperren durch den CP stabil aufrecht erhalten werden. Eine Möglichkeit besteht darin, nicht einen, sondern mehrere CP einzusetzen, welche die Tabellen mehrfach, von verschiedenen Rechner aus, in `SHARE MODE` sperren. Die so erzeugten CP's sind dann unabhänig voneinander sehr einfach zu programmieren und erhöhen die Sicherheit erheblich. Eine Garantie für den Ausfall eines CP kann zwar dann nicht gegeben werden, doch ist die Wahrscheinlichkeit, daß durch den Ausfall eines CP die Sperren aufgegeben werden, gering. Die Gewährung einer Garantie ist wiederum nur durch die Entkopplung der Transaktion von einem Aktionsprogramm zu realisieren, und somit ist eine Erweiterung des DBMS notwendig.
- Es werden keine Sperren verwaltet, die sich auf eine Granularität beziehen, die unter der von Tabellen liegt. D.h. es ensteht kein Performanceverlust durch die Verwaltung einer größeren Anzahl von Sperren.

Nachteil:

- Die erwähnten Voraussetzungen schränken den Anwendungsbereich erheblich ein. Dies bedeutet insbesondere, daß während der Phase des Sperrens keine Änderungen durchgeführt werden können. Insbesondere kann es sein, daß der Cursor nur wenige Tupel selektiert, jedoch alle Datensätze der Tabelle für Änderungen gesperrt sind.

Bemerkung:

- Bei dem oben skizzierten Ansatz ist es für keine Applikation möglich, Änderungen während der Lebensdauer des Cursor in den Datenbestand einzubringen. Wird der Cursorprozeß nicht nur zum Sperren verwendet, sondern zur im vorherhegenden Abschnitt erwähnten Zwischeninstanz erweitert, so können die einzelnen Aktionen auch ändernd zugreifen.

2. **Sperren durch Simulation von Tuppelsperren.**
Das Sperren einer Menge von Tupel kann nicht nur durch das DBMS erfolgen. Vielmehr kann bei geeigneter Modellierung die Kontextverwaltung selbst Sperren auf Objekte der Datenbank verwalten. Dies geschieht dadurch, daß vor der Benuztung des Cursor die Kontextverwaltung an den betroffenen Datensätzen der Relationen eine Kennung hinterläßt, die angibt, von wem und in welcher Weise die einzelnen Zeilen gesperrt sind. Z.B. wird die Relation um zwei Attribute erweitert. Das erste gibt an, von wem das Tupel gesperrt ist, und das zweite bestimmt den Sperrmodus. Die Kontextverwaltung übernimmt hier also die Aufgaben, die vergleichbar sind mit denen von 'Checkin/out' Funktionen, die dafür sorgen, daß die Objekte aus dem Multiuserbetrieb ausgeschlossen bzw. diesem wieder zur Verfügung gestellt werden. Normale Transaktionen können auf diese Tupel nicht zugreiffen, wenn die entsprechenden Attribute gesetzt sind. Realisiert werden kann dieses Verbot in SQL durch das VIEW–Konzept (Transaktionen sehen Tupel nur, wenn das Attribut, das den Sperrmodus bestimmt, einen entsprechenden Wert bzw. Werte besitzt).

Vorteile:

- In diesem Fall besteht kein Bedarf einer Sperrverwaltung durch das DBMS. Eine Entkopplung von Transaktion und Prozeß ist nicht erforderlich. Allein das ConTract–System, in Zusammenarbeit mit der Kontextverwaltung, muß für eine Konsistenz der Sperren sorgen. Dies ist besonders im Fehlerfall nötig, bereitet aber keine größere Schwierigkeit, da hier die angesprochenen Komponenten ohnehin tätig werden müssen.
- Das Sperrgranult ist so gering wie möglich, Parallelität bleibt weitestgehend erhalten.
- Die Sperren sind sicher gegenüber dem Ausfall von Rechnerknoten, Kommunikationsverbindungen, der Datenbank und Teilen davon, sowie des ConTract-Systems selbst.
- Die Sperranforderung und Freigabe können als einfache und atomare UPDATE-Statements formuliert werden. Bei entsprechendem Sperrprotokoll der Datenbank kann die Konsitenz garantiert werden.
- Bei Benutzung des VIEW–Konzepts kann dem normalen Benutzer seine logische Sicht auf die Daten erhalten bleiben (er sieht die Erweiterung um zusätzliche Attribute über den VIEW nicht).

Nachteil:

- Das eigentliche Sperren und Freigeben der betroffenen Tupel benötigt einen erheblichen Aufwand vor allem an Zeit, ist also nur dann sinnvoll, wenn der Nutzen dieses Verfahren entsprechend groß ist.

3. **Temporäre Relation.**
Das Ergebnis der Cursor–Anfrage wird vor der ersten Benutzung durch ein Aktionsprogramm von der Kontextverwaltung berechnet und in einer temporären Relation gespeichert, auf die andere Transaktionen keinen Zugriff haben. Die Aktionsprogramme lesen dann nur aus der Hilfsrelation. Hierzu ist es erforderlich, die Anfrage, welche dem Cursor zugrunde liegt, dahingehend zu modifizieren, daß sie nur auf die entsprechende temporäre Tabelle zugreift. Die Kontextverwaltung sichert also in diesem Fall die Anfrage, die Abarbeitungsposition und den Namen der Zwischenrelation. Ist die letzte Aktion, die den Cursor verwendet, abgeschlossen, wird die Hilfsdarstellung gelöscht. Dieses Vorgehen wurde im ConTract-Prototypen implementiert, der Kontexttyp erhielt den Namen `Materialisierter Cursor`.

Vorteile:

- Einfache Realisierung.
- Ist die Evaluierung des Cursors aufwendig, können Kosten eingespart werden, da der Zugriff auf die temporäre Relation i.allg. billiger ist.
- Die Vorgehensweise im Fehlerfall ist einfach. In Normalfall sind keine Tätigkeiten erforderlich, denn die Markierung der Tupel bleibt erhalten.

Nachteil:

- Es können keinerlei Garantien für die Existenz der Datenbankobjekte, die bei der Evaluierung betroffen waren, zum Zeitpunkt des Zugriffs gegeben werden.

Keine Sperrverwaltung

Es werden durch die Kontextverwaltung nur der Anfragetext des Cursor–Statements sowie die Abarbeitungsposition gesichert. Jegliche Konsistenzzusicherung entfällt, die Aufrechterhaltung der semantischen Integrität des Datenbestandes obliegt den Applikationen.

Sinn macht dieses Vorgehen u.a. bei statistischen Auswertungen, welche die anderen Anwenderprogramme nicht beeinträchtigen sollen und zudem nicht notwendigerweise eine 'Konsistente Analyse' erfordern. Ein weiteres Einsatzgebiet ist im Zusammenhang mit Anwendungen wie dem *Molekül-Atom-Datenmodell (MAD)* zu sehen. Dieses ist ein Konzept zur Verarbeitung von DB-Objekten im Bereich der Ingenieuranwendungen durch Unterstützung der Verarbeitung netzwerkartiger Strukturen mit Hilfe eines *Hierarchischen Cursor–Konzepts*. Hierarchische Cursor sind hier zusammenhängende SQL-Cursor, auf die Art, daß beim Weiterschalten eines Cursors auf das nächste Atom einer Hierarchiestufe alle Cursor der tieferliegenden Stufen ebenfalls auf einen neu zu bestimenden Cursorbereich verweisen[17]. Die Aufrechterhaltung der Konsistenz obliegt hier allein dem MAD-System. Eine Modellierung im Rahmen eines ConTracts kann für die Kontextverwaltung kaum mehr bedeuten, als zu den benutzten Cursor die oben erwähnten Attribute ohne Konsistenzzusicherung zu verwalten.

3.2.3 Schwach strukturierte Kontextobjekte

Einfache Kontextobjekte wurden im vohergehenden Abschnitt ausführlich vorgestellt. Worin liegt nun der Unterschied zwischen 'Einfachen Kontextobjekten' und 'Schwach strukturierten'?

Im Gegensatz zu den einfachen Kontextobjekten bedürfen schwach strukturierte einer Interpretation. Diese Interpretation muß nicht unbedingt von einem Compiler oder Interpreter einer Programmiersprache vorgenommen werden können. Ein Beispiel hierfür sind Listen, Bäume, oder allgemeiner, Netze. Die Sicherung eines solchen Netzes kann nicht zwangsweise von der Kontextverwaltung vorgenommen werden. Die einzelnen Knoten entsprechen i.allg. Strukturen und können wie oben besprochen verwaltet werden. Die Sicherung des Zusammenhangs der einzelnen Knoten muß aber vom Programmierer vorgenommen werden. Dies trifft besonders dann zu, wenn auf den einzelnen Knotenobjekten noch Prädikate existieren, deren Wert darüber bestimmt, ob ein solcher Knoten überhaupt als Kontextelement anzusehen ist.

D.h. schwach strukturierte Kontextobjekte können meist auf einfache Kontexttypen abgebildet werden. Diese Abbildung ist aber nicht zwangsweise trivial und durch einen Automatismus auszuführen.

3.2.4 Komplexe Kontextobjekte

Diese Klassifikation ist wahrscheinlich am besten zu verstehen, wenn man sich vor Augen führt, welche Zielsetzung ConTracts haben. Sie dienen zur unverwundbaren Ausführung von Vorgängen in verteilter, paraller Umgebung. Einsatzgebiete im kommerziellen Bereich ergeben sich hieraus z.B. im Fertigungsbereich oder in der Bürokommunikation.

Im Fertigungsbereich gilt als Ziel die *Fabrik der Zukunft*. In ihr wird eine durchgängige Unterstützung der Planungs-[18], Entwurfs-[19], Fertigungs-[20] und Qualitätssicherungsaufgaben[21] durch den Rechner[22] angestrebt. Daraus ergeben sich typische Abläufe, die in ConTracts

[17] Entwickelt wurde das *MAD* am Zentrum für Rechnergestützte Ingenieursysteme an der Universität Kaiserslauten (siehe [HS89] und [HHS89]).

[18] **CAP** ··· Computer Aided Planning

[19] **CAD** und **PPS** ··· Computer Aided Design, Product Planning System

[20] **CAM** ··· Computer Aided Manufacturing

[21] **CAQ** ··· Computer Aided Quality Assurance

[22] **CIM** ··· Computer Integrated Manufacturing

modelliert werden, und somit einem Transaktions– und Recovery- bzw. Kompensationskonzept unterliegen. Die Aufgabe der Kontextverwaltung in diesen Abläufen ist Sicherung und Weitergabe des 'erarbeiteten'Wissens. Gerade am Beispiel der Fertigungstechnik wird sichtbar, daß es auf die Interpretation der Daten ankommt. Denn aus einer CAD-Konstruktion läßt sich nicht trivialerweise ein Programm zur Steuerung einer Fertigungseinrichtung erstellen. D.h. , der Output eines Programmes muß interpretiert werden. Diese Eigenschaft führt zur Einstufung des Kontextes als *komplex*. Für die *Fabrik der Zukunft* ist aber dieses Zusammenspiel notwendig. Es ist im Bereich der Fertigungstechnik noch erheblicher Entwicklungsaufwand notwendig, um geeignete Schnittstellen zu entwerfen, zu testen und zu normen. Solange dies aber nicht geschehen ist, wird es keine andere Möglichkeit geben, als Transformationsprogramme zur implementieren. Somit bleibt die Interpretation aus der Kontextverwaltung ausgegrenzt[23]. Allerdings entspricht dies nicht dem Kontextprinzip, da der Verarbeitungszustand einer Aktion nicht zwangsweise von der Kontextverwaltung übernommen und sicher verwahrt wird.

Die Einstufung als **komplexer Kontext** beruht in diesen Beispielen auf dem Aufwand, der für die Interpretation notwendig ist, um Objekte in Folgen von Aktionen verwenden zu können.

Im zweiten angeführten Bereich, der Bürokommunikation, stößt man auf ähnliche Schwierigkeiten. Hier geht es direkt um die Integration verschiedenster Rechnersysteme. Vom Ablauf und somit von einem Contract sind hier Rechner mit unterschiedlichster Hardware (z.B. Perconal Computer, Workstations bis hin zur Großrechnern), unterschiedlichsten Betriebssystemen und Standardsoftwarepaketen zu integrieren. Das Zusammenspiel dieser Systeme zu gewährleisten kann 'Kontext' erfordern, der ebenfalls nicht trivial sein muß.

Eine Kontextart, die auch als komplex eingestuft werden muß, ist die Verwaltung der Kommunikationsverbindungen. Bei der Integration in verteilten Systemen spielen Netze mit unterschiedlichen Kommunikationsprotokollen ein große Rolle. Die Aufrechterhaltung der Verbindung und die Schaffung von Ersatzwegen ist eine Anforderung an das ConTract-System, die nicht nur eine sichere Ausführung der Aktionen gewährleistet, sondern auch eine Verwaltung dieser durch den Aktionsprogrammierer überflüssig macht und damit eine erhebliche Kosteneinsparung zur Folge hat.

Abschließend läßt sich sagen, daß als **Komplexe Kontextarten** eingestufter Kontext sicherlich einsetzbar ist. Die Verwaltung wird aber, bis auf wenige Ausnahmen, auf den Aktionsprogrammierer zurückfallen.

[23]Ob eine Intergration des Interpretationsbedarfs in die Kontextverwaltung überhaupt sinnvoll oder möglich ist, wird in anderen Studienarbeiten ansatzweise untersucht. Das hier erwähnte Gesamtziel ist durchaus als Traumvorstellung aufzufassen.

Kapitel 4

Schemata zur Kontextverwaltung

Nachdem im Kapitel 3 die Möglichkeiten zur Verwaltung verschiedener Kontexttypen diskutiert wurden, soll hier nun eine Betrachtung der Variationsmöglichkeiten, die für die prinzipiellen Schemata zur Verwaltung des Kontextes in Frage kommen, näher untersucht werden. Nicht jede Variante ist in allen Fällen einsetzbar; für die Fälle, in denen mehrere Möglichkeiten zur Verfügung stehen, gelten folgenden Bewertungskriterien:

- Das Lesen und Schreiben des Kontexts soll mit möglichst wenigen Datenbankzugriffen erfolgen.
- Hohe Zugriffsraten auf einzelne Datenelemente (Hot Spots) sollten vermieden werden.
- Das Schema soll einen möglichst hohen Grad an Parallelität zulassen.
- Vermeidung von Typenumwandlung der Kontextelemente bei der Übertragung vom und zum Kontextspeicher, damit keine Compiler– und Rechnerabhängigkeiten entstehen.
- Der Kontext eines ConTracts muß jederzeit verfügbar sein, um eine Blockierung der Anwendung zu verhindern.
- Zukünftige Erweiterungen des Schemas und der Zugriffsfunktionen sollten Berücksichtigung finden.
- Das Schema soll die verteilte Verarbeitung unterstützen.
- Soweit möglich soll zur Erhöhung der Ausfallsicherheit des ConTract-Systems beigetragen werden.

4.1 Kontextelemente als Attribute einer Relation

Alle vorkommenden Kontextelemente eines ConTracts werden in einer Relation gespeichert. Die verwendete Tabelle hat somit folgendes Aussehen:

Historie	Änderung durch Aktion	Element 1	Element 2	⋯	Element n

Vorteile:

- Das Schema ist einfach.
- Alle Elemente können mit einem Zugriff gelesen werden. Hierbei wird vorausgesetzt, daß die Kontextverwaltung beim Sichern des Kontextes einer Aktion jeweils die neueste Version des Eintrags liest, diesen ändert und als neues Tupel einträgt.

Nachteile:

- Auf die Relation ist kaum eine paralleler Zugriff möglich. Insbesondere beim Einfügen eines neuen Eintrags muß gewährleistet werden, daß eine strenge Sequentialisierung stattfindet, da sonst das 'Lost–Update–Problem' auftreten könnte.
- Da für jeden ConTract ein anderes Schema erforderlich ist, müssen die Funktionen der Kontextverwaltung flexibel genug sein, um mit diesen umgehen zu können. Dies erfordert u.a. dynamische Anfragen zum Lesen und Schreiben. Insbesondere der Mechanismus zum Lesen einer beliebigen Anzahl von Attributen unterschiedlichster Typen wirft Probleme auf, denn er ist im **ANSI/ISO–SQL**–Standard nicht spezifiziert. Es entsteht hier also eine Abhänigkeit vom DBMS.
- Eine Abspeicherung in einer Relation reicht nicht immer aus. Manche Kontexttypen erfordern eine Verteilung über mehrere Tabellen. Als Beispiel sei hier auf die Sicherung von Feldern verwiesen (siehe Kapitel 3 Seite 24).
- Es besteht keine Möglichkeit, Kontext dynamisch zu erzeugen.

Bewertung:

Das Schema unterstützt nicht die parallele Verarbeitung, erzwingt eine Sequentialisierung und enthält keine dynamischen Möglichkeiten. Ein weiterer Mangel ist die unzureichende Allgemeinheit. Es können zudem nicht alle Kontexttypen verwaltet werden.

4.2 Pro Aktion eine Relation mit allen Kontextelementen

Für jede Aktion wird eine Relation erzeugt, in der ihr Ausgabekontext gesichert wird.

Vorteile:

- Der Zugriff beim Schreiben und Lesen der Kontextelemente ist schnell.
- Dadurch treten keine Hot-Spots auf.
- Es entstehen viele kleine Relationen mit einem oder wenigen Tupel. Dies ermöglicht einen schnellen Zugriff, da die Tabelle und ihre Zugriffstrukturen im Pufferbere:w ich des DBMS Platz finden.
- Der Parallelitätsgrad beim Lesen und Schreiben ist groß.
- Durch die vielen Tabellen wird die verteilte Verarbeitung unterstützt. Außerdem wird dadurch die Ausfallsicherheit erhöht. Eine Diskussion dieses Punktes erfolgt im Kapitel über Maßnahmen zur Erhöhung der Ausfallsicherheit (Kapitel 6 ab Seite 57).
- Es ist möglich, den Kontext im Hinblick auf die Aktionen dynamisch zu erweitern. Soll auch eine Erweiterung dahingehend erfolgen, daß dieselben Aktionen bei verschiedenen Aktivierungen verschiedenartigen Kontext produzieren, steigt der Verwaltungsaufwand erheblich.

Nachteile:

- Beim Lesen der Kontextelemente muß die Kontextverwaltung die Elemente i.allg. aus verschiedenen Relationen zusammentragen. Dies erfordert eine geeignete Synchronisation, um bei einer parallele Verarbeitung in jedem Fall die richtigen Werte zu erhalten.
- Die Verwaltung des Kontexts in einer Tabelle ist nicht allgemein genug. Allerdings läßt sich dieser Nachteil hier leichter beheben als bei der vorhergehenden Variante, denn hier können einfach zusätzliche Tabellen erzeugt werden.

Bewertung:

Diese Variante verspricht den größten Grad an Parallelität und die besten Zugriffszeiten. Allerdings steigt der Verwaltungs- und Synchronisationsaufwand für die Kontextverwaltung, wenn keinerlei Einschränkungen gemacht werden.

4.3 Kontextelemente als Tupel einer Relation

Alle vorkommenden Kontextelemente werden in einer Relation verwaltet.

Historie	Änderung durch Aktion	Elementname 1	Wert
Historie	Änderung durch Aktion	Elementname 2	Wert
Historie	Änderung durch Aktion	Elementname 3	Wert
Historie	Änderung durch Aktion	Elementname 2	Wert
⋮	⋮	⋮	⋮

Für die Abspeicherung der Werte ergeben sich mehrere Möglichkeiten:

- Das mit 'Wert' bezeichnete Feld entspricht in Wirklichkeit einer Ansammlung von Attributen verschiedenen Typs, und zwar von jedem Typ eines. Der Wert des Kontextelementes wird dann je nach Typ im entsprechenden Attribut gespeichert. Für den Fall, daß es Kontexttypen gibt, bei denen zur Verwaltung mehrere Attribute benötigt werden, kann das Schema entsprechend ergänzt werden.
- Das Feld Wert ist vom Typ `STRING` bzw. `BINARY OBJECT`[1]. Gespeichert wird dann die Darstellung der Kontexttypen im Rechner. Eine weitere Möglichkeit besteht darin, alle Kontexttypen als Zeichenketten darzustellen. Für Zahlen ergibt sich daraus keine Schwierigkeit, bei anderen Typen muß eine entsprechende Kodierung vorgenommen werden.

Vorteile:

- Das Schema ist einfach.
- Kontext kann dynamisch erzeugt werden.
- Kontextverwaltung kann einfach und kompakt implementiert werden.

Nachteile:

- Die Relation wird sehr groß.
- Auf die Relation greifen viele Transaktionen gleichzeitig zu. Dies kann zum Phänomen der 'Hot-Spots' führen.
- Je nach Wahl der Wertabspeicherung muß eine Typumwandlung durchgeführt werden. Daraus ergeben sich Konvertierungsfehler und Abhängigkeiten von den einzelnen Rechnern bzw. deren CPU's.
- Es können Fehler bei der Typumwandlung enstehen.

[1] Die Entwicklung im Bereich der kommerziellen Datenbank geht dahin, sogennante *Binary Large Objects* (*BLOBS*) zur Verfügung zu stellen. In diesen können Bitfolgen gesichert werden, die vom DBMS uninterpretiert bleiben.

Bewertung:

Diese Implementierung scheitert an den starken Abhängigkeiten vom Rechner. Außerdem ist der Parallelitätsgrad gering und das Auftreten von 'Hot-Spots' ist hier am wahrscheinlichsten.

4.4 Für jedes Kontextobjekt eine eigenen Relation

Jedes Objekt wird in einer eigene Relation bzw. Relationen, wie im Kapitel über Kontextarten und Klassifikation bereits erläutert, verwaltet. Die Verwaltung kann erfolgen durch:

- Referenzierung durch den Typ.
  Die Kontextelemente werden so verwaltet, daß anhand ihres Typs die entsprechende Tabelle gewählt wird. Die Referenzierung der einzelnen Elemente erfolgt dann innerhalb der Relation durch den Namen und entsprechende Angaben zur Unterscheidung der Versionen.
- Adressierung durch den Namen.
  Der Name und die Versionsangaben sind in einer Hauptrelation gespeichert. In dieser steht ein Verweis auf den entsprechenden Datensatz in einer zuständigen Relation.

Vorteile:

- Es ist möglich, alle Kontexttypen zu verwalten.
- Durch den Einsatz mehrerer Tabellen ist eine Verteilung auf verschiedene Speichermedien möglich. Dies erhöht die Ausfallsicherheit und unterstützt die parallele Verarbeitung.
- Die Migration von ConTracts und auch der Kontextverwaltung und Teilen bzw. Relationen davon ist dadurch leichter.
- 'Hot Spots' auf einzelnen Datenelementen werden weitgehend vermieden.
- Eine dynamische Erweiterung ist möglich.
- Die Funktionen zur Kontextverwaltung können leicht nach Kontextobjekten modularisiert werden. Dies vereinfacht die Funktionen und erleichtert somit den Entwurf, die Implementierung, Wartung und die Kontrolle der Software.

Nachteil:

- Es müssen mehrere Relationen verwaltet werden.

Bemerkung:

- Es besteht die Möglichkeit, die Tabellen systemweit oder für jeden ConTract zu erzeugen. Wird die Variante 'für jeden ConTract' gewählt, so ist eine weitere Verteilung und Parallelisierung möglich.

Bewertung:

Aufgrund der Vorteile im Bereich der Parallelisierung und Verteilung der Aufgaben und im Hinblick auf die mangelnde Allgemeinheit der anderen Varianten ist diese Möglichkeit eindeutig zu bevorzugen.

Im ConTract-Prototypen wurde diese Variante implementiert. Zur Referenzierung der Elemente wurde die oben erwähnte Methode durch Referenzierung des Typs gewählt, da der Typ des Kontextelementes beim Sichern bekannt ist und daraus direkt die entspechende Relation adressiert werden kann.

Kapitel 5

Die Kontextreferenzierung

Das folgende Kapitel stellt die Referenzierung eines Kontextelementes durch den Anwendungsprogrammierer im Skript und mögliche Implementierungsalternativen vor. Dabei werden nur die Teile des Kontextes betrachtet, die auf Skriptebene sichtbar sind und im folgenden als Kontextelemente bezeichnet werden. Alle anderen Teile eines ConTracts, wie z.B. die aktuelle Ausführungsposition im Skript, verwaltet der Skript-Interpreter. Dieser Teil des Kontextes sollte nicht von einem Step aus referenzierbar sein und wird deshalb hier nicht näher betrachtet.

5.1 Aufgaben der Kontextverwaltung

Innerhalb eines ConTracts werden Zwischenwerte, die ein Step berechnet hat, im Kontext abgelegt. Nachfolgende Steps können auf diese Werte zugreifen und neue Werte hinzufügen. Darüberhinaus liest und schreibt auch der Skript-Interpreter Kontextelemente; z.B. kann die Auswertung der Bedingung einer `if`-Anweisung Werte aus dem Kontext benötigen. Zum Zugriff auf den Wert eines Kontextelementes verwendet man den Namen des Kontextelementes, der auf Skriptebene global gültig ist und verschiedene Kontextelemente unterscheidet.

Bei der Erstellung eines Steps ist der im Skript verwendete Name des Kontextelementes, das gelesen oder geschrieben werden soll, noch nicht bekannt. Dies folgt aus der Tatsache, daß der Code eines Steps unabhängig von einem bestimmten Skript erstellt werden kann. Im Beispiel-Skript in Abbildung 2.1 verwenden die Steps `Init_MiniBatch` und `Compute_LoopCount` die Namen `first` und `anfang` für das in diesem Skript gleiche Kontextelement `low_acc_no`. Damit beide Steps das gleiche Kontextelement refernzieren können, ist der im Step verwendete Name für ein Kontextelement an den im Skript gegebenen Namen zu binden. Durch die Namensbindung im Skript kann man außerdem einen Step in einem Skript mehr als einmal oder auch in verschiedenen Skripten verwenden. Eine Teilaufgabe der Kontextverwaltung bei der Referenzierung eines Kontextelementes ist damit die Umsetzung des im Step verwendeten symbolischen Namens in den vom Skriptprogrammierer gewählten Namen eines Kontextelementes. Abschnitt 5.2 stellt Realisierungsmöglichkeiten dieser Teilaufgabe vor, die im oberen Teil von Abbildung 5.1 dargestellt ist.

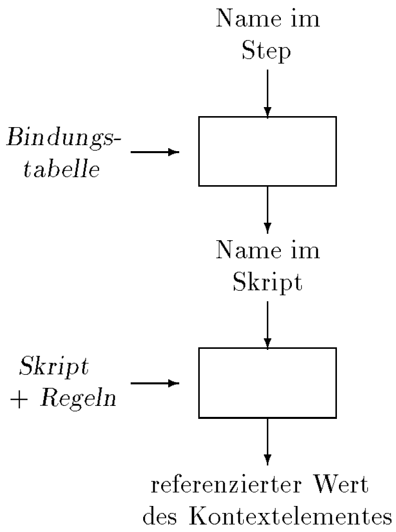

Abbildung 5.1: Referenzierung eines Kontextelementes

Das Schreiben eines neuen Wertes eines Kontextelementes überschreibt nicht den alten Wert, sondern legt eine neue Version dieses Kontextelementes mit dem neuen Wert an. Während der Ausführung eines ConTracts existiert damit für ein Kontextelement nicht nur ein einziger Wert, sondern es besitzt eine Menge von Werten, die ihm während der Ausführung zugewiesen wurde – die sogenannte *Wertehistorie*. Soll nun ein Step den aktuellen Wert eines Kontextelementes lesen, so muß ein Element aus dieser Menge von Werten ausgewählt werden. Das Anlegen einer neuen Version und die Auswahl eines Wertes aus der Wertehistorie durch implizite Regeln oder explizite Angabe im Skript ist eine weitere Teilaufgabe der Kontextverwaltung (Abbildung 5.1, unterer Teil). Auf diese Teilaufgabe wird ab Abschnitt 5.3 eingegangen und anhand der im ConTract-Prototyp verwendeten Skriptsprache werden Implementierungsmöglichkeiten und dabei auftretende Probleme vorgestellt.

An die Referenzierung und die Verwaltung der Kontextelemente stellen sich die folgenden Anforderungen:

- Über die Menge der Werte eines Kontextelementes muß eine Ordnung bestehen, da diese Menge die Wertehistorie dieses Kontextelementes darstellt. Dies kann man durch Angabe einer aufsteigenden *Versionsnummer* erreichen.
- Nach Beendigung eines Steps müssen alle von diesem Step erzeugten Kontextelemente ausfallsicher und von einem nachfolgenden Step referenzierbar geschrieben werden.
- Kontextelementwerte, die ein noch nicht beendeter Step erzeugt, dürfen für einen anderen Step nicht referenzierbar sein. Bei sequentieller Ausführung aller Steps ist dies

ohnehin ausgeschlossen. Die Probleme, die bei parallel ablaufenden Steps auftreten, werden in Abschnitt 5.5 näher betrachtet.

- Das Sichern des Kontextes eines Steps sollte keine Synchronisierung benötigen bzw. durch doch notwendige Synchronisierung sollten keine Deadlocks auftreten, da die Synchronisierung der Steps explizit im Skript durch die Angabe des Kontrollflusses erfolgt.
- Durch Kontrollflußanweisungen im Skript ist es möglich, Steps gleichzeitig parallel zueinander ablaufen zu lassen. Dies erlaubt auch das Auslagern der Bearbeitung auf andere Rechner (*Migration*). Um dabei die Parallelität nicht ungünstig einzuschränken, sollten die Kontextoperationen möglichst unabhängig auch auf mehreren Rechnern erfolgen. Z.B. könnte zu Beginn der parallelen Ausführung der aktuelle Kontext zu den anderen Rechnern übertragen und dort unabhängig verändert werden. Am Ende der parallelen Ausführung wären die entstandenen Kontextänderungen zum Gesamtkontext des ConTracts hinzuzufügen. Die Referenzierung eines Kontextelement-Wertes muß dabei unabhängig von einer etwaigen Migration sein.

5.2 Umsetzung des im Step verwendeten Namens

Steps können unabhängig von ihrer späteren Verwendung in einzelnen Skripten erstellt werden. Dadurch kann ein Kontextelement nur durch seinen symbolischen Namen im Step-Code referenziert werden. Zu welchem Zeitpunkt diese symbolische Referenz aufgelöst wird, unterscheidet nun die einzelnen Realisierungsmöglichkeiten:

1. Auflösung zur Compile-Zeit,
2. Auflösung zur Link-Zeit,
3. Auflösung zur Laufzeit.

Wird der symbolische Name vor dem oder während des Compilierens ersetzt, kann man den ausführbaren Code des Steps nur für diese Verwendung im Skript einsetzen. Verwendet man jedoch einen Step mehrmals in einem Skript oder in mehreren Skripten, so muß dafür der Code vervielfacht und recompiliert werden, da man nicht annehmen kann, daß der Step bei jeder Verwendung die gleichen Kontextelemente unter demselben Namen referenziert.

Bei der **Auflösung zur Link-Zeit** kann man zwei Fälle des Bindens unterscheiden. Das Binden des ausführbaren Codes, der noch offene Referenzen besitzen kann, ergänzt diesen um die bei der Ausführung notwendigen Referenzierungsinformationen. Diese Referenzierungsinformationen können vor der Ausführung des gesamten Skripts oder vor Ausführung des einzelnen Steps zum Code gebunden werden.

Im ersten Fall werden alle im Skript verwendeten Steps gebunden und bis zum Ende des ConTracts gespeichert. Wie schon bei der Auflösung zur Compile-Zeit vervielfacht sich auch hier der ausführbare Code eines mehrfach verwendeten Steps. Diese Möglichkeit erweist sich durch den hohen Bedarf der Ressource Speicherplatz als ungünstig.

Im zweiten Fall werden die Referenzierungsinformationen während der Ausführung des ConTracts ergänzt. Vor dem Start eines einzelnen Steps bindet man zum ausführbaren Step-Code, der offene Referenzen besitzt, die noch fehlenden Referenzierungsinformationen hinzu, so daß dieser Step anschließend ausgeführt werden kann. Bei dieser Vorgehensweise entfällt die Belegung unnötigen Speicherplatzes. Durch das wiederholte Binden desselben Steps in einer Schleife oder bei mehrmaligem Ablauf des gleichen Skripts erhöht sich gegenüber den bisher vorgestellten Realisierungen der Rechenzeitbedarf, jedoch ist er durch den simplen Link-Vorgang so gering, daß man ihn gegenüber der insgesamt benötigten Ausführungszeit eines Steps vernachlässigen kann.

Die **Auflösung** der offenen Referenzen erfolgt **zur Laufzeit** durch einen Teil des Step-Codes. Jeder Zugriff auf eine offene Referenz stößt eine Routine an, die die fehlenden Referenzierungsinformationen liest (z.B. aus einer Bindungstabelle) und mit Hilfe dieser Information das entsprechende Kontextelement referenziert. Im Vergleich zu den anderen Realisierungsmöglichkeiten werden hier keine Kopien des Step-Codes benötigt, analog zum Binden vor der Ausführung des Steps. Durch das Lesen der Referenzierungsinformation kann jedoch der Bedarf an Rechenzeit für diese Aufgabe relativ hoch sein, insbesondere bei sehr aufwendigen Zugriffsoperationen auf die notwendige Bindungsinformation.

Da die Referenzierungsinformation für jeden Step vor der Ausführung des Skripts bekannt ist und sich somit während des Ablaufs nicht ändern kann, ist die Möglichkeit des Bindens vor der Ausführung des einzelnen Steps den anderen aufgeführten Möglichkeiten vorzuziehen. Dies kann durch die konstante Information sehr effizient im Skript-Interpreter und im Step-Executor, die die Steps starten und die notwendigen Informationen aus dem Skript besitzen, implementiert werden. Als weitere Möglichkeit bietet sich die Auflösung zur Laufzeit an, falls man dies durch Angabe von Zugriffspfaden auf die Referenzierungsinformationen unterstützt. Stellt der Skript-Interpreter dem Step die Referenzierungsinformation explizit zur Verfügung, entspricht dies konzeptionell einem Link-Vorgang.

5.3 Referenzierungsmöglichkeiten

Jedes Schreiben eines neuen Wertes eines Kontextelementes erzeugt eine neue Version. Zu jedem Zeitpunkt der Ausführung eines Skriptes existiert damit für jedes Kontextelement eine Wertehistorie, eine Totalordnung dieser Versionen. Die Folge der *Versionsnummern* bildet diese Anordnung.

Ein Element aus der Wertehistorie eines Kontextelementes besteht aus dem Tupel

(ConTract, Kontextelementname, Versionsnummer, Wert).

Damit hat man die folgenden Referenzierungsmöglichkeiten eines Wertes:

1. Lesen der i-ten Version eines Kontextelementes,

2. Lesen der i.-letzten Version eines Kontextelementes,
3. Lesen der letzten Version eines Kontextelementes
   (Sonderfall der zweiten Möglichkeit).

Das Lesen der zuletzt erzeugten Version ist dabei die häufigste Zugriffsart auf den Wert eines Kontextelementes und sollte deshalb möglichst effizient unterstützt werden. Dieser Fall entspricht dem lesenden Zugriff auf den Wert einer Variablen in einer prozeduralen Programmiersprache.

Die beiden anderen Referenzierungsmöglichkeiten werden bei Bezug auf die Wertehistorie eines Kontextelementes benötigt, wie er z.B. beim kontrollierten Aufsetzen auf einen alten Ausführungszustand erfolgt.

Weitere Referenzierungsmöglichkeiten eines Kontextelementes kann man durch die Einschränkung der Versionsmenge des Kontextelementes durch die Angabe einer oder mehrerer zu erfüllenden Eigenschaften erhalten.

Solche Eigenschaften könnten z.B. durch zu erfüllende Attribute einer Kontextelement-Version angegeben werden. Dies könnte sein,

- der Step, der diese Version erzeugt hat,
- der Rechner, auf dem der Step ablief,
- der Zeitraum, in dem diese Version erzeugt wurde.

Diese Daten müssen bei der Erzeugung einer Version zusätzlich zum Wert gesichert werden. Durch diese Einschränkung der Versionsmenge eines Kontextelementes erhält man wieder eine Menge, die durch die Versionsnummer die gleiche Ordnung wie die Gesamtmenge aufweist. Aus dieser Menge kann man z.B. – wie aus der Gesamtmenge – die zuletzt erzeugte Version auswählen.

In dem in Abbildung 2.1 gezeigten Beispiel eines ConTracts wird bei der Referenzierung in den Steps `Set_NewValues` und `Receipt` der Erzeuger-Step angegeben, um die alten, vor der Ausführung der Schleife geschriebenen Werte der Kontextelemente zu erhalten.

5.4 Referenzierung bei serieller Ausführung

Beschränken sich die möglichen Ausführungsfolgen der Steps eines Skripts auf eine streng sequentielle Abfolge kann jeder erzeugten Version eines Kontextelementes sehr einfach eine eindeutige, aufsteigende Versionsnummer zugewiesen werden, die die Wertehistorie dieses Kontextelementes ersichtlich macht.

Will ein Step nun auf die letzte Version eines Kontextelementes zugreifen, so kann anhand der Versionsnummer eindeutig entschieden werden, welche der existierenden Versionen auszuwählen ist. In diesem Falle wäre dies die Version mit der höchsten existierenden

Versionsnummer. Die Referenzierungsmöglichkeiten i-te bzw. i.-letzte Version kann auch sehr einfach durch sequentielles auf- bzw. absteigendes Durchwandern der Versionen realisiert werden, falls die Versionsnummern nicht lückenlos vergeben wurden. Bei lückenloser Vergabe kann dies durch direktes Selektieren der entsprechenden Version über die Versionsnummer erfolgen. Eine lückenlose Vergabe der Versionsnummern wird in den folgenden Betrachtungen nicht angenommen, da ein ConTract parallele Abläufe erlaubt, die eine lückenlose Vergabe erschweren. Abschnitt 5.6 geht auf die Probleme bei der Vergabe einer Versionsnummern ein und stellt Realisierungsmöglichkeiten vor.

5.4.1 Sequenz

Die Kontrollflußanweisung **seq** legt fest, daß die Steps sequentiell ablaufen müssen. Von jedem dieser Steps ist außerdem bekannt, welche Kontextelemente von ihm geschrieben werden, so daß für die gesamte Step-Sequenz zu jedem Zeitpunkt bekannt ist, welche Kontextelemente erzeugt werden und wieviele Versionen existieren.

Wird von einem der Steps auf die letzte Version eines Kontextelementes zugegriffen, so kann man eindeutig vor Ablauf dieser Sequenz bestimmen, von welchem der Vorgänger des Steps diese Version erzeugt wird, so daß hier Versionsnummern nicht notwendig wären. Diese Möglichkeit existiert auch für die i.-letzte Version eines Kontextelementes.

In einem Skript existieren häufig Step-Sequenzen, deren Steps auf Versionen eines Kontextelementes zugreifen, die von einem Vorgänger in dieser Step-Sequenz erzeugt werden. Dadurch kann man explizit den Step angeben, der diese Version des referenzierten Kontextelementes erzeugt hat. Ein Skript-Compiler hätte damit die Möglichkeit, die Referenzierung zu beschleunigen, indem er die Auswahlmenge durch Angabe des Erzeuger-Steps einschränkt. Bei der verteilten Ausführung auf mehreren Rechnern könnte außerdem die Kontextmenge eingrenzt werden, die übertragen werden muß, um die Teile des Skripts auf dem anderen Rechner auszuführen.

5.4.2 Bedingte Verzweigung

Bei der bedingten Verzweigung, der **if**-Anweisung, wird anhand einer zu überprüfenden Bedingung einer von zwei Ausführungszweigen ausgewählt. Betrachtet man den Skriptabschnitt

S_0
if (Bedingung**)**
    S_1
else
    S_2
endif
S_3

so ist eindeutig festgelegt, daß S_0 vor S_1, S_2, S_3 und S_3 nach S_0, S_1 oder S_2 ausgeführt wird. Es ist jedoch im voraus nicht bekannt, welcher der beiden Steps S_1 und S_2 tatsächlich abläuft.

Jeder der vier Steps kann auf die zuletzt erzeugte Version eines Kontextelementes zugreifen, indem diejenige Version mit der höchsten Versionsnummer ausgewählt wird. Gibt man jedoch bei der Referenzierung eines Kontextelementes im Step S_3 (oder einem nachfolgenden Step) die einschränkende Eigenschaft an, daß nur Versionen ausgewählt werden, die Step S_1 in der **if**-Anweisung erzeugt hat, so ist die Existenz dieser Version nicht sicher, weil hier auch der **else**-Zweig gewählt werden könnte. Analog gilt dies für Step S_2.

Eine Lösungsmöglichkeit wäre das Verbot dieser Referenzierung durch den Skript-Compiler. Dazu muß er diese Referenzierungsmöglichkeit als unsicher erkennen und gegebenenfalls eine Warnung an den Skriptprogrammierer ausgeben oder das Skript als fehlerhaft zurückweisen. Als weitere Möglichkeit könnte auch die fehlende Version des Kontextelementes durch den Skriptprogrammierer abgefragt und entsprechend darauf reagiert werden. In diesem Fall verlegt man die Verantwortung für die Existenz einer Kontextelement-Version vom Compiler auf den Programmierer.

