Lernskript Nebenläufige und Verteilte Programmierung
Wintersemester 2012/2013

Stefan Exner

Zusammenfassung

Dieses Skript stellt lediglich meine Zusammenfassung des Vorlesungsstoffes dar und erhebt keinerlei Anspruch auf Korrektheit oder Vollständigkeit!

Nomenclature

Monitor	Sammlung von Prozeduren und Datenstrukturen. Innerhalb des Monitors ist immer nur ein Thread gleichzeitig erlaubt.
Scheduling, faires	Jeder Thread muss nach endlicher Zeit wieder an die Reihe kommen
Scheduling, kooperatives	Ein Thread darf so lange rechnen, bis er die Kontrolle abgibt
Scheduling, präemptives	Der Scheduler kann Threads die Kontrolle selbstständig entziehen und an andere Threads vergeben. Dies führt zu einer gleichmäßigeren Verarbeitung unter den Threads
Semaphore	Integer und Prozesswarteschlange. Kann genutzt werden, um die Anzahl von Threads in einem Programmbereich zu beschränken.
Synchronisation, client-side	Synchronisation von Methoden oder Code-Abschnitten in Objektmethoden
Synchronisation, server-side	Synchronisation der Aufrufe eines Objekts, bspw. eine synchronisierte Liste

Kapitel 1

Nebenläufige Programmierung

Warum benötigt man nebenläufige Programmierung?

Reaktive Systeme (GUIs, Betriebssystemprogramme, ...) müssen auf Eingaben reagieren können.

Sequentieller Ansatz Busy Waiting, Eine Schleife kann alle möglichen Anfragen bearbeiten.

- Busy Waiting verbraucht Systemresourcen
- Die Schleife blockiert sobald eine Anfrage bearbeitet wird und muss für die nächste Anfrage "neu gestartet" werden.
- Der Anwendungscode ist schlecht strukturiert, da ja alle möglichen Anfragen in der Schleife bearbeitet werden können müssen.

Nebenläufiger Ansatz Mehrere Threads stellen die verschiedenen Schichten des Systems zur Verfügung (GUI, Calculation, ...).

- Kein Busy Waiting (da die Threads sich untereinander verständigen können)
- Bessere Code-Strukturierung
- Mehrere Anfragen können ggf. gleichzeitig bearbeitet werden

1.1 Kommunikation

1.2 Synchronisation

1.2.1 Semaphoren

Semaphoren bestehen aus einem Integer und einer zugeordneten Prozesswarteschlange. Sie können genutzt werden, um die Anzahl der Prozesse innerhalb eines bestimmten Programmabschnitts zu beschränken.

Nachteile von Semaphorennutzung:

- P / V Operatoren sind schwer zuordbar, da sie sich zwar wie Klammern verhalten, dies jedoch im Code selbst nicht ersichtlich ist.
- Codestrukturierung ist erschwert
- Fehler sind leicht durch falsche Reihenfolge der Operationen möglich (Deadlocks, etc)

Kapitel 2

Nebenläufige Programmierung in verschiedenen Programmiersprachen

2.1 Realisierung der Nebenläufigkeit

2.1.1 Java

Zur Realisierung der Nebenläufigkeit stehen 2 Optionen zur Verfügung:

- Extenden der Thread-Klasse
  - Code innerhalb der "run"-Methode der neuen Klasse
  - Anlegen von neuen Threads über sehr schnellen/kleinen Konstruktur, der nur die Attribute setzt
  - Ausführen des Threads mit der "start()"-Methode
- Implementieren des "Runnable"-Interfaces
  - Nötig, wenn die neue Klasse von einer anderen Klasse erben soll (in Java ist keine Mehrfachvererbung möglich)
  - Der aktuelle Thread kann selbst innerhalb der run-Methode nicht mehr über `this` ermittelt werden (dies liefert ein "Runnable"-Objekt), sondern über `Thread.currentThread()`.

2.1.1.1 Threads

Interface eines Thread-Objekts

Name Name des Threads, z.b. "Thread-0", "Thread-1". Kann über `getName()` / `setName(val)` erreicht werden.

Priority Priorität des Threads für den Scheduler. Wichtig: Auf die Priorität eines Threads kann man sich nicht verlassen, ein Thread mit der Priorität 10 wird nicht zwingend vor einem mit niedriger Priorität bearbeitet.

Threadgruppe Mehrere Threads können zu einer Gruppe zusammengefasst werden um zu signalisieren, dass sie möglichst zeitnah verarbeitet werden sollen.

Zustand Ein Thread kann folgende Zustände haben:

erzeugt Der Thread wurde über `new Thread()` erzeugt

aktivierbar Der Thread wurde gestartet und ist bereit zu rechnen

terminiert Die run-Methode wurde beendet

nicht aktivierbar Der Thread wartet auf eine Eingