Seminararbeit

Software Transactional Memory in Clojure

Otto Allmendinger

12. Dezember 2012

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel der Arbeit

Bedingt durch physikalische Limitation kann die Taktrate von Prozessoren nicht bis ins Unendliche erhöht werden - Mehrkernprozessoren setzen sich zunehmend durch, die statt sequenzieller Ausführung von Programmen eine nebenläufige Arbeitsweise anbieten[Sut05]. Um alle Vorteile der Nebenläufigkeit ausnutzen zu können, sind Programmiersprachen gefragt, die die Entwicklung derartiger Programme erleichtern.

Vor allem sogenannte funktionale Programmiersprachen haben hier in letzter Zeit Interesse geweckt. Eine davon ist die Sprache *Clojure*, ein Dialekt der Programmiersprache Lisp. Programme in dieser Sprache können auf der *Java Virtual Machine* und der *Common Language Runtime (.NET)* ausgeführt werden[Hic12].

In dieser Arbeit werden die verschiedenen neuartigen Mechanismen zur nebenläufigen Speicherverwaltung in Clojure untersucht und anschaulich an Beispielen erklärt.

1.2 Funktionale Programmierung

Funktionale Programmierung betont einen Programmierstil, der großen Wert auf *„First-Class“* Funktionen ohne Nebenwirkungen legt. [Bur00] Eine Funktion ist First-Class, wenn sie wie ein Objekt behandelt werden kann, also etwa durch eine Variable referenziert werden kann, die dann den Typ „Funktion“ hat. Funktionen können dann auch Parameter zu anderen Funktionen oder Rückgabewerte von anderen Funktionen sein.

Eine Funktion hat dann keine Nebenwirkungen, wenn sie bei wiederholtem Aufruf mit den gleichen Parametern auch das gleiche Ergebnis liefert und sonst kein Veränderung eines Zustands erzeugt, im Unterschied zu Funktionen die von einem Zustand abhängen, der nicht in den Parametern definiert ist. Eine Funktion, die in eine Datei oder in ein Terminal schreibt, hat also Nebenwirkungen, da auf das Dateisystem zugegriffen wird oder der Zustand des Terminals verändert wird.

In funktionalen Programmiersprachen wird der Zustand also meist mit in Parametern definiert, die auf dem Stack liegen, statt durch globale Variablen, die im Heap alloziert sind.

Jedes sinnvolle Programm enthält Nebenwirkungen, wie etwa die Ausgabe von Text. Das Ziel funktionaler Programmiersprachen ist es, diese Nebenwirkungen zu isolieren und es zu vermeiden diese im Programmcode zu verteilen.

1.3 Die Lisp-Familie

Clojure gehört der *Lisp*-Familie der Programmiersprachen an, die im Jahre 1958 von John McCarthy spezifiziert wurden[McC62]. Lisp-Sprachen zeichnen sich dadurch aus, dass sie eine Äquivalenz von Programmcode und Daten herstellen, was eine besonders dynamische Programmierung ermöglicht. Die Syntax von Lisp ist im Unterschied zu den meisten heute gebräuchlichen Programmiersprachen sehr einfach – die gesamte Programmstruktur besteht aus verschachtelten Listen. Ein Beispiel ist etwa `(A B C)`. Ein üblicher Lisp-Interpreter würde dieses Programm so evaluieren, dass die Funktion `A` mit den Parametern `B` und `C` ausgeführt wird, wobei `B` und `C` selbst wieder Listen sein können. Anders als in anderen Sprachen gibt es keine Operatoren oder Schlüsselwörter. Das bedeutet, dass Ausdrücke aus Algol-abgeleiteten Sprachen (wie Java) auf andere Weise dargestellt werden.

```
1 // Java, C, ...
2 4 + (1 + 2) * 3
```

In Clojure sind auch Operatoren als Funktionen implementiert. Der äquivalente Programmcode hätte dann folgende Gestalt:

```
1 ; Lisp
2 (+ 4 (* (+ 1 2) 3))
```

So lässt sich mit einer Lisp-Sprache und nur einigen wenigen vorher definierten Funktionen eine eigene Programmiersprache spezifizieren, die eigene Kontrollstrukturen, Vererbungsmechanismen, Objektorientierung und andere Eigenschaften enthält. Lisp wird daher auch oft als „programmierbare Programmiersprache" bezeichnet.

1.4 Clojure

Die Programmiersprache Clojure wurde von Rich Hickey, dem primären Autor, speziell für die Probleme der nebenläufigen Programmierung entwickelt. Nach jahrelanger Erfahrung mit Programmierung von nebenläufigen Systemen in klassischen Programmiersprachen gelangte er zur Meinung, dass Probleme des gleichzeitigen Zugriffs auf Daten nicht durch traditionelle Methoden wie Locks und Semaphoren lösbar sind und bessere Kontrollstrukturen auf der Sprachebene notwendig sind.

Clojure ist ein dynamisch stark typisiertes, funktionales Lisp, das primär in Java entwickelt und auf der *Java Virtual Machine* ausgeführt wird. Hierbei wird großer Wert auf die eine einfache Zusammenarbeit mit der Java-Runtime Library gelegt.

1.4.1 Hello World

Das Programm *Hello World* hat diese Gestalt

```
(defn main [name]
  (println "Hello," name))
```

Der Verzicht auf syntaktische Elemente wie Kommata und geschweiften Klammern ist für Lisp-Entwickler von hoher Bedeutung. Der Nachteil der Syntax ist die hohe Anzahl von Klammern, die Anfangs von Vielen als störend empfunden wird.

1.5 Datenstrukturen in Clojure

In funktionalen Programmiersprachen sind Datenstrukturen meist unveränderlich. Das bedeutet, dass nach der Erzeugung eines Objekts dessen Zustand nicht verändert werden kann. In vielen Fällen ist die Verwendung von unveränderlichen Objekten sicherer und leichter zu verstehen, besonders im Hinblick auf Nebenläufigkeit und Parallelität.