5.4.3 Schleife

Die Steps innerhalb einer **while**-Schleife werden wiederholt, solange die Bedingung erfüllt ist.

Analog zur bedingten Verzweigung kann man durch die sequentielle Ausführung der Steps sehr einfach die letzte Version eines Kontextelementes bestimmen. Es tritt auch hier das Problem auf, daß ein nachfolgender Step nicht Versionen, die ein Step innerhalb der **while**-Schleife erzeugt hat, referenzieren kann, da die Steps innerhalb der Schleife nicht unbedingt abgelaufen sein müssen. Dies ist der Fall, wenn die Bedingung beim Eintritt in die Schleife nicht erfüllt ist.

Im Gegensatz zur bedingten Verzweigung, bei der die Steps höchstens einmal ablaufen, können Steps innerhalb der Schleife mehr als einmal ablaufen. D.h., ein Kontextelement, das einer dieser Steps erzeugt, kann eine vor Ablauf nicht bekannte Anzahl an Versionen besitzen. Daher kann das System bei der Referenzierung der i.-letzten Version eines Kontextelementes von einem Step in oder nach einer Schleife nicht eindeutig den Step, der diese Version erzeugt, bestimmen.

Als Abhilfe sollte daher die gewünschte Version des Kontextelementes durch explizite Angabe des Step-Namens, der diese Version erzeugt hat, referenziert werden. Der ConTract aus Abbildung 2.1 verwendet dies zur Referenzierung einer älterer Version eines Kontextelementes im Step `Receipt`.

5.5 Referenzierung bei paralleler Ausführung

Die Skriptsprache des Prototypen bietet dem Benutzer zwei Kontrollflußanweisungen an, die die parallele Ausführung von Steps erlauben. Diese Anweisungen unterscheiden sich durch den Zeitpunkt, an dem der Parallelitätsgrad entschieden wird, voneinander. Die **par**-Anweisung legt schon zum Zeitpunkt der Skripterstellung fest, welche Steps parallel ablaufen können, während bei der **parforeach**-Anweisung erst zur Laufzeit entschieden wird, wie oft ein Step parallel ablaufen darf.

Im Skriptausschnitt

S_0
par
  S_1
  S_2
endpar
S_3

läuft zuerst Step S_0, anschließend können S_1 und S_2 parallel zueinander ablaufen und erst nach Beendigung von S_1 und S_2 kann mit S_3 fortgefahren werden. Es ist damit schon vor Ablauf des ConTracts bekannt, welche Steps parallel ablaufen können und wie groß der maximale Parallelitätsgrad ist.

Dagegen wird bei **parforeach** der Parallelitätsgrad durch die Mächtigkeit einer Menge bestimmt, die man als Parameter der Anweisung angibt. Im Skriptausschnitt

S_0
parforeach (i **in** $[1..n]$)
  S_1
  S_2
endeach
S_3

wird die Step-Folge S_1 und S_2 für jedes Element der Menge $[1..n]$ sequentiell ausgeführt. Dabei können diese n Step-Sequenzen parallel ablaufen. Ein Anwendungsbeispiel für **parforeach** wurde im Skript in Abbildung 2.11 vorgestellt.

Neben der variablen Parallelität wird hier ein weiterer Unterschied zur **par**-Anweisung deutlich: Während bei **par** die parallel ablaufenden Steps sich zumindest durch die Step-Nummer unterscheiden, werden hier in jedem parallelen Zweig die gleichen Steps ausgeführt. Damit kann jedoch die zur Unterscheidung eingeführte, für jeden Step eindeutige Step-Nummer nicht zur Unterscheidung dieser parallel ablaufenden Steps verwendet werden. Dasselbe Problem ergibt sich auch bei einer sequentiellen Schleife, wo Steps mit derselben Step-Nummer mehrmals ausgeführt werden. Da im Unterschied dazu die Steps

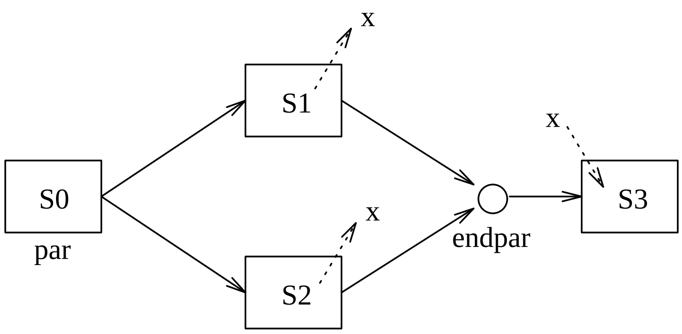

Abbildung 5.2: Referenzierung von x nach einer **par**-Anweisung

bei paralleler Ausführung gleichzeitig ablaufen können, muß ein weiteres Unterscheidungsmerkmal eingeführt werden. Bevor hierauf näher eingegangen wird, sollen noch mögliche Probleme, die durch die **par**-Anweisung auftreten können, betrachtet werden.

5.5.1 Ablauf mit einem festen Parallelitätsgrad

In Abbildung 5.2 ist der Ausführungsgraph des Skriptausschnittes

S_0
par
    S_1 (out: x)
    S_2 (out: x)
endpar
S_3 (in: x)

dargestellt. Dabei schreiben die parallel ablaufenden Steps S_1 und S_2 jeweils eine neue Version eines Kontextelementes x. Der Step S_3 liest danach die zuletzt erzeugte Version dieses Kontextelementes x. Durch die Parallelität ist selbst *nach* Ablauf von S_1 und S_2 nicht festgelegt, welche der beiden erzeugten Versionen vom Step S_3 gelesen wird, da das Skript eine beliebige Ausführungsreihenfolge von S_1 und S_2 zuläßt. Der Skript-Compiler sollte deshalb bei dieser Referenzierung eine Warnung an den Stepprogrammierer ausgeben, um sicherzustellen, daß diese nicht deterministische Referenzierung gewollt ist.

Gibt man im obigen Falle bei der Referenzierung den Step an, der die Version von x erzeugt hat, erhält man eine eindeutig bestimmte Version.

Abbildung 5.3 zeigt den zum folgenden Skriptausschnitt gehörenden Ablaufgraph.

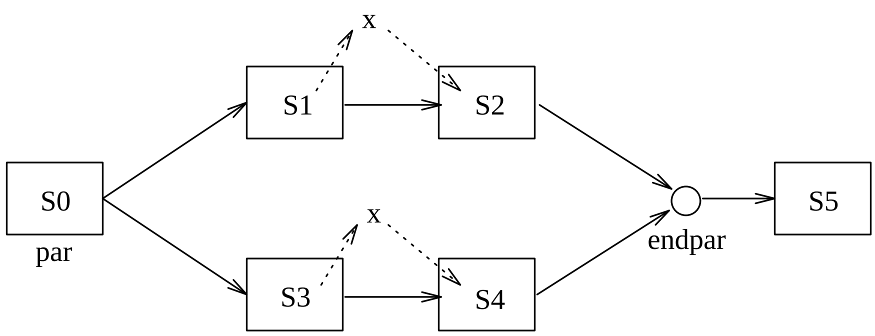

Abbildung 5.3: Referenzierung von x innerhalb einer **par**-Anweisung

S_0
par
  seq
    S_1 (out: x)
    S_2 (in: x)
  endseq
  seq
    S_3 (out: x)
    S_4 (in: x)
  endseq
endpar
S_5

Die Steps S_1 und S_3 erzeugen jeweils eine neue Version von x. Nach Step S_1 läuft S_2 und nach S_3 S_4. Dabei soll S_2 die von S_1 und S_4 die von S_3 zuletzt erzeugte Version von x lesen. Durch die mögliche Parallelität ist dies jedoch nicht eindeutig, da die tatsächliche Ablaufreihenfolge die gelesene Version bestimmen würde. Z.B. liest Step S_2 bei der Ausführungsfolge S_1,S_2,S_3,S_4 die von S_1 erzeugte Version, während bei der Folge S_1,S_3,S_2,S_4 die von S_3 erzeugte Version gelesen würde.

Eine sehr einfache und naheliegende Lösung wäre die explizite Angabe des Steps, der die Version erzeugt hat. Als weitere Lösung bietet sich die Einführung eines weiteren Kontextelementes mit anderem Namen an, das von S_3 anstelle von x geschrieben und von S_4 gelesen wird. Durch die Verwendung unterschiedlicher Kontextelementnamen kann man eindeutig die zuletzt geschriebene Version referenzieren.

5.5.2 Ablauf mit einem variablen Parallelitätsgrad

Kann man keine der beiden Lösungsmöglichkeiten anwenden, so ist die eindeutige Referenzierung nicht möglich und man müßte in diesem Fall auf die Möglichkeit der parallelen Ausführung verzichten.

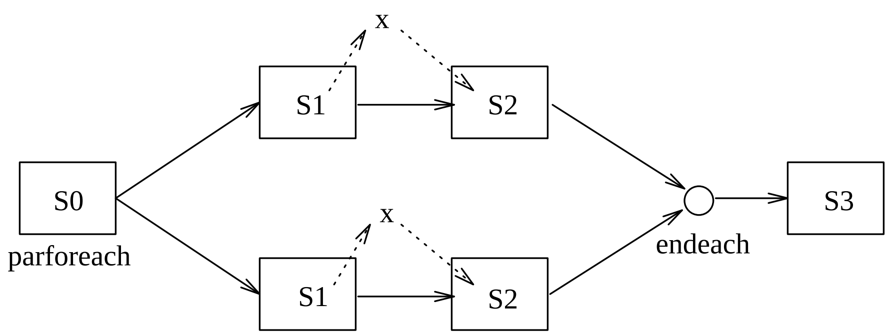

Abbildung 5.4: Referenzierung von x innerhalb einer **parforeach**-Anweisung

Betrachtet man den folgenden Skriptteil, einem vereinfachten Ausschnitt aus dem in Abbildung 2.11 vorgestellten Beispiel-Skript, so kann man erkennen, daß man keine der beiden oben genannten Lösungsmöglichkeiten bei der **parforeach**-Anweisung anwenden kann. Der Ablaufgraph in Abbildung 5.4 sollte dies noch einmal verdeutlichen.

S_0
parforeach (i **in** $[1..n]$)
    S_1 (out: x)
    S_2 (in: x)
endeach
S_3

Dies resultiert aus der Tatsache, daß die gleichen Steps in jedem parallelen Zweig ausgeführt werden und damit die Step-Nummer zur Unterscheidung der Steps nicht mehr ausreichend ist.

Jeder Step, der parallel zu anderen Steps ausgeführt wird, erhält deshalb neben der Step-Nummer ein zusätzliches Attribut, den sogenannten *Index* des Steps. Zu Beginn der parallelen Verzweigung werden die einzelnen Ausführungszweige durchnumeriert und diese Nummer, der Index eines parallelen Steps, kann dann zur eindeutigen Referenzierung einer Version eines Kontextelementes mitverwendet werden.

Mit Hilfe dieses Indexes muß nun die Referenzierung der letzten Version von x durch den parallel ablaufenden Step S_2 so geschehen, daß die in den anderen parallel ablaufenden Zweigen von S_1 erzeugten Versionen nicht sichtbar sind. Dies erreicht man durch logische Isolation der einzelnen parallel ablaufenden Sequenzen, wie in Abbildung 5.5 schematisch dargestellt.

Durch diese Isolation ergeben sich für jeden parallelen Ausführungszweig eine sequentielle Ausführungsfolge der Steps. Als erster Ansatz, um eine Version eines Kontextelementes zu referenzieren, bietet sich folgende Vorgehensweise an.

Zu jeder Version eines Kontextelementes wird der Index des diese Version erzeugenden

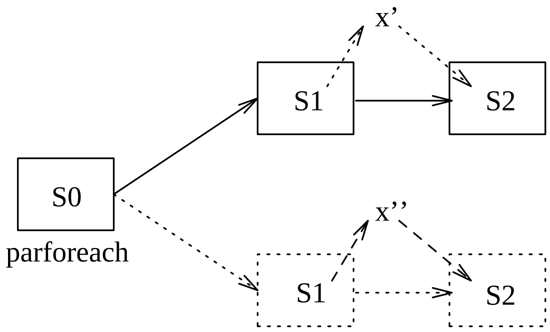

Abbildung 5.5: Isolation der referenzierbaren Kontextelementversionen parallel ablaufender Step-Sequenzen

Steps mitabgelegt. Will man nun aus einem Step, der einen Index besitzt und damit parallel zu anderen Steps abläuft, ein Kontextelement referenzieren, so betrachtet man zuerst nur diejenigen Versionen des Kontextelementes, die den gleichen Index besitzen. Existieren keine solche Versionen, so wird der Zugriff ohne Einschränkung auf alle Versionen des Kontextelementes noch einmal durchgeführt. Im ersten Fall erhält man eine Version eines Vorgängers desselben parallel ablaufenden Zweiges, im zweiten Fall ist die Version von einem Step erzeugt worden, der vor der parallelen Verzweigung ausgeführt wurde.

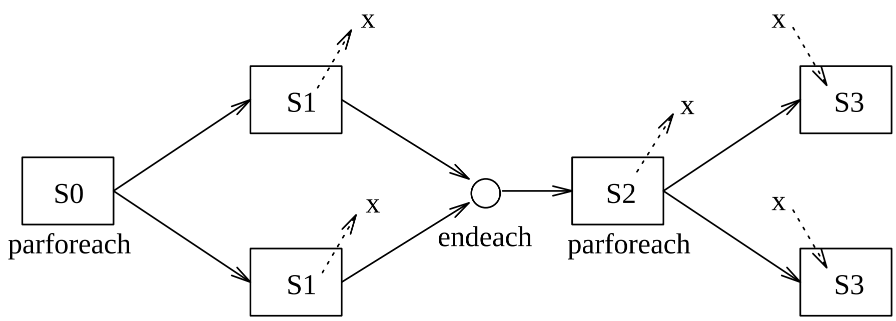

Abbildung 5.6: Korrekte Referenzierung der zuletzt erzeugten Version von x

Dieser Ansatz liefert in einigen Situationen nicht die zuletzt erzeugte Version eines Kontextelementes. Geht man in dem in Abbildung 5.6 gezeigten Skriptabschnitt wie beschrieben vor und referenziert in Step S_3 die zuletzt erzeugte Version von x, so erhält man nicht die von S_2, sondern die von S_1 erzeugte Version von x. Da man zuerst ausschließlich nach Versionen mit gleichem Index sucht, kann gerade dieser Fall eintreten.

Um dies zu vermeiden, sucht man gleichzeitig nach Versionen mit dem gleichen Index und nach Versionen, die keinen Index besitzen. Die Version, die nun die größere Versionsnummer aufweist, ist die korrekte, aus Sicht dieses Steps zuletzt erzeugte Version. In obigem Beispiel erhielte man damit die beiden Versionen von x, die die Steps S_1 und S_2 erzeugt haben, würde aus diesen beiden dann die mit der höchsten Versionsnummer auswählen und damit die Version referenzieren, die S_2 erzeugte.

Jedoch kann immer noch nicht auf die vollständige Wertehistorie des Kontextelementes x zugegriffen werde, da man alle früheren Versionen ausschließt, die einen Index besitzen, aber nicht den gleichen Index aufweisen. Der oben angegebene Referenzierungsmechanismus war in den bisher betrachteten ConTracts ausreichend und ist darüberhinaus eine leicht zu implementierende Näherung der gewünschten Referenzierung.

5.6 Vergabe von Versionsnummern

Jede Version eines Kontextelementes erhält eine Versionsnummer, die die Aufgabe hat einerseits die einzelnen Versionen zu unterscheiden, andererseits die Wertehistorie des Kontextelementes zu dokumentieren. Während die Vergabe einer fortlaufenden Nummer bei einem sequentiellen Ablauf der Steps sehr einfach ist, ergeben sich bei Zulassung von Parallelität einige Probleme, die z.B. durch die notwendige Synchronisierung bei der Vergabe einer globalen Versionsnummer verursacht werden.

Zur Realisierung einer Versionsbezeichnung bieten sich die folgenden Möglichkeiten innerhalb eines ConTracts an:

1. Step-Nummer, Step-Version und Step-Index als Versionsnummer,
2. fortlaufende lückenlose Numerierung,
3. fortlaufende lückenlose Numerierung unter Hinzunahme des Step-Indexes,
4. fortlaufende Numerierung mit Lücken.

Im ersten Fall benötigt man für die Versionsnummer eines Kontextelementes keine gesonderte Numerierung. Vom Skript-Compiler und -Interpreter werden alle Steps eines Skriptes mit einer Step-Nummer versehen. Zusätzlich zu dieser Nummer besitzt jeder Step zur Laufzeit noch eine Step-Version und einen Step-Index, um innerhalb von Schleifen und **parforeach**-Anweisungen mehrfache Aktivierungen eines Steps zu unterscheiden. Diese Numerierung wird bei der Ausführung vom ConTract-System benutzt, so daß man sie ohne weiteren Aufwand verwenden könnte.
Werden die Step-Nummern ohne Ordnung vergeben, so kann man ohne Kenntnis des Skripts nicht auf eine bestimmte Version eines Kontextelementes zugreifen. Könnte man die Steps so durchnumerieren, daß jede mögliche Ausführungsreihenfolge der Steps dieser Ordnung folgt, so wäre eine Vergabe der Versionsnummern eines Kontextelementes nach dieser Ordnung realisierbar. Durch Schleifen oder **parforeach**-Anweisungen kann man die Steps jedoch nicht nach Step-Nummer, -Version und -Index anordnen, da die Anzahl der Schleifendurchläufe bzw. der Parallelitätsgrad erst zur Laufzeit bekannt ist. Somit benötigt jede Referenzierung bei dieser Realisierung einer Versionsnummer zusätzliche Informationen über die Reihenfolge der Steps aus dem Skript.

Im zweiten Fall wird jedes Kontextelement mit einem Versionszähler versehen, der die bisher höchste vergebene Versionsnummer beinhaltet. Bei der parallelen Ausführung von

Steps kann ein mehrfacher Schreibzugriff auf diesen Zähler erfolgen. Werden Sperren als Synchronisationsmittel verwendet, so muß diese Sperre bis zur Beendigung des sperrenden Steps gehalten werden. Da jedes Kontextelement einen Versionszähler besitzt und ein Step mehr als ein Kontextelement schreiben kann, können durch die Möglichkeit des wechselseitigen Ausschlusses Deadlocks auftreten. Dies kann durch Verwenden des ESCROW-Sperrmechanismus [ONe86] oder der Freigabe der Sperre sofort nach der Änderung verhindert werden.
Gibt man die Sperre vor Ende des Steps frei und setzt den Zähler nicht zurück, so wird bei einem Abbruch des Steps die verwendete Versionsnummer nicht an eine Version des Kontextelementes vergeben. Um diese Lücken in der Vergabe zu vermeiden, müßte man diese Versionsnummern erneut vergeben, dazu müßte das System jedoch zusätzlich zum Versionszähler eine Liste nicht vergebener Versionsnummern verwalten.

Im dritten Fall wird die Versionsnummer durch den Stepindex ergänzt, um die Synchronisierung paralleler Zugriffe bei der Vergabe der Versionsnummer zu vermeiden. Dadurch benötigt jedoch jeder Zweig bei einer parallelen Ausführung einen eigenen Versionszähler, dessen Einrichtung dem Skript-Interpreter unterliegen muß. Durch die Vergabe der Versionsnummer in Abhängigkeit des Step-Indexes existiert zwischen den Versionen eines Kontextelementes keine Ordnungsbeziehung, so daß die Referenzierung des i-ten bzw. der i.-letzten Version nicht eindeutig ist. Dies kann man vermeiden, indem man bei der Zusammenführung der parallelen Zweige alle Versionen neu numeriert, die von den parallel abgelaufenen Steps erzeugt wurden.

Im letzten Fall werden, analog zum zweiten Fall, Versionsnummern für jedes Kontextelement global vergeben. Bei der Vergabe der Nummern sind Lücken zugelassen, so daß man eine bestimmte Version eines Kontextelementes nicht direkt über die Versionsnummer referenzieren sollte, da die Version des Kontextelementes mit dieser Versionsnummer nicht existieren muß. Soll auf die i-te bzw. die i.-letzte Version zugegriffen werden, so kann man dies durch sequentielles auf- bzw. absteigenden Durchsuchen der Versionsmenge realisieren. Bei der Vergabe einer neuen Versionsnummer wird nur während der Dauer der Erhöhung und Abfrage des Versionszählers eine Sperre gesetzt. Die aufsteigende Numerierung der Versionen eines Kontextelementes wird dabei durch die Ausführungsreihenfolge der Steps erreicht. Damit entspricht die Wertehistorie eines Kontextelementes der Ausführungshistorie der Steps, die dieses Kontextelement verändern.

Zur Realisierung bieten sich eine der beiden letztgenannten Möglichkeiten an. Beide haben die Eigenschaft, daß keine unnötige Einschränkung der Parallelität erfolgt. Bei der dritten Möglichkeit, Versionsnummer und Step-Index, ist bei verteilter Ausführung durch den für jeden parallelen Zweig vorhandenen eigenen Versionszähler kein Nachrichtentransfer über Rechnergrenzen erforderlich. Jedoch benötigt die Neunumerierung bei Zusammenführung der parallelen Zweige zusätzlichen Aufwand. Im Vergleich dazu ist dies bei der letzten Möglichkeit nicht notwendig, jedoch ist bei verteilter Ausführung durch die Verwendung eines einzigen Versionszählers zur Vergabe einer Versionsnummer Nachrichtentransfer erforderlich.

Kapitel 6

Maßnahmen zur Erhöhung der Ausfallsicherheit

Die Ausfallsicherheit und das sichere, schnelle und korrekte Fortfahren im Fehlerfall sind für ein System, dessen Zielsetzung in einer verteilten, parallelen Verarbeitung liegt, von erheblicher Bedeutung. Die in diesem Abschnitt angesprochene Problematik der Ausfallsicherheit bezieht sich allerdings auf die Ausfallsicherheit der Kontextverwaltung und der darunterliegenden Datenbank. Das Konzept der logischen Kompensation einer Aktion, um einen Ablauf in jedem Fall zu einem definierten Endzustand zu bringen, bleibt unberücksichtigt.

Die vorhandenen Ebenen der Eingriffsmöglichkeiten sind:

1. Hardware.
   Längere MTBF[1] wird durch entsprechende Konzeption der Hardware und durch Zusatzeinrichtungen erreicht.

2. Betriebssystem.
   Das Betriebsystem stellt eine Funktionalität zu Verfügung, die zur Erhöhung der Ausfallsicherheit beiträgt.

3. Datenbanksystem.
   Das eigentliche Datenbanksystem stellt einerseits bereits heute die meisten Mechanismen zur Verfügung, z.B. durch geeignete Recovery–Verfahren und die verteilte Datenhaltung, andererseits ergeben sich durch die Zusammenarbeit mit der Kontextverwaltung erweiterte Möglichkeiten.

4. Kontextverwaltung.
   Die Kontextverwaltung kann auf den Gesichtspunkt der Ausfallsicherheit hin entworfen werden. Hinzu kommt die angesprochene Zusammenarbeit mit dem DBMS.

[1] Meantime between Failure

6.1 Hardware

Bewährt haben sich in diesem Bereich bereits die Konzepte der Duplizierung der Speichermedien (*Spiegelplatten*[2]), Sicherung des dynamischen Speichers durch Batteriepufferung gegen Stromausfall oder Mehrfachauslegung wichtiger Hardwarekomponenten (z.B. Controller der Festplatte)[3].

Neuere Entwicklungen auf dem Hardwaresektor ermöglichen z.B. folgende Konzepte:

- Kommunikationshardware.
  Komponenten, die automatisch Verbindungen kontrollieren, Ersatzwege schaffen und im Fall der Überlastung zusätzliche Wege aufbauen, werden zunehmend eingesetzt. Dadurch wird es möglich, die Aufgaben der Überwachung und Ersatzbeschaffung zu erleichtern und diese dem ConTract–System zu übertragen. Dem Gesichtspunkt der sicheren, verteilten Verarbeitung wird damit Rechnung getragen.

- Zur Sicherung der Log–Informationen des Datenbanksystems und der Kontextverwaltung (dazu genaueres später) kann ein WORM–Datenträger, i.allg. eine optische Speicherplatte, eingesetzt werden. Diese Platten können, einmal beschrieben, nicht gelöscht werden und durch die berührungslose Abtastung ist die Wahrscheinlichkeit eines Ausfall signifikant geringer als bei einer herkömmlichen Festplatte.

6.2 Betriebssystem

Die Möglichkeiten des Betriebssystems beruhen i.allg. auf der Unterstützung der vorhandenen Hardwaremöglichkeiten. Die Verwaltung der bereits erwähnten Kommunikationsverbindungen kann vom Betriebssystem übernommen oder unterstützt werden. So könnten logische Kommunikationskanäle bereitgestellt werden, die in Zusammenarbeit mit entsprechender Hardware überwacht und verwaltet werden[4]. Andere Gesichtspunkte sind die Umkonfigurierbarkeit des Systems im laufenden Betrieb oder die Verlagerung von Prozessen auf andere Rechner oder Knoten, ohne daß der Prozeß und andere davon abhängige Applikationen dies feststellen.

Bei den Möglichkeiten, die auf der Betriebssystemebene anzusiedeln sind, ist stets zu beachten, daß eine Erweiterung des Betriebssystems nur in den seltensten Fällen möglich ist. Ein Beispiel hierfür ist der Markt der **UNIX**-Systeme. Dieser ist heftig umkämpft und durch die ohnehin divergierenden Bemühungen, einen gemeinsamen Standard zu definieren, wäre

[2] Der Einsatz gespiegelter Platten ist i.allg. nicht ohne entsprechende Zusammenarbeit mit dem DBMS möglich. Eine Einstufung in die Ebene Hardware erschien allerdings naheliegender als das Konzept bei 'Datenbanksystemen' einzuordnen.

[3] Die aufgezählten Möglichkeiten sind alle im Rechner der Firma **TANDEM** des IPVR zum Einsatz gekommen. Nähere Informationen hierüber sind den **TANDEM**–Unterlagen zu entnehmen.

[4] Als Beispiel für diese Zusammenarbeit dient das *NFS (Network File System)* der Firma SUN–Microsystems sein. Hier werden dem Benutzer durch Anbindung an ein lokales Netz Dateisysteme zugänglich gemacht, die nicht lokal auf seinem Rechner sind. Der Benutzer braucht darauf aber keine Rücksicht zu nehmen.

es sinnlos, eine Erweiterung des Betriebssystems ins Auge zu fassen. Eine Insellösung und der Untergang in der Masse sind sicherlich die Folge. Anders formuliert: Eine echte Erweiterung des Betriebssystems ist unrealistisch, es kann höchstens auf vorhandene Mechanismen zurückgegriffen bzw. aufgesetzt werden.