Ein Beispiel für einen unveränderlichen Datentyp ist die Java-Klasse `String`:

```
String s = "ABC";
s.toLowerCase();
```

Die Methode `toLowerCase` ändert den Inhalt des durch `s` referenzierten String nicht, sondern gibt einen neuen String zurück. Damit der mit `s` angegebene String die Zeichenkette `"abc"` enthält, muss die alte Referenz überschrieben werden.

```
String s = "ABC";
s = s.toLowerCase();
```

In Clojure sind alle Datentypen grundsätzlich unveränderlich. Um etwa eine Liste zu erweitern wird nicht etwa die ursprüngliche Liste manipuliert, sondern eine neue Liste mit einem ergänzten Element erzeugt.

```
(def mylist (list 1 2 3 4))
(def mylist-plus (conj list 5)) ; (1 2 3 4 5)
```

Dieser Ansatz funktioniert gut in Kombination mit funktionaler und nebenläufiger Programmierung. Eine Funktion, die als Parameter unveränderliche Datenstrukturen erhält, kann zum Beispiel keine Nebenwirkungen auf diese Datenstrukturen haben.

2 Nebenläufigkeit

2.1 Nebenläufige Prozesse

Die Kontrolle nebenläufiger Prozesse (oder Threads) ist ein großes Thema der Theoretischen Informatik. Das zentrales Problem ist hier der gleichzeitige Zugriff und Manipulation gemeinsamer Daten.

Ein einfaches Beispiel ist hier etwa die Verschiebung eines Elements aus einer Datenstruktur in eine Andere. Für andere Prozesse, die ebenfalls auf diese Datenstrukturen zugreifen, soll gewährleistet sein, dass das zu verschiebende Element sich immer in genau einer Datenstruktur befindet.

Üblicherweise gibt es auf Computersystemen nur wenige elementare Operationen, sofort für alle Prozesse sichtbar gemacht werden können. Solche Operationen werden auch als *atomar* bezeichnet (von griechisch *atomos*, „unteilbar"). Daher sind komplizierte, zusammengesetzte Operationen wie das Verschieben von Daten-Elementen, nicht ohne weiteres als einzelne atomare Operation abbildbar.

2.2 Locking

Bei imperativen Programmiersprachen kommt hier oft Locking zum Einsatz. Dieser Mechanismus erlaubt es, eine Menge von Instruktionen, die gemeinsam genutzte Daten verändern können, vor gleichzeitigem Zugriff zu schützen, so dass nur ein Prozess Zugriff auf eine Gruppe von Daten erhält. Somit wird gewährleistet, dass Änderungen, die mehrere Daten-Gruppen umfassen, atomar sind[Ora12]. Beispiel

```
1 synchronized void transfer(
2     Account source, Account target, int amount) {
3
4   source.withdraw(amount);
5   target.deposit(amount);
6 }
```

Der Vorteil dieser Methode ist, dass sie weit verbreitet ist und viele Entwickler damit Erfahrung haben. Zu den Nachteilen gehört

- Locks sind prinzipiell pessimistisch, da sie auch Zugriff verweigern, der nur lesend ist. Dies führt zu einer Reduktion der theoretisch möglichen Nebenläufigkeit.

- Bei Prozessen, die gegenseitige Abhängigkeiten haben, kann es zu unerwünschtem Verhalten kommen. So kann es etwa zu einem **Deadlock** (dt *Verklemmung*) kommen, wenn der Prozess A auf Prozess B wartet und umgekehrt, so dass beide Prozesse zum erliegen kommen. Ein **Livelock** kann entstehen, wenn zudem beide Prozesse Arbeit verrichten, während sie auf die Freigabe eines Locks warten[SGG06].

- Der korrekte Einsatz von Locks beruht auf informalen Konventionen. Bei hierarchisch geordneten Locks wird etwa vereinbart, dass Locks alphabetisch gesetzt und in umgekehrter Reihenfolge wieder frei gegeben werden müssen. Verstöße gegen die Konventionen sind sprachlich und semantisch Korrekt, können aber zu Fehlern führen, die nur schwer zu finden sind [Har+05].

- Datenstrukturen, die Locks zu Prozesskontrolle verwenden, können nur schwer mit anderen solchen kombiniert werden. Als Beispiel betrachten wir eine Datenstruktur `HashMap`, welche die Methoden `delete` und `insert` in einer Weise implementiert, die nebenläufigen Zugriff erlaubt. Wenn ein Element von einer `HashMap` in eine Andere übertragen werden soll, ohne dass der Zwischenzustand sichtbar ist, in dem das Element in einer der beiden Strukturen fehlt, kann dies nicht durch bestehende Mechanismen erreicht werden. Die Implementierung kann zwar die Methoden `LockMap` und `UnlockMap` bereit stellen, verliert dann aber an Abstraktion bringt die Gefahr eines Deadlocks mit sich, da dann wieder die zuvor beschriebenen Konventionen eingehalten werden müssen. Individuell korrekt implementierte Methoden können also nicht zu größeren, ebenfalls aus Sicht der Prozess-Kontrolle korrekten Operationen vereint werden[Har+05].

2.3 Software Transactional Memory

Eine Alternative zum Locking ist ein Ansatz, der aus dem Umfeld von Datenbanken kommt. Dort werden zur gleichzeitigen Manipulation mehrerer Datenstrukturen Transaktionen genutzt, um die sogenannte *ACID*-Kriterien zu erfüllen.

Atomic Es werden entweder alle in einer Transaktion beschriebenen Änderungen übernommen, oder keine.