6.3 Datenbanksystem

Die interessanteste Möglichkeit der Verteilung und Parallelisierung der Verarbeitung bietet sicherlich die Ebene der Datenbank. Zusammen mit dem Gesichtspunkt der Ausfallsicherheit ergeben sich in existierenden, kommerziellen Systemen u.a. folgende Aspekte:

- DBMS erlauben es zunehmend, Relationen horizontal und vertikal zu zerlegen. Horizontal bedeutet, daß einzelne Tupel einer Relation auf verschiedenen physikalischen Speichermedien oder Dateien gespeichert werden. Vertikale Zerlegung bedeutet, daß die Aufteilung bezüglich der Attribute einer Tabelle erfolgen kann[5]. Für die Kontextverwaltung im Rahmen des ConTract–Systemes ist die horizontale Zerlegung von größerer Bedeutung. Wird der Kontext so verwaltet, daß Tabellen für jeden Kontexttyp erzeugt werden, so kann dies auf verschiedenen Speichermedien geschehen. Im Fehlerfall müßen dann nicht alle Daten wiederhergestellt werden, sondern nur einzelne Teile davon. Entsprechendes gilt bei horizontaler bzw. vertikaler Zerlegung der Relation(en).

- Ein weiterführendes Konzept ist die Integration einzelner Datenbanken[6]. Für die Kontextverwaltung ist hier der Gesichtspunkt der Partitionierung sogar über Netze hinweg interessant.

- Ein bereits bekanntes Konzept ist die Sicherung des Datenbestandes im laufenden Betrieb. Bei sinnvoller Nutzung wird die Zeit der Wiederherstellung (*Recovery*) der Daten verkürzt und somit eine Fortführung des normalen Betriebs und damit auch der ConTracts schneller möglich.

Folgende Ansätze sind nicht überall realisiert:

- Das Log einer Datenbank kann an eine andere Datenbank, i.allg. auf einem anderen Rechner, weitergegeben und dort 'nachgefahren' werden. Dieses Nachfahren kann synchron oder asynchron erfolgen, dies hängt von der einzelnen Anwendung ab. Das Ergebnis ist eine *Heiße Reserve*. Im Fehlerfall kann die 'Schattendatenbank' die Aufgaben des Hauptsystems übernehmen.

- Ähnlich der *heißen Reserve* kann ein DBMS prinzipiell Änderungsanweisungen (zum Beispiel `INSERT`–, `UPDATE`– oder `DELETE`–Statements in **SQL**) mehrfach auf getrenn-

[5] Eine mögliche vertikale Zerlegung ist z.B. , die ersten 5 Attribute auf einer Festplatte, die restlichen auf einer anderen zu verwalten.

[6] Mit Hilfe der *Gateway*–Technik ist sogar die Integration unterschiedlicher Datenbanken verschiedener Datenbankhersteller möglich. Beispiel hierfür sind weiterreichende Möglichkeiten, die von der Firma **INGRES** bzw. von ihrem gleichnamigen DBMS angeboten werden.

ten Datenbeständen ausführen. D.h. das DBMS verwaltet Relationen mehrfach, sinnvollerweise im Zusammenhang mit einer Speicherung der Tabellen auf verschiedenen Medien.

6.4 Kontextverwaltung

Die Möglichkeiten der Kontextverwaltung stellen meist eine Fortführung der Recovery–Konzepte des DBMS auf logisch höherer Ebene dar. Allerdings soll nicht vergessen werden, daß die Wahl des Schemas auch Einfluß auf die Ausfallsicherheit hat. Werden Relationen für jeden Kontexttyp und zusätzlich noch für jeden ConTract erzeugt, kann eine weitgestreute Verteilung der Tabellen über mehrere Rechner und viele Platten erfolgen. Dies hat zur Folge, daß das Gesamtsystem bei Ausfall eines Rechners/einer Festplatte nicht zwangsweise komplett zum Erliegen kommt.

Es folgen Beispiele für Möglichkeiten der Kontextverwaltung:

- Die Kontextverwaltung kann ein 'logging' der Befehle durchführen, welche den Datenbestand bzw. die Struktur der Datenbank verändern. D.h. , die ausgeführten SQL–Statements werden protokolliert, um im Fehlerfall das Recovery zu unterstützen.
- Der Mechanimus kann dahingehend erweitert werden, daß die Protokolle zeitnah (asychron) oder synchron parallel ausgeführt werden[7].
- Ein weiteres, ebenfalls aus der Datenbanktechnik bekanntes Instrument, ist der Sicherungspunkt. Dieses Konzept kann auf die Kontextverwaltung übertragen werden. So kann zu ausgezeichneten oder zufällig gewählten Zeitpunkten eine Synchronsisation mit einem anderen Datenbestand stattfinden. Dies kann bedeuten, daß Relationen dupliziert werden (logisch, mit Hilfe von der Datenmanipulationssprache oder physikalisch, durch Kopieren von Dateien), oder Protokolle bzw. Logs der Datenbank 'nachgefahren' werden.

[7] Probleme, die durch die Transaktion als Ausführungeinheit entstehen, sind geeignet zu berücksichtigen. Denn besteht der Bedarf der Synchronisation, dies ist bei ausschließlicher Sicherung von Kontextelementen nicht notwendig der Fall, sei auf das *Zwei–Phasen Commit Protokoll* verwiesen (siehe hierzu u.a. [GW90] Seite 603 ff.).

Kapitel 7

Vergleich mit anderen Ansätzen

In einem ConTract wird ein langer Vorgang in einzelne Verarbeitungschritte zerlegt, deren Ausführung das Skript durch Kontrollstrukturen miteinander verbindet. Durch diese Trennung der Verarbeitung und des Kontrollflusses kann das ConTract-System diesen Vorgang nach einem Fehler automatisch an der richtigen Stelle fortsetzen. Dazu muß der Verarbeitungszustand des Vorgangs im Kontext gesichert werden.

Der hier vorgestellte Ansatz der Kontextverwaltung versucht dabei, eine einfache, anwendungsneutrale Schnittstelle für Zugriffe auf den Aktionskontext zur Verfügung zu stellen.

TP-Monitore [MW88] ermöglichen dem Anwender die Verkettung mehrerer Transaktionsprogramme durch entsprechende Kontrollmechanismen und explizite Kommunikationsbereiche mit unterschiedlicher Reichweite und Lebensdauer zwischen den Transaktionen.

Unter diesen Kommunikationsbereichen sind das *Vorgangsgedächtnis* und der *lokale sekundäre Speicherbereich* vergleichbar mit dem Kontext eines ConTracts, unterscheiden sich jedoch in wesentlichen Punkten von ihm:

- Das Vorgangsgedächtnis ist unstrukturiert, d.h. das Layout und der Zugriff darauf müssen zwischen den einzelnen Transaktionsprogrammen abgestimmt werden. Die Verwaltung ist somit anwendungsabhängig.
- Das Vorgangsgedächtnis befindet sich im flüchtigen Hauptspeicher. Die Sicherung dieses Speicherbereichs am Ende einer Transaktion liegt in der Zuständigkeit des Transaktionsprogrammes.
- Der lokale sekundäre Speicherbereich ist ebenso unstrukturiert. Im Gegensatz zum Vorgangsgedächtnis befindet er sich auf Platte und muß deshalb nicht explizit gesichert werden, jedoch bezieht sich jede Lese- oder Schreiboperation immer auf den gesamten Bereich.
- Da auf die Bereiche nicht parallel zugegriffen werden kann, sind sie nur für die sequentielle Verkettung der Verarbeitungschritte, wie sie vom TP-Monitor angeboten wird, verwendbar.

Parallele und verteilte Programmiersprachen, wie Linda [Lel89, BT88] oder Avalon [CLNW89] besitzen dem Kontext ähnliche Konstrukte, die jedoch für andere Zielsetzungen entworfen wurden.

In Linda können verteilt ablaufende Prozesse gemeinsame Variablenwerte in einen Shared Data-Bereich, den sogenannten *tuple space*, eintragen. Der Zugriff auf Variablenwerte erfolgt dabei assoziativ über Teile des Tupels, vergleichbar dem Zugriff auf Kontextelemente über den Namen und weitere einschränkende Attributeigenschaften. Eine gewisse Fehlertoleranz kann durch die Replikation des *tuple space* auf mehrere Rechner erreicht werden. Da dieser Speicherbereich jedoch nicht auf nichtflüchtige Speicher gerettet wird, ist er ohne weitere Sicherung bei Ausfall aller Rechner nicht mehr verfügbar und kann auch nicht wiederhergestellt werden.

Im Gegensatz zu Linda bietet Avalon die persistente Sicherung einzelner Objekte an, die man im Unterschied zum Kontext für jedes zu sichernde Objekt vom System explizit anfordern muß.

Das **klassische ACID-Transaktionsmodell** kennt keinen Kontext-Begriff, da angenommen wird, daß alle relevanten Zustandsinformationen zwischen zusammenhängenden Transaktionen in der Datenbank stehen und vom Anwender verwaltet werden.

In **Programmiersprachen**, die dieses Modell durch die **Einbettung von Transaktionen** erweitern, wie DBPL [Böt89] oder Avalon, können Zwischenwerte durch lokale Variable der Programmiersprache über Transaktionsgrenzen hinweg übergeben werden. Im Unterschied zum Kontext erfolgt dies jedoch nicht fehlertolerant, so daß bei einem Prozeßabsturz diese Werte verloren gehen. Avalon erlaubt zwar die persistente Sicherung einzelner Variablen, jedoch liegt dies in der Verantwortung des Anwenders.

Neue Aspekte

- Der Kontext wird automatisch vom ConTract-System gerettet und geht dadurch bei einem Systemfehler nicht verloren. Der Vorgang kann deshalb mit dem unterbrochenen Verarbeitungschritt eventuell auch auf einem anderen Rechner fortgesetzt werden.
- Er umfaßt nicht nur den momentanen Ausführungszustand, sondern beinhaltet eine komplette Ausführungshistorie. Durch den Zugriff auf diese Historie können im Fehlerfall einzelne Verarbeitungsschritte kompensiert und mit einem früheren Ausführungszustand fortgefahren werden.
- Der Zugriff auf den Aktionskontext erfolgt über eine definierte Schnittstelle. Mittels eines Bindungsmechanismus für die Namen der Kontextelemente können Steps in mehreren Vorgängen verwendet werden.

Anhang A

Kontextschnittstelle für das Laufzeitsystem

A.1 Typdefinitionen

Für die anschließend beschriebenen Funktionen werden die folgenden Typen definiert.

```
typedef unsigned long  t_cid;      /* ConTract-Id */

typedef unsigned long  t_sid;      /* Step-Nummer  */
typedef unsigned long  t_version;  /* Step-Version */
typedef unsigned long  t_index;    /* Step-Index   */

typedef struct {
    t_sid     id;
    t_version version;
    t_index   index;
} t_step;
```

Der Typ `t_version` ist nur der Vollständigkeit halber mit aufgeführt; er wird für die Referenzierung eines Kontextelementes nicht benötigt. Die Version eines Kontextelementes sollte dabei nicht mit der Version eines Steps verwechselt werden.

A.2 Aufbau der Bindungstabelle

Die in Kapitel 5 beschriebenen notwenigen Bindungs- und Referenzierungsinformationen für den Zugriff auf ein Kontextelement wird in einer Bindungsrelation eingetragen. Die nachfolgenden Funktionen unterstützen mit diesen Informationen den Zugriff auf die zuletzt erzeugte Version oder auf die zuletzt von einem angegebenen Step erzeugte Version eines Kontextelementes.

Für jedes Kontextelement aus dem Ein- bzw. Ausgabekontext eines Steps muß ein Tupel angegeben werden, daß aus den folgenden Attributen besteht.

Attribut	**Bedeutung**
cid	ConTract-Identifier
sname	Step-Nummer
nameinstep	Name, der im Step für dieses Kontextelement verwendet wird
inout	Flag, ob Step Kontextelement liest oder schreibt; gleich 'I', wenn Kontextelement gelesen wird, gleich 'O', wenn es geschrieben wird
ctxtelname	Name des Kontextelementes auf Skriptebene
ctxttyp	Typ des Kontextelementes
prodstep	Erzeugerstep; falls *NULL*-Wert angegeben, keine Einschränkung auf einen Step

Die vier Attribute `cid, sname, nameinstep` und `inout` bilden den Primärschlüssel dieser Relation.

A.3 Schnittstelle für den Interpreter

A.3.1 Lesen eines Kontextelementes

Anforderungen:
Während der Ausführung benötigt der Skript-Interpreter für bedingte Verzweigungen, Schleifen und andere Kontrollflußanweisungen Zugriff auf die Werte eines oder mehrerer Kontextelemente. Dazu wird dem Interpreter eine Funktion `read_ctxt` zur Verfügung gestellt. Im Unterschied zur Funktion `get_context` ist kein Zugriff auf die Bindungstabelle notwendig, da alle Bindungsinformationen aus dem Skript verfügbar sind.

- Für das durch den Namen angegebene Kontextelement wird der Wert der letzten Version zurückgegeben.

- Gibt man einen Erzeuger-Step an, erhält man die letzte Version, die dieser Step geschrieben hat.
- Ist der angegebene Index $\neq 0$, so wird die letzte Version zurückgegeben, deren Index gleich dem angegebenen oder gleich 0 ist. Innerhalb `parforeach` ist der Index $\neq 0$, um sicherzustellen, daß keine Versionen des Kontextelementes referenziert werden, die ein parallel ablaufender Step erzeugt hat.

Parameter:

- ConTract-Id
- Step-Index
- Kontextelement-Name und -Typ
- Erzeuger-Step (optional)

Spezifikation:

```
int read_ctxt ( void   *ctxt_wert,   /* out */
                t_cid   cid,
                t_index index
                char   *ctxt_elname,
                char   *ctxt_typ,
                t_sid   producer_step,
                int     nullflag )
```

Parameter	**Bedeutung**
ctxt_wert	Rückgabeparameter: Zeiger auf den Wert des Kontextelementes
cid	ConTract-Identifier
index	Index; ungleich 0, wenn Zugriff innerhalb eines `parforeach` erfolgt
ctxt_elname	Name des Kontextelementes
ctxt_typ	Typ des Kontextelementes
nullflag	Flag; gleich TRUE, wenn **kein** Erzeuger-Step angegeben wird
producer_step	Erzeuger-Step der gesuchten Version des Kontextelementes, falls 'nullflag' gleich 'FALSE'

Der Parameter `ctxt_wert` ist ein Zeiger, der auf einen vom Aufrufer der Funktion zur Verfügung zu stellenden Speicherplatz zeigt. Es wird *nicht* überprüft, ob dieser Speicherplatz zur Aufnahme des Wertes ausreicht. Es wird außerdem nicht die Typkompatibilität dieses Parameters überprüft.

A.3.2 Abspeichern eines Kontextelementes

Anforderungen:

Verschiedene Kontrollflußanweisungen des Skripts schreiben Kontextelemente (z.B. foreach, parforeach). Dazu muß der Interpreter neue Versionen eines Kontextelementes erzeugen können.

Der neue Wert wird mit einer neuen Versionsnummer versehen und unter dem angegebenen Kontextelementnamen abgelegt. Dabei ist die neue Versionsnummer die um eins erhöhte, höchste vergebene Versionsnummer dieses Kontextelementes. Als weitere Attribute erhält diese Version die angegebene Step-Nummer und den Step-Index. Beide Attribute werden der Funktion in der Struktur `t_step` übergeben.

Spezifikation:

```
int save_index_ctxt ( void   *ctxt_newvalue,
                      t_cid   cid,
                      t_step *step_id,
                      char   *ctxt_elname,
                      char   *ctxt_typ )
```

Parameter	**Bedeutung**
ctxt_newvalue	Zeiger auf den Wert der neuen Version dieses Kontextelementes
cid	ConTract-Identifier
step_id	Zeiger auf Step-Struktur, in dem die Komponenten `id` (Step-Nummer) und `index` (Step-Index) auf definierte Werte gesetzt werden müssen; diese Werte werden als Erzeuger-Step dieser Version abgespeichert
ctxt_elname	Name des Kontextelementes
ctxt_typ	Typ des Kontextelementes

Wie beim Lesen eines Kontextelementes findet auch hier keine Überprüfung der Typkompatibilität des Zeigers `ctxt_newvalue` statt, der auf den neuen Wert zeigt.

A.4 Schnittstelle für die Steps

A.4.1 Lesen eines Kontextelementes

Anforderungen:

Zum Lesen eines Kontextelementes muß nur der symbolische Name und der Typ des Kontextelementes angegeben werden. Mit diesem Namen wird in der Bindungstabelle der im Skript verwendete Name nachgeschlagen und mit diesem auf das Kontextelement zugegriffen.

Zum Zugriff auf den Wert werden zusätzlich noch die ConTract-Id, die Step-Nummer und der Step-Index benötigt. Diese Werte existieren im Step-Executor-Prozeß als globale Variablen, auf die der Step-Programmierer keinen Zugriff haben sollte. Mit diesen Werten wird dann (analog zur Funktion `read_ctxt`, siehe Abschnitt A.3.1) der Wert des Kontextelementes aus der Kontextrelation gelesen.

Spezifikation:

```
int get_context ( void *ctxt_value,      /* out */
                  char *name_in_step,
                  char *ctxt_typ )
```

Parameter	Bedeutung
ctxt_value	Rückgabeparameter: Zeiger auf den Wert des Kontextelementes
name_in_step	Name des Kontextelementes im Step
ctxt_typ	Typ des Kontextelementes

A.4.2 Schreiben eines Kontextelementes

Anforderungen:

Analog zum Lesen eines Kontextelementes wird mit dem gegebenen Namen in der Bindungstabelle der im Skript verwendete Name des Kontextelementes nachgeschlagen.

Zum Schreiben des Wertes werden zusätzlich noch die ConTract-Id, die Stepnummer und der Stepindex benötigt. Diese Werte existieren im Step-Executor-Prozeß als globale Variablen. Mit diesen Werten wird dann analog zur Funktion `save_index_ctxt` (siehe Abschnitt A.3.2) der Wert als neue Version des Kontextelementes geschrieben.

Spezifikation:

```
int save_context ( void *ctxt_value,
                   char *name_in_step,
                   char *ctxt_typ )
```

Parameter	**Bedeutung**
ctxt_value	Zeiger auf den Wert der neuen Version dieses Kontextelementes
name_in_step	Name des Kontextelementes im Step
ctxt_typ	Typ des Kontextelementes

Anhang B

Kontextverwaltung und -abfrage im ConTract–Prototypen

Zur Verwaltung der Kontextelemente und zum Erfragen des Kontextes zu einem beliebigen Zeitpunkt sind gewisse Funktionen erforderlich. Die Funktionen, welche im Rahmen der Studienarbeit für diesen Zweck implementiert wurden, werden hier spezifiziert. Unterteilt werden sie in Verwaltungs- und Abfragefunktionen. Die Verwaltungsfunktionen werden für den ConTract–Manager zur Verfügung gestellt, die Abfragefunktionen hingegen dienen zur Überwachung des Ablaufs bzw. dem Nachvollziehen des Ablauf. Sie sind also Basis für einen Monitor gedacht.

B.1 Verwaltungsfunktionen

B.1.1 Löschen des gesamten Kontextes eines ConTracts

Anforderungen:

Alle gespeicherten Kontextelemente eines ConTracts sollen gelöscht werden. Verwendung findet diese Funktion beim Entfernen der Kontextdaten eines beendeten ConTracts. Identifiziert werden die Kontextelemente durch die systemweit eindeutige ConTract–Id.

Spezifikation:

Die C–Deklaration dieser Funktion lautet:

```
int delete_ctxt(cid)
long   cid;
```

Parameter	Bedeutung
cid	ConTract–Identifier

Die Funktion liefert einen Fehlercode zurück. Es werden folgende Codes unterstützt:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.

Das Löschen der Kontextelemente erfolgt im Rahmen einer Transaktion, d.h. auftretende Fehler führen zum Rücksetzen der Löschtransaktion.

B.1.2 Ändern eines Kontextelementes

Anforderungen:

Um einer privilegierten Instanz den steuernden Zugriff auf einzelne Kontextelemete zu ermöglichen, gibt es eine Funktion, mit der die Werte einzelner Kontextelemente geändert werden können. Die Funktion erlaubt nur das Ändern von Kontextelementen. Das Erzeugen von neuen Kontextelementen oder neuen Versionen ist nicht gestattet.

Das Kontextelement, das geändert werden soll, muß eindeutig identifiziert werden. Hierzu werden folgende Angaben benötigt:

1. Systemweit eindeutige ConTract–Id.
2. Name des Kontextelementes.
3. Versionsnummer des Kontextelementes
4. Index des Kontextelementes

Spezifikation:

Die Deklaration dieser Funktion für die Programmiersprache C lautet:

```
int modify_ctxt(cid, ctxt_elname, ctxt_version, ctxt_index, ctxt_newvalue)
long   cid;
char   *ctxt_elname;
long   ctxt_version;
long   ctxt_index;
char   *ctxt_newvalue;
```

Parameter	Bedeutung
cid	ConTract–Identifier
ctxt_elname	Name des Kontextelementes
ctxt_version	Version des Kontextelementes
ctxt_index	Index des Kontextelementes
ctxt_newvalue	Zeiger auf den neuen Wert des Kontextelementes

Die Fehlercodes dieser Funktion haben die Bedeutung:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_NOCTXTELFOUND	Angaben selektieren kein(e) Kontextelement(e).
CT_ERR_TYPINKONSISTENCY	Typangabe in der Kontextbinderelation ist inkonsistent.

Die Funktion erhält einen Zeiger auf den neuen Wert. Der Typ des Wertes muß mit dem entsprechenden Kontextelementtyp bzw. mit der entsprechenden Darstellung in der Programmiersprache übereinstimmen. Eine Typüberprüfung findet <u>nicht</u> statt. Fehler aufgrund eines falschen Typs können zum Absturz des Programms führen.

B.1.3 Erzeugen der Kontextbindung für die ConTract–Ausführung

Anforderungen:

Zum Ablauf eines ConTracts muß eine Kontextbindung vorgenommen werden. Diese Bindung legt fest, welche interne Variable einer Aktion an welches ConTract–weite Kontextelemente gebunden wird. D.h, welcher interne Name welchem globalen Namen entspricht. In der Kontextbindung ist außerdem vermerkt, ob ein Element Aus- oder Eingabekontext ist, von welchem Typ es ist und welche Aktion dieses Kontextelement erzeugt hat. Die Angabe einer Erzeugeraktion ist keine Pflicht. Die Angaben zur Kontextbindung werden aus einer Tabelle für alle ConTracts in die Relation mit den aktiven ConTracts kopiert.

Spezifikation:

Die Deklaration lautet:

```
int init_ctxtbind(cid, contractname)
long   cid;
char   *contractname;
```

Parameter	**Bedeutung**
cid	ConTract–Identifier
Contractname	Name desConTracts

Die Fehlercodes und ihre Bedeutung:

Fehlercode	**Bedeutung**
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_CONTRACTUNKOWN	Unbekannter ConTract–Name.
CT_ERR_CIDEXISTS	`cid` exisitert bereits.

B.2 Abfragefunktionen

B.2.1 Anzeigen aller Kontextelemente eines ConTracts

Anforderungen:

Alle Kontextelemente eines ConTracts können mit dieser Funktion angezeigt werden. Die Ausgabe erfolgt gemäß der unten aufgeführten Grammatik auf eine Ausgabedatei und kann so von anderen Programmen weiterverwendet werden, ohne direkten Zugriff auf die Kontextrelationen gewähren zu müssen und somit zu sehr vom Aufbau der Relationen abzuhängen.

Outputlist	→	Valuelist |
	→	Valuelist Outputlist
Valuelist	→	**NAME:** Elementname **TYP:** Typ <**nl**> List
List	→	**VERSION:** Version <**nl**> **INDEX:** Index **VALUE:** Value **PRODUCER_STEP:** Step–Id <**nl**> |
	→	**VERSION:** Version <**nl**> **INDEX:** Index **VALUE:** Value **PRODUCER_STEP:** Step–Id <**nl**> List

Die erzeugte Ausgabe orientiert sich an den Kontextelementnamen und gibt alle erzeugten Versionen aus. Sie gibt somit kein Abbild des zeitlichen Ablaufs. Sollte der Wert eines Kontextelementes sich aus mehreren Attributen zusammensetzen, so werden die einzelnen Attribute durch '|' getrennt ausgegeben. Dies trifft beim Prototypen auf den Kontexttyp `CURSOR` zu, der als Querystring und Position gespeichert wird.

Spezifikation:

Die C–Deklaration hierzu lautet:

```
int show_ctxt(cid, outputfile)
long   cid;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
outputfile	FILE–Pointer für Ausgabedatei

Die Funktion liefert Fehlernummern zurück. Foglende Fehler werden generiert:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_NOCTXTELFOUND	Angaben selektieren kein(e) Kontextelement(e).
CT_ERR_NOMEMORY	Während der Ausführung trat ein Speichermangel auf.

Die Ausgabe erfolgt über einen Filepointer und kann somit in eine Datei umgelenkt werden oder über eine 'Pipe' an ein anderes Programm weitergeleitet werden.

B.2.2 Anzeigen der neuesten Version eines Kontextelementes

Anforderungen:

Die neueste Version eines Kontextelementes soll angezeigt werden. Die Ausgabe erfolgt auch hier auf eine Ausgabedatei, und es wird auch die oben eingeführte Syntax verwendet.

Spezifikation:

Die Deklaration für C:

```
int show_ctxt_elname(cid, ctxt_elname, outputfile)
long   cid;
char   *ctxt_elname;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
ctxt_elname	Name des Kontextelementes
outputfile	FILE–Pointer für Ausgabedatei

Unterstützt werden die Fehlercodes:

Fehlercode	**Bedeutung**
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_TYPUNKNOWN	Unbekannte Kontexttypangabe.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Tritt ein Fehler auf, so erfolgt keinerlei Ausgabe. Die Ausgabe folgt der oben eingeführten Syntax.

B.2.3 Anzeigen eines bestimmten Kontextelementes

Anforderungen:

Zum Anzeigen einer exakt spezifizierten Ausprägung eines Kontextelementes soll diese Funktion dienen. Das Kontextelement wird durch seine gesamte Adresse gekennzeichnet.

Spezifikation:

Die Deklaration lautet:

```
int show_ctxt_element(cid, ctxt_elname, ctxt_version, ctxt_index,
                      outputfile)
long   cid;
char   *ctxt_elname;
long   ctxt_version;
long   ctxt_index;
FILE   *outputfile;
```

Parameter	**Bedeutung**
cid	ConTract–Identifier
ctxt_elname	Name des Kontextelementes
ctxt_version	Version des Kontextelementes
ctxt_index	Index des Kontextelementes
outputfile	FILE–Pointer für Ausgabedatei

Die Funktion liefert einen Fehlercode zurück. Es werden folgenden Codes unterstützt:

Fehlercode	**Bedeutung**
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_TYPUNKNOWN	Unbekannte Kontexttypangabe.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Die Ausgabe erfolgt natürlich in der oben angegebenen Syntax und auch nur dann, wenn kein Fehler aufgetreten ist.

B.2.4 Anzeigen der Historie eines Kontextelementes

Anforderungen:

Die zeitliche Entwicklung eines Kontextelementes zeigt diese Funktion. Es ist erlaubt, einen Bereich anzugeben, der sich auf die Versionsnummer bezieht. Hier werden dann nur Versionen ausgegeben, die diese Bereichseingrenzung erfüllen.

Durch diese Funktion ist es möglich, die zeitliche Entwicklung bzgl. der Veränderung eines Kontextelementes zu verfolgen. Zum Beispiel kann die Entwicklung eines Elementes in einer Schleife näher betrachtet werden.