Consistent Validierungs-Regeln, die für die Datenstrukturen definiert sind, werden nicht verletzt.

Isolated Für andere Prozesse sind die veränderten Datenstrukturen erst sichtbar, nachdem sie erfolgreich übernommen wurden.

Durable Erfolgreich übernommene Änderungen sind beständig, auch wenn die Prozesse unerwartet beendet werden (etwa durch einen Softwarefehler). Dieses Kriterium wird bei STM üblicherweise nicht erfüllt.

Transaktionen werden meist mit einer speziellen Syntax deklariert, die je nach Implementierung unterschiedlich ist. Aus der Sicht anderer Prozesse ändern sich alle betroffenen Daten gleichzeitig beim erfolgreichen Abschluss der Transaktion.

Jede Transaktion arbeitet mit einem konsistenten Schnappschuss der Daten. Nur wenn die Daten, die in der Transaktion `A` gelesen werden, vor Abschluss der Transaktion von einer anderen Transaktion `B` verändert wurden, werden die berechneten der Transaktion `A` verworfen und auf Basis der veränderten Eingabe-Daten wiederholt. Hier wird optimistische Strategie der Prozess-Kontrolle deutlich, die im Unterschied zum Locking gleichzeitigen Zugriff auf Datenstrukturen zulässt. Wenn der Zugriff hauptsächlich lesend ist, kommt es auch selten zur Wiederholung von Transaktionen.

Da eine Transaktion trotzdem wiederholt werden kann, muss beachtet werden, dass die Berechnung der neuen Werte keine Nebenwirkungen enthält sondern nur die Beschreibung der zu ändernden Daten ist. Andernfalls kann eine mehrfacher Ausführung unerwünschte Folgen haben können. Zu den Vorteilen transaktionaler Speicher-Systeme gehört

- Vereinfachte Handhabung als Entwickler. Statt das korrekte Verhalten des Programms als Entwickler selbst durch Auflösung der gegenseitigen Abhängigkeiten und Einhaltung informeller Konventionen zu gewährleisten, wird diese Aufgabe der Implementierung des transaktionalen Speicher-Systems überlassen, welche die Beschreibung der Abhängigkeiten enthält.
- Erhöhte Nebenläufigkeit durch optimistische Strategie. Während Locks Prozesse an gleichzeitigem Zugriff auf Datenstrukturen hindern, auch wenn Beispielsweise im Fall zweier nebenläufiger Lese-Operationen kein Bedarf besteht, können Transaktionen immer parallel ablaufen.
- Ein transaktionales Speicher-System kann garantieren, dass es nicht zu einem Deadlock oder Livelock kommt.
- Im Unterschied zu Locks lassen sich einzelne Transaktionen vereinen (Komposition). Dadurch ist eine höhere Abstraktion möglich. [Har+05]

Die Nachteile sind

- Hoher Verwaltungsaufwand durch Erfassung der Referenzen und Erfassung von Schreib-Vorgängen bei denen referenzierte Werte überschrieben werden
- Gefahr unnützer Arbeit wenn viele Transaktionen wiederholt werden müssen
- Bei einfachen Transaktions-Systemen kann es passieren, dass Transaktionen auch nach vielen Wiederholungen nicht erfolgreich beendet werden können.
- Ungeschickt abgegrenzte Transaktionen mit hohem Zeitaufwand laufen Gefahr, auch bei intelligenteren Transaktions-Systemen oft wiederholt zu werden und können unter Umständen ebenfalls nicht erfolgreich abgeschlossen werden.

2.3.1 Beispiel MySQL

In Datenbanken ist die Unterstützung von Transaktionen auf Sprachebene implementiert. Beispiel MySQL:

```
START TRANSACTION;

UPDATE TABLE money
  SET balance = (balance - amount)
  WHERE accountId = source;

UPDATE TABLE money
  SET balance = (balance + amount)
  WHERE accountId = target;

COMMIT;
```

Für keinen anderen Client der Datenbank ist der ein Zustand sichtbar, in dem das Geld auf einem Konto fehlt und auf einem Anderen sichtbar ist.

2.3.2 Beispiel Haskell

Die funktionale Programmiersprache *Haskell* bietet STM als Teil des Typensystems

```
withdraw :: TVar Int -> Int -> STM ()
withdraw acc n = do bal <- readTVar acc
                    if bal < n then retry
                    writeTVar acc (bal-n)

deposit :: TVar Int -> Int -> STM ()
deposit acc n = do bal <- readTVar acc
                   writeTVar acc (bal+n)

deposit :: TVar Int -> TVar Int -> Int -> STM ()
transfer from to n = do withdraw from n
                        deposit to n
```

Das Typensystem garantiert hier, dass der Aufruf `transfer` nur atomar in einem `atomically` Block ausgeführt werden kann[Jon07].

2.3.3 Beispiel Java

Für viele imperative Sprachen kann STM als Bibliothek implementiert werden. Für Java gibt es das Projekt *Multiverse*, das mithilfe spezieller Datentypen die Verarbeitung von Transaktionen erlaubt:

```
1  import org.multiverse.api.references.*;
2  import static org.multiverse.api.StmUtils.*;
3
4  public class Account{
5
6    private final TxnRef<Date> lastUpdate;
7    private final TxnInteger balance;
8
9    public Account(int balance){
10     this.lastUpdate = newTxnRef<Date>(new Date());
11     this.balance = newTxnInteger(balance);
12   }
13
14   public void incBalance(final int amount, final Date date){
15     atomic(new Runnable(){
16       public void run(){
17         balance.inc(amount);
18         lastUpdate.set(date);
19
20         if(balance.get()<0){
21           throw new IllegalStateException("Not enough money");
22         }
23       }
24     });
25   }
26 }
```

Da es keine direkte Unterstützung in der Sprache gibt, müssen Transaktionsanweisungen als Methodenaufrufe wie `balance.inc`, `balance.set` und `lastUpdate.set` kodiert werden. Dadurch wird die Verwendung unbequem und das Verständnis einschränkt. Trotzdem gibt es auch hier die gleichen Vorteile gegenüber dem Locking.

2.4 Clojure STM

In Clojure ist die Unterstützung von Transaktionen ähnlich, wie bei Datenbanken, Teil der Sprache. Als Lisp-Dialekt hat Clojure den Vorteil, dass Programm-Anweisungen und Daten die gleiche Struktur haben und die Beschreibung einer Transaktion sich nicht grundlegend

von gewöhnlichen Anweisungen unterscheidet. Die Unterstützung für unveränderlicher, persistenter Datenstrukturen macht es zudem einfacher, für Transaktionen unproblematische, performante Routinen zu schreiben, die keine Nebenwirkungen haben. Die Transaktion für die Veränderung eines Kontostands wäre etwa

```
;; Definition einer Funktion
(defn transfer [source target amount]
  ;; Deklaration einer Transaktion
  (dosync
    ;; Beschreibung der vorzunehmenden Aenderungen
    (alter source - amount)
    (alter target + amount)))
```

Die Voraussetzungen für diese ist, dass die Parameter `source` und `target` vom Typ `ref` sind. Dieser Datentyp ist etwa Vergleichbar mit einem Pointer in der Programmiersprache `C`, dadurch dass der Wert nur durch eine Dereferenzierung gelesen werden kann

```
;; Definition einer Referenz
(def value (ref 0))

;; Auslesen des Werts
(println "value= " @value)
```

Innerhalb einer Transaktion können verschiedene Formen zur Manipulation der Referenzen eingesetzt werden

`(deref ref)` oder `@ref`
: Dereferenzierung

`(ref-set ref val)`
: Setzt die Referenz auf einen neuen Wert.

`(alter ref fun & p)`
: Setzt die Referenz auf den Rückgabewert der Ausdrucks `(apply fun alte-referenz p)`. (Die Schreibweise `& p` zeigt an, dass die Parameter `p` optional sind).

`(commute ref fun & p)`
: Gleiche Funktion wie `alter`, mit dem Unterschied dass hier eine kommutative Manipulation deklariert wird. Wenn also eine Transaktion wiederholt werden muss, weil sich die Eingangsdaten geändert haben, so werden Änderungen die durch `commute` beschrieben werden, nicht unbedingt in der gleichen Reihenfolge durchgeführt.

`(ensure ref)`
: Stellt sicher, dass in der Transaktion die Referenz `ref` nicht verändert wurde

2.4.1 Multi-Version Concurrency Control

Die Implementierung des Transaktions-Systems in Clojure basiert auf dem Konzept der *Multi-Version Concurrency Control*[BG81], (kurz MVCC), das traditionell in Datenbanken Verwendung findet. Die zentrale Idee dahinter ist, die Aktualisierung von Datensätzen nicht durch Überschreiben zu realisieren, sondern dadurch, mehrere Versionen eines Wertes zu Pflegen. Wenn also ein Wert `x` geändert werden soll, wird nicht etwa der alte Wert gelöscht, sondern die Referenz von `x` auf die neuste Version geändert, unter Beibehaltung der vorigen Versionen von `x`.

2.4.2 Snapshot Isolation

Das Konzept der *Snapshot Isolation* beinhaltet, jedem Prozess zu Beginn einer Transaktion eine konsistente Sicht auf die Daten zu gewähren. Bei Ende der Transaktion findet eine Übernahme nur dann statt, wenn die gelesenen Werte nicht zwischenzeitlich verändert wurden.

Ein dabei potenziell auftretendes Problem ist der sogenannte *write skew*. Dieser Fehlerfall kann auftreten, wenn zwei Transaktionen getrennte Werte in einem Aggregat ändern, das letztlich einen ungültigen Zustand hat, obwohl beide Transaktionen allein gültig wären. Ein Beispiel dafür wäre die Bedingung, in der Mensa entweder ein Stück Obst oder ein Dessert zu nehmen, aber nicht beides. Die gleichzeitigen Transaktionen *„nehme Obst"* und *„nehme Dessert"* sind zwar für sich gesehen valide, der endgültige Zustand ist es aber nicht. Hier bietet Clojure die Form `(ensure ref)`, die gewährleistet, dass sich eine gegeben Referenz während einer Transaktion nicht ändert, ohne ihr explizit einen neuen Wert zuzuweisen.

2.4.3 Implementierung

Eine Transaktion wird erzeugt, in dem eine beliebige Anzahl von Ausdrücken dem Makro `dosync` übergeben wird, die dann *atomar* abgewickelt werden. Der Entwickler entscheidet dann, welche Referenzen am Ende der Transaktion einen konsistenten Zustand haben sollen. Die Verwaltung des Transaktions-Status und dem Status der Referenzen ist in Java implementiert[Vol09].

Der interne Zustand einer Transaktion ist einer von

- `RUNNING` - wird ausgeführt
- `COMMITTING` - übernimmt die Werte der Referenzen
- `RETRY` - wird in Kürze wiederholt
- `KILLED` - wurde abgebrochen (siehe priorisierten Wiederholung)
- `COMMITTED` - Werte wurden übernommen

Werte-Verlauf und Zwischenwerte

Jede Referenz verwaltet eine Kette übernommener Werte und der Transaktionen, die für die Werte verantwortlich wird. Diese Kette wird hier als *Werte-Verlauf* bezeichnet. Dieser Kette wird eine kleinste und größte Länge zugeordnet.