Spezifikation:

```
int show_ctxt_history(cid, ctxt_elname, ctxt_from_version,
                      ctxt_to_version, outputfile)
long   cid;
char   *ctxt_elname;
long   ctxt_from_version;
long   ctxt_to_version;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
ctxt_elname	Name des Kontextelementes
ctxt_from_version	Listen ab Version
ctxt_to_version	Listen bis Version
outputfile	FILE–Pointer für Ausgabedatei

Mögliche Fehlercodes sind:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Die Ausgabe in obiger Syntax und nur, wenn kein Fehler aufgetreten ist. Wird als Wert von `ctxt_from_version` und `ctxt_to_version` 0 übergeben, so werden alle Versionen dieses Elementes ausgegeben. Selektiert die Versionsangabe mindestens ein Element, so wird keine Fehlermeldung ausgegeben, auch wenn Versionen im Bereich fehlen.

B.2.5 Anzeigen der letzten Versionen eines Kontextelementes

Anforderungen:

Mit dieser Funktion können die letzten Versionen eines Kontextelementes zur Anzeige gebracht werden. Die Anzahl der anzuzeigenden Elemente wird als Parameter übergeben.

Spezifikation:

```
int show_ctxt_last_versions(cid, ctxt_elname, version_count, outputfile)
long   cid;
char   *ctxt_elname;
long   version_count;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
ctxt_elname	Name des Kontextelementes
version_count	Anzahl der Version die ausgegeben werden sollen.
outputfile	FILE–Pointer für Ausgabedatei

Unterstützte Fehlercodes sind:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.
CT_ERR_TYPUNKNOWN	Unbekannte Kontexttypangabe.

Wird mindestens eines der n letzten Elemente selektiert, wird keine Fehlermeldung ausgegeben.

B.2.6 Ausgabekontext einer Aktion

Anforderungen:

Der Kontext, den eine bestimmte Aktion als Ausgabekontext gesichert hat, kann durch diese Funktion zur Anzeige gebracht werden.

Spezifikation:

Die Deklaration für die Programmiersprache C:

```
int show_ctxt_outstep(cid, step_id, step_version, outputfile)
long   cid;
long   step_id;
long   step_version;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
step_id	Nummer der Aktion (Step)
step_version	Version der Aktion
outputfile	FILE–Pointer für Ausgabedatei

Mögliche Fehlercodes sind:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Die Ausgabe erfolgt in obiger Syntax, wenn kein Fehler aufgetreten ist.

B.2.7 Anzeigen des Ausgabekontextes einer Menge von Aktionen

Anforderungen:

Diese Funktion entspricht der vorherigen bis auf die Tatsache, daß hier eine Menge von Aktionen spezifiziert wird, deren Ausgabekontext zur Anzeige gebracht werden soll. Die Aktionen werden durch ihren Identifikator und ihre Version spezifiziert, wobei die Version die Anzahl der Aktivierungen bezeichnet.

Die Funktion ermöglicht es, die Entwicklung von Kontextelementen entlang dem Ausführungspfad einzelner Aktionen zu verfolgen und erleichtert somit die Rekonstruktion des Ablaufs.

Spezifikation:

Die Deklaration lautet:

```
int show_ctxt_steplist(cid, steplist, outputfile)
long                cid;
struct LIST_OF_STEP *steplist;
FILE                *outputfile;
```

Die verwendete Listenstruktur hat folgendes Aussehen:

```
struct LIST_OF_STEP
 {
    long              step_id;
    long              step_version;
    struct LIST_OF_STEP *next;
 };
```

Parameter	**Bedeutung**
cid	ConTract–Identifier
steplist	Zeiger auf eine Liste mit Step–Ids
outputfile	FILE–Pointer für Ausgabedatei

Die Funktion liefert Fehlernummern zurück. Es werden folgende Codes unterstützt:

Fehlercode	**Bedeutung**
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_STEPLIST	Fehler in der Step–Liste. Angaben in der Liste spezifizieren keine Aktion. Der Listeneintrag wird ausgelassen.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Sollte ein Stepname in der Liste nicht vorhanden sein, so wird die Ausgabe nicht unterbrochen, sondern zum nächsten Element in der Liste weitergeschaltet. Tritt kein schwerwiegender Fehler auf, wird der Fehlercode `CT_ERR_STEPLIST` zurückgeliefert. Eine Ausgabe der anderen Daten erfolgt aber dennoch.

B.2.8 Eingabekontext einer Aktion

Anforderungen:

Der Eingabekontext einer bestimmten Aktion kann durch diese Funktion angezeigt werden.

Spezifikation:

```
int show_ctxt_instep(cid, step_id, step_version, outputfile)
long   cid;
long   step_id;
long   step_version;
FILE   *outputfile;
```

Parameter	Bedeutung
cid	ConTract–Identifier
step_id	Nummer der Aktion (Step)
step_version	Version der Aktion
outputfile	FILE–Pointer für Ausgabedatei

Die Fehlercodes dieser Funktion haben die Bedeutung:

Fehlercode	Bedeutung
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_STEPUNKOWN	Unbekannter Aktionsname.
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Die Ausgabe auf das Ausgabefile entspricht der oben eingeführten Syntax und erfolgt wiederum nur dann, wenn kein Fehler aufgetreten ist.

Die Versionsnummer hat folgende Semantik:

1. Ist der Erzeugerstep bekannt, d.h. in der Kontextbinderelation eingetragen, so wird das entsprechende Element mit der gleichen Versionsnummer selektiert.

2. Wenn der Erzeugerstep unbekannt ist, so werden die Versionen von Kontextelementen ausgegeben, die eine um 1 kleinere Versionsnummer haben als angegeben. Dies ist z.B. in einer Schleife der Fall.

B.3 Tabelle der benützten Fehlercodes

Die Fehlercodes, die bei den Kontextelementverwaltungs- und Kontextabfragefunktionen auftreten können, sind hier noch einmal in einer Tabelle zusammengefaßt. Ihre genauen Werte sind der Include–Datei (`ktxtver.h`) zu entnehmen.

Fehlercode	**Bedeutung**
NO_ERROR	Fehlerfreie Ausführung.
CT_ERR_CIDUNKOWN	Unbekannte `cid`.
CT_ERR_DB_ROLLBACK	Datenbankfehler, Transaktion zurückgesetzt.
CT_ERR_TYPUNKNOWN	Unbekannte Kontexttypangabe.
CT_ERR_NOCTXTELFOUND	Angaben selektieren kein(e) Kontextelement(e).
CT_ERR_NOVERSION	Es hat sich kein Kontextelement für die Ausgabe qualifiziert.
CT_ERR_STEPLIST	Fehler in der Step–Liste. Angaben in der Liste spezifizieren keine Aktion. Der Listeneintrag wird ausgelassen.
CT_ERR_TYPINKONSISTENCY	Typangabe in der Kontextbinderelation ist inkonsistent.
CT_ERR_CONTRACTUNKOWN	Unbekannter ConTract-Name.
CT_ERR_CIDEXISTS	`cid` exisitert bereits.
CT_ERR_NOMEMORY	Während des Ausführung trat ein Speichermangel auf.

Anhang C

Bemerkungen zum ANSI/ISO–SQL–Standard

Im Rahmen dieser Studienarbeit wurde der **ANSI/ISO**–SQL–Standard[1] ([ANSI]) des öfteren erwähnt. Hier nun die wichtigsten Auszüge und einige Anmerkungen.

C.1 Datentypen im Standard

Der **ANSI/ISO**–SQL Standard spezifiziert verschiedene Datentypen, welche in auf SQL-basierenden Datenbanksystemen manipuliert werden können. Die im Standard erwähnten Datentypen bilden aber nur eine minimale Menge. Alle kommerziellen Datenbankhersteller unterstützen diese oder haben Datentypen, die denen des Standards entsprechen. Folgende Kategorien existieren:

- Zeichenketten fester Länge.
- Ganze Zahlen.
- Festpunktzahlen.
- Gleitkommazahlen.

[1] Der Standard selbst konnte nicht beschafft werden, da an der Universtätsbibliothek der Universität Stuttgart nur ein Verzeichnis aller erhältlichen ANSI–Standards vorhanden ist. Dieses Verzeichnis enthielt zwar Verweise auf die Bezugsmöglichkeit in den USA, jedoch erschien dem Autor der Preis dafür zu hoch. Die aufgeführten Angaben sind dem Buch *USING SQL* von James R. Groff und Paul N. Weinberg ([GW90]) entnommen . Dort wird u.a. ausführlich auf den Standard, seine Schwächen und die Akzeptanz bei den kommerziellen Datenbankherstellern eingegangen.

Datentyp	Beschreibung
`CHAR`(length) `CHARACTER`(length)	Zeichenkette fester Länge
`INT` `INTEGER`	Ganze Zahlen
`SMALLINT`	Kleine ganze Zahlen
`NUMERIC`(precision, scale) `DECIMAL`(precision, scale) `DEC`(precision, scale)	Festpunktzahlen
`FLOAT`(precision)	Gleitkommazahlen
`REAL`	Gleitkommazahlen mit kleiner Genauigkeit
`DDOUBLE PRECISION`	Gleitkommazahlen mit großer Genauigkeit

Die im Rahmen der ConTract-Prototypimplementierung verwendeten relationalen Datenbanksysteme **Nonstop–SQL** und **TRANSBASE** beinhalten den **ANSI/ISO**–SQL Standard als echte Untermenge.

C.2 Dynamisches SQL

Dynamische Anfragen sind im ANSI–Standard nicht spezifiziert. Nahezu alle Datenbanken bieten allerdings diese Möglichkeit des Zugriffs an. Dementsprechend ergeben sich je nach System große Unterschiede in der Behandlung von dynamischen Anfragen sowie in der zur Verfügung stehenden Funktionalität.

Dieser Punkt ist für eine allgemeine Implementierung von Wichtigkeit, da bei bestimmten Implementierungsmöglichkeiten dynamische Anfragen gebraucht werden und dies nicht nur mit konstanter Anzahl von Attributen und Attributtypen. Aus der Nutzung der von der DB zur Verfügung gestellten Funktionalität kann also eine Abhängikeit vom DBMS resultieren.

C.3 Neuere Entwicklungen

Eine Vielzahl von Datenbankherstellern geht dazu über, ihre DBMS dahingehend zu erweitern, daß sogenante *Binary Large Objects (BLOBS)* als neuer Datentyp verwendet werden können. Diese Erweiterung bringt für das ConTract-Projekt Vorteile, da es dann auf einfache Art möglich wird, ganze Dateien, Scanner–Daten oder Inhalte von Fenster einer graphischen Benutzeroberfläche zu verwalten.

Eine ausführliche Gegenüberstellung der Datentypen und Erweiterungen der einzelnen Datenbanksysteme findet sich in [GW90] auf Seite 82 ff. Vergleiche und eine umfassende Diskussion über *Dynamic SQL* sowie über den Standard und dessen Zustandekommen sind ebenfalls vorhanden.

Anhang D

Literaturverzeichnis

[Ahu88] Sudhir Ahuja et.al. Matching Language and Hardware for Parallel Computation in the Linda Machine. *IEEE Transactions on Computers*, 37(8):921–929, August 1988.

[ANSI] *Draft Proposed American Standard Database Language SQL.* ANSI, New York, 1985, Draft ISO/TC97/SC21/WG5-15/N90.

[ASU88] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. *Compilerbau.* Addison-Wesley, 1988.

[Böt89] Stefan Böttcher. Prädikative Selektion als Grundlage für Transaktionssynchronisation und Datenintegrität. Dissertation, Fachbereich Informatik der Universität Frankfurt, Dezember 1989. Enthält Beschreibung der Datenbankprogrammiersprache DBPL.

[BT88] Henrie E. Bal, Andrew S. Tanenbaum. Distributed Programming with Shared Data. In *Proceedings of the IEEE CS 1988 International Conference on Computer Languages*, S. 82–91, New York, 1988.

[CG89] Nicholas Carriero, David Gelernter. Linda: Some Current Work. In *IEEE Spring Compcon*, S. 98–101, 1989.

[CLNW89] S. M. Clamen, L. D. Leibengood, S. M. Nettles, J. M. Wing. Reliable Distributed Computing with Avalon/Common Lisp. Technischer Bericht CMU-CS-89-186, Carnegie Mellon University, September 1989.

[GMS87] Hector Garcia-Molina, Kenneth Salem. Sagas. In *ACM Proceedings*, S. 249–259, 1987.

[GW90] James R. Groff, Paul N. Weinberg. *Using SQL.* Osborne McGraw-Hill, 1990.

[HHS89] Cristoph Hübel, Christoph Hussong, Bernd Sutter. *Entwurf und Implementierung eines Preprocessors für EMQL (Embedded Molecule-Query-Language).* Universtät Kaiserslautern, Fachbereich Informatik, Dezember 1989.

[HS89] Cristoph Hübel, Bernd Sutter. *Verarbeitung komplexer DB-Objekte in Ingenieuranwendungen* 5/89. Zentrum Rechnergestützter Ingenieursysteme, Universität Kaiserslautern, Juni 1989.

[KR83] Brian W. Kernighan, Dennis M. Ritchie. *Programmieren in C.* Carl Hanser Verlag, München Wien, 1983.

[Lel89] Wm Leler. Linda Meets UNIX (A Version of Linda for System-Level Programming). Technical Note, Cogent Research, Beaverton, Oregon USA, Oktober 1989.

[LSch87] P.C. Lockemann, J.W. Schmidt (Hrsg.). *Datenbankhandbuch.* Springer-Verlag, 1987.

[Mas90] Ioannis Masmanidis. Definition einer Sprache zur Spezifikation verteilter, paralleler Abläufe und Implementierung eines Compilers für diese Sprache. Studienarbeit, Institut für Informatik der Universität Stuttgart, in Vorbereitung, 1990.

[MW88] Klaus Meyer-Wegener. *Transaktionssysteme.* Teubner Verlag, Stuttgart, 1988.

[ONe86] Patrick E. O'Neil. The Escrow Transactional Method. *ACM Transactions on Database Systems*, 11(4):405–430, Dezember 1986.

[PR88] Peter Peinl, Andreas Reuter. High Contention in a Stock Trading Database: A Case Study. In *Proceedings of the ACM SIGMOD Conference*, S. 260–268, 1988.