Wenn `ref-set` oder `alter` auf einer Referenz aufgerufen wird, wird dieser ein für die aktuelle Transaktion gültiger Wert zugeordnet. Dabei wird im Werte-Verlauf hinterlegt, welche Transaktion für die Änderung verantwortlich ist und wann diese Transaktion begonnen hat. Auf diese Weise kann ermittelt werden, ob eine andere Transaktion eine noch nicht übernommene Referenz verändert hat. Der neue Wert für diese Referenz wird für die verbleibende Transaktion als Zwischenwert gespeichert.

Verhalten bei Transaktionsfehlern

Wenn in einer Transaktion ein Wert aus einer Referenz gelesen wird, kein Zwischenwert vorliegt und im Werte-Verlauf kein Eintrag zu finden ist, der *vor* Beginn der aktuellen Transaktion gültig war, kommt es zum einem Transaktionsfehler. In diesem Fall wird die Transaktion wird wiederholt. Dieser Fall kann eintreten, wenn eine andere Transaktion den Wert der Referenz verändert hat. Bei der Übernahme eines Wertes für eine Referenz, die in einer anderen Transaktion einen Fehler übernommen hat, wird ein neuer Knoten in der Werte-Verlaufs-Kette angelegt, falls die für die Kette hinterlegte maximale Länge nicht überschritten wird. Dieser Vorgang erfolgt auch dann, wenn die Mindestlänge für den Werte-Verlauf nicht erreicht ist. Diese Erweiterung des Werte-Verlaufs reduziert die Wahrscheinlichkeit eines Fehlers in folgenden Transaktionen.

Bei einer Wiederholung werden alle Änderungen, die in der Transaktion vorgenommen wurden, verworfen. Vorraussetzung dafür ist, dass in der Transaktion keine Nebenwirkungen verursacht worden (siehe Abschnitt „Nebenwirkungen")

Falls es für diese Referenz keine fehlerhaften Transaktionen vorliegen, wird der letzte Knoten in der Kette als erster Knoten gesetzt, der dann den neusten gültigen Wert beschreibt.

Priorisierte Wiederholung

Die bisherigen Mechanismen schützen nicht vor dem Zustand, in dem eine Transaktion dauerhaft wiederholt wird, ohne erfolgreich abzuschließen. Für dieses Szenario existiert eine Strategie, die es erlauben soll, laufende Transaktionen zugunsten von wartenden Transaktionen zu unterbrechen[Eme12]. Dazu müssen folgende Kriterien erfüllt sein.

Eine Transaktion kann eine Wiederholungs-Versuch unternehmen, während eine andere Transaktion abgearbeitet wird, wenn drei Kriterien zutreffen

1. Die zu wiederholende Transaktion hat vor der laufenden Transaktion begonnen. Dadurch wird die Abwicklung wartender Transaktionen begünstigt.

2. Die zu wiederholende Transaktion wurde eine gewisse Zeit ausgeführt. Dieser Wert wurde auf 10ms fest gelegt.

3. Die laufende Transaktion hat den Status `RUNNING` und kann auf `KILLED` gesetzt werden. Dies verhindert, das Transaktionen, die gerade den Zustand geänderter Referenzen übergeben (`COMMITTING`) unterbrochen werden.

Wenn die Transaktion es trotzdem nicht schafft, erfolgreich ausgeführt zu werden, wird sie nach 10000 Versuchen abgebrochen und eine generische `Exception` ausgelöst, da ein Abbruch des Programms einem Livelock zu bevorzugen ist.

Nebenwirkungen

Da Transaktionen wiederholt werde können, ist es unerwünscht, in einer Transaktion den Zustand des Programms ohne die Nutzung von Referenzen direkt zu verändern. Hier bietet Clojure das Makro `io!`, mit dem Funktionen gekennzeichnet werden können, die Nebenwirkungen haben.

```
1 (defn io-print [& arguments]
2   (io! (apply println arguments)))
```

Die Anweisung `(dosync (io-print "abc"))` führt dann zu einer `IllegalStateException`.

3 Beispiele

3.1 Transfer zwischen Konten

3.1.1 Beschreibung

Eine einfaches Beispiel für eine Transaktion ist der zuvor beschriebene Vorgang, von einem Konto einen Betrag abzubuchen und ihn gleichzeitig auf einem anderen Konto erscheinen zu lassen. Dieser Fall lässt sich dank STM leicht abdecken, indem die Operation "dekrementierenünd ïnkrementierenïn eine Transaktion aufgenommen werden.

3.1.2 Implementierung

Wir lassen in dieser und in folgenden Implementierungen an geeigneten Stellen mittels `Thread/sleep` etwas Zeit verstreichen, um den Zeitaufwand komplexerer Berechnung zu imitieren.

```
1  ;; Definiere einen Vektor von 6 Referenzen
2  ;; mit Initialwert 0
3  (def accounts (mapv ref (repeat 6 0)))
4
5  ;; Funktion, die eine Transaktion ausfuehrt.
6  ;; Es wird 10-110ms gewartet, dann wird die
7  ;; Referenz 'source' dekrementiert. Nach genau
8  ;; 100ms wird die Referenz 'target' inkrementiert.
9  (defn transfer [source target]
10   (dosync
11     (Thread/sleep (+ 10 (rand-int 100)))
12     (alter source dec)
13     (Thread/sleep 100)
14     (alter target inc)))
```

Eine Schleife gibt den Zustand des Programms aus, um das Verhalten nachzuvollziehen.

```
;; Gibt alle 50ms die einzelnen Kontostaende
;; und die Summe aus. Es wird erwartet, dass die
;; Summe zu jedem Zeitpunkt 0 ist.
(defn print-status-loop []
  (doseq [i (range)]
    (dosync
      (let [balances (map deref accounts)]
        (println
          (format "[%3d]" i) "sum" balances "=" (apply + balances))))
    (Thread/sleep 50)
    (recur (inc i))))

;; Starte Thread
(future (print-status-loop))
```

Die Form `(future & body)` führt die in `body` spezifizierten Ausdrücke in einem getrennten Thread aus. Zuletzt werden die nebenläufigen Transaktionen gestartet

```
;; Fuehre 100 parallele Transfer-Transaktionen
;; zwischen zwei zufaellig gewaehlten Referenzen aus
(doseq [i (range 100)]
  (future (transfer (rand-nth accounts) (rand-nth accounts))))
```

Die Ausgabe zeigt, dass die Transaktions-Kriterien immer eingehalten wurden

```
[  0] sum ( 0  0  0  0  0  0) = 0
[  1] sum ( 0  0  0  0  0  0) = 0
[  2] sum ( 0  0  0  0  0  0) = 0
[  3] sum ( 0  0  0  0  0  0) = 0
[  4] sum (-1  1  0  1 -1  0) = 0
[  5] sum (-1  1 -1  1 -1  1) = 0
[  6] sum (-1  1 -1  1 -1  1) = 0
[  7] sum (-1  1 -1  1 -1  1) = 0
[  8] sum ( 0  0 -2  1  0  1) = 0
[  9] sum ( 0  0 -2  1  0  1) = 0
[ 10] sum ( 0  0 -2  1  0  1) = 0
...
```

Wird statt `alter` die Form `commute` benutzt, erhöht sich die Nebenläufigkeit, da die Reihenfolge der Transaktionen nicht mehr berücksichtigt wird und Transaktionen nicht samt Wartezeit wiederholt werden müssen. Die Form `commute` ist dann sinnvoll, wenn Aggregate über Mengen gebildet werden und die Reihenfolge in der Menge unerheblich ist.

```
[  0] sum ( 0  0  0  0  0  0) = 0
[  1] sum ( 0  0  0  0  0  0) = 0
[  2] sum ( 0  0  0  0  0  0) = 0
[  3] sum (-1  0  1  0 -1  1) = 0
[  4] sum (-5 -1  1 -1  0  6) = 0
[  5] sum (-3 -3  1 -3  2  6) = 0
...
```

3.2 Sleeping Barber Problem

3.2.1 Beschreibung

Eine beliebte Klausuraufgabe und klassisches Problem der nebenläufigen Programmierung ist das „*Sleeping Barber Problem*" (Problem des schlafenden Frisörs), das dem Informatiker *Edsger Dijkstra* zugeschrieben wird. Das abzubildende Szenario ist folgendes:

> Es gibt einen Frisör-Salon mit einem Frisör. Der Frisör hat einen Frisierstuhl und ein Wartezimmer mit einer bestimmten Anzahl von Plätzen. Wenn der Frisör einen Kunden bedient hat, verlässt dieser den Salon. Der Frisör sieht dann im Wartezimmer nach, ob neue Kunden warten. Wenn dies zutrifft, bestellt er einen Kunden und setzt diesen auf den Frisierstuhl und scheidet ihm die Haare. Wenn dies nicht zutrifft, setzt er sich selbst auf den Frisierstuhl und schläft.
>
> Jeder Kunde, der den Frisör-Salon betritt, sieht nach was der Frisör macht. Wenn dieser schläft, weckt der Kunde diesen, setzt sich in den Frisierstuhl und lässt sich bedienen. Wenn der Frisör statt dessen beschäftigt ist, geht der Kunde ins Wartezimmer und prüft, ob ein Platz frei ist. Ist ein Platz frei, so setzt er sich in diesen und wartet bis er an der Reihe ist. Andernfalls verlässt er den Salon.

Diese Beschreibung sollte ausreichen, um einen ordnungsgemäß arbeitenden Frisör-Salon zu simulieren. In der Praxis müssen hier allerdings viele Randbedingungen betrachtet werden, welche die allgemeinen Probleme nebenläufigen Prozesssteuerung verdeutlichen.

Diese Probleme hängen alle damit zusammen, dass die beschriebenen Tätigkeiten eine unbestimmte Zeit in Anspruch nehmen (Betreten des Salons, Prüfen des Wartezimmers, Bedienen eines Kunden usw.). Ein Kunde kann gerade auf dem Weg ins ein Wartezimmer sein während der Frisör das leere Wartezimmer betrachtet hat und auf dem Weg zu seinem Frisörstuhl ist, um dort zu schlafen. In diesem Fall wartet der Frisör auf den Kunden und der Kunde wartet auf den Frisör, was zu einer Verklemmung der Prozesse führt.

3.2.2 Implementierung

Durch eine geschickte Definition einiger Transaktionen lässt sich dieses Problem in Clojure implementieren. Zuerst werden Referenzen auf den aktuellen Kunden und das Wartezimmer definiert

```
(def customer (ref nil))
(def waiting-room (ref ()))
(def waiting-room-size 10)
```

Es folgen die notwendigen Operationen, die Zugriffe und Änderungen in Transaktionen kapseln

```
;; Voraus-Deklaration
(def barber-check)

;; Nehme Kunden an - wenn i gleich
;; nil ist geht der der Friseur
;; schlafen, schaut also nicht
;; nach neuem Kunden
(defn barber-take-customer [i]
  (dosync (ref-set customer i))
  ;; wenn ein Kunde bedient wird,
  ;; sehe 100ms spaeter nach
  ;; einem neuen Kunden
  (when-not (nil? i)
    (future
      (Thread/sleep 100)
      (barber-check))))

;; Setze letzten Kunden in der
;; Warteschlange als neuen Kunden
;; Ist die Warteschlange leer, ist
;; (last @waiting-room) gleich nil
(defn barber-check []
  (dosync
    (barber-take-customer (last @waiting-room))
    (alter waiting-room drop-last)))
```

Eine gesonderte Funktion simuliert einen Kunden, der den Salon betritt

```
(defn salon-enter [i]
  (dosync
    (if (nil? @customer)
      ;; Falls kein kein Kunde bedient wird,
      ;; stoesst dieser Thread den Friseur an,
      ;; "weckt" ihn also auf
      (barber-take-customer i)
      ;; andernfalls wird geprueft ob noch
      ;; Platz im Wartezimmer ist und dort
      ;; gegebenenfalls Platz genommen
      (when-not (> (count @waiting-room) waiting-room-size)
        (alter waiting-room conj i)))))
```

Zuletzt wird eine Funktion definiert, die zyklisch Kunden in den Salons schicken, eine Funktion, die zyklisch den Status ausgibt, und eine Reihe von Threads wird erzeugt, die diese Funktionen aufrufen.

```
(defn customer-loop [prefix]
  (doseq [i (range)]
    (salon-enter (str prefix i))
    (Thread/sleep (rand-int 500))))

(defn status-loop []
  (doseq [i (range)]
    (dosync
      (println "customer" @customer "waiting-room" @waiting-room))
    (Thread/sleep 40)))

(future (status-loop))
(future (customer-loop "a"))
(future (customer-loop "b"))
(future (customer-loop "c"))
```

Die Transaktionen Kapseln den Zugriff auf die Referenzen `waiting-room` und `customer` durch einen Kunden oder durch den Friseur. Die in `salon-enter` abgebildete Aktivität, die in der Transaktion abgewickelt wird, besteht aus

1. Wenn kein Kunde bedient wird, Frisör wecken und bedienen lassen
2. Wenn bereits ein Kunde bedient wird
   a) Wenn das Wartezimmer nicht voll ist, dort Platz nehmen
   b) Andernfalls den Salon verlassen

Wenn mehrere Kunden gleichzeitig das Wartezimmer betreten wollen, so wie es durch die drei mit `future` erzeugten Threads provoziert wird, wird versucht, alle Transaktionen gleichzeitig abzuwickeln. Im Laufe der Transaktion wird durch die Ausdrücke `(if (nil? @customer))` und `(count @waiting-room)` zwei Referenzen ausgelesen. Diese Werte können im Laufe der Transaktion durch andere Transaktionen überschrieben werden. In diesem Fall wird die Transaktion abgebrochen und mit neuen Werten wiederholt.

Dadurch kann es etwa nicht vorkommen, dass zwei Kunden gleichzeitig ankommen, sehen dass der Friseur schläft, und gleichzeitig aufwecken, da die Funktion, die den Friseur weckt, die Referenz `customer` überschreibt und den Neustart der anderen Transaktionen veranlasst. Gleiches gilt sowohl für die Referenz `waiting-room`, die durch `(alter waiting-room)` verändert wird, als auch für die in der Funktion `barber-check` definierte Transaktion, die `waiting-room` dereferenziert und gleichzeitig bedingt verändert.

Innerhalb der Transaktion werden also Änderungen vorgenommen, die auf bestimmten Annahmen beruhen. Das transaktionale Speicher-System sorgt dafür, dass diese Annahmen bestehen bleiben, bis die Änderungen übernommen sind. Wenn die Annahmen einer Transaktion nicht mehr gültig sind, wird diese unter den neuen Umständen wiederholt.

3.3 Philosophenproblem

3.3.1 Beschreibung

Ein weiteres Problem aus dem Bereich der nebenläufigen Programmierung ist das Philosophenproblem.

> An einem Runden Tisch sitzen fünf Philosophen, die abwechselnd denken und essen. Jeder Philosoph hat einen Teller mit einer unbegrenzten Menge von Spaghetti vor sich. Rechts neben dem Teller liegt eine Gabel. Zum Essen werden zwei Gabeln benötigt, also wird zum essen die Gabel des linken Nachbarn benötigt. Wenn der Philosoph satt ist, legt er die Gabel zurück auf den Tisch und denkt nach.

Bei diesem Szenario kann es zum Beispiel zu einer Verklemmung kommen, wenn alle Philosophen gleichzeitig essen wollen. Jeder Philosoph nimmt dann die rechte Gabel in die Hand, wartet, bis er die linke Gabel bekommt und verhungert.

3.3.2 Implementierung

Zunächst werden die benötigten Datenstrukturen definiert. Philosophen und Gabeln werden durch Listen repräsentiert. Die Gabeln sind zyklisch angeordnet, so dass die rechte Gabel des letzten Philosophen die linke Gabel des Ersten ist. Des Weiteren werden einige Operationen für die Manipulation der Gabeln fest gelegt.

```
1  ;; Die Philosophen sind durch einen Zustand gekennzeichnet
2  ;; (initial :none)
3  (def philosophers (vec (map ref (repeat 5 :none))))
4
5  ;; Die Zugehoerigkeiten der Gabeln. Der Wert nil bedeutet
6  ;; frei (auf dem Tisch liegend), ein Zahlenwert entspricht
7  ;; dem derzeitigen Besitzer
8  (def forks (cycle (map ref (repeat 5 nil))))
9
10 ;; gibt Referenz auf rechte und linke Gabel am Platz i zurueck
11 (defn forks-get [i]
12   [(nth forks i) (nth forks (inc i))])
13
14 ;; setzt linke und rechte Gabel-Referenz auf Wert
15 (defn forks-set [i v]
16   (dosync (doseq [f (forks-get i)] (ref-set f v))))
17
18 ;; pruefe, ob Gabeln am Platz i frei sind
19 (defn forks-free? [i]
20   (dosync (every? nil? (map deref (forks-get i)))))
21
22 ;; wenn Gabeln frei sind, annehmen und true zurueck geben
23 (defn forks-pick? [i]
24   (dosync
25     (when (forks-free? i)
26       (forks-set i i) true)))
27
28 ;; linke und rechte Gabel am Platz i zurueck legen
29 (defn forks-put [i]
30   (forks-set i nil))
```

Operationen, in den mit `deref` oder `ref-set` auf Referenzen zugegriffen wird, werden hier mit `dosync` gekapselt.

Danach wird die eine Funktion definiert, die in einer Transaktion die Prüfung auf freie Gabeln, das Aufheben und Zurück-Legen der Gabeln, und das anschließende Nachdenken beschreibt. Zusätzlich wird der Zustand des Philosophen gesetzt.

```clojure
(defn eat-and-think [i eat-time think-time retry-time]
  (let [p (philosophers i)]
    (if (forks-pick? i)
      (do
        (dosync
          (ref-set p :eating))
        (Thread/sleep eat-time)
        (dosync
          (forks-put i)
          (ref-set p :thinking))
        (Thread/sleep think-time)
        (dosync
          (ref-set p :waiting)))
      (Thread/sleep retry-time))))
```

Anschließend wird für jeden Philosophen ein Thread erzeugt, der ihn unentwegt Essen und Denken lässt.

```clojure
(doseq [i (range 5)]
  (future (loop [] (eat-and-think i 100 100 10) (recur))))
```

Durch eine Ausgabe-Routine in einem weiteren Thread lässt sich darstellen, dass jeder Philosoph an die Reihe kommt und keine zwei benachbarten Philosophen gleichzeitig Essen

```
   0ms (:none      :none      :none      :none      :none      )
  50ms (:eating    :none      :eating    :none      :none      )
 100ms (:eating    :none      :eating    :none      :none      )
 150ms (:thinking  :eating    :thinking  :none      :eating    )
 200ms (:thinking  :eating    :thinking  :none      :eating    )
 250ms (:eating    :thinking  :waiting   :eating    :thinking  )
 300ms (:eating    :thinking  :waiting   :eating    :thinking  )
 350ms (:thinking  :waiting   :eating    :thinking  :eating    )
 400ms (:thinking  :waiting   :eating    :thinking  :eating    )
 450ms (:waiting   :eating    :thinking  :eating    :thinking  )
 500ms (:waiting   :eating    :thinking  :eating    :thinking  )
 [...]
```

4 Fazit

4.1 Zusammenfassung

Transaktionale Speicher-Systeme haben sich in Datenbanksystemen seit Jahrzehnten bewährt und finden zunehmend auch in gewöhnlichen Programmiersprachen Anwendung. Im Hinblick auf die Entwicklung nebenläufiger Systeme, die das Potenzial moderner Hardware ausnutzen können, und den andauernden Schwierigkeiten bei der Verwendung traditioneller Methoden zur Nebenläufigkeits-Kontrolle wie dem Locking, sind STMs eine wichtige Entwicklung.

Clojure ist eine lebendige, wachsende Sprache, die mit ihrem STM-System eine moderne, leicht verständliche und zuverlässige Alternative zu herkömmlichen Verfahren bietet. Ähnliche Ansätze finden sich auch in anderen funktionalen Programmiersprachen wie Haskell.

4.2 Ausblick

Im Paper *The Transactional Memory / Garbage Collection Analogy* von Prof. Dan Grossman[Gro07] wird die historische Bewertung von transaktionalen Speicher-Systemen mit früheren Vorbehalten gegenüber der heute der Heute weit verbreiteten Garbage Collection verglichen. Diese waren

- Viele dachten, Garbage Collection ist ohne Hardware-Unterstützung zu langsam
- Viele dachten, Garbage Collection wird in kürze die Welt erobern, Jahrzehnte bevor dies tatsächlich Eintrat
- Viele dachten, man bräuchte Wege um die Garbage Collection herum, wenn diese nicht präzise genug ist.

Es bleibt Abzuwarten, ob die Bedenken gegenüber STM-Systemen einen ähnlichen Entwicklung erfahren.

Literatur

[BG81] P.A. Bernstein und N. Goodman. "Concurrency control in distributed database systems". In: *ACM Computing Surveys (CSUR)* 13.2 (1981), S. 185–221.

[Bur00] R. Burstall. "Christopher Strachey—understanding programming languages". In: *Higher-Order and Symbolic Computation* 13.1 (2000), S. 51–55.

[Eme12] Chas Emerick. *Clojure programming*. Beijing Sebastopol, Calif: O'Reilly, 2012. ISBN: 1449394701.

[Gro07] D. Grossman. "The transactional memory/garbage collection analogy". In: *ACM SIGPLAN Notices* 42.10 (2007), S. 695–706.

[Har+05] Tim Harris u. a. "Composable memory transactions". In: *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '05. Chicago, IL, USA: ACM, 2005, S. 48–60. ISBN: 1-59593-080-9. DOI: 10.1145/1065944.1065952. URL: http://doi.acm.org/10.1145/1065944.1065952.

[Hic12] Rich Hickey. *Clojure Rationale*. 2012. URL: http://clojure.org/rationale.

[Jon07] Simon Peyton Jones. "Beautiful concurrency". In: *Microsoft Research* (2007). URL: http://research.microsoft.com/~simonpj/papers/stm/beautiful.pdf.

[McC62] J. McCarthy. *LISP 1.5 programmer's manual*. MIT press, 1962.

[Ora12] Oracle. *Synchronized Methods*. 2012. URL: http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html.

[SGG06] A. Silberschatz, P.B. Galvin und G. Gagne. *Operating System Principles*. John Wiley & Sons, 2006.

[Sut05] H. Sutter. "The free lunch is over: A fundamental turn toward concurrency in software". In: *Dr. Dobb's Journal* (2005). URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[Vol09] R. Mark Volkmann. *Software Transactional Memory*. 2009. URL: http://java.ociweb.com/mark/stm/article.html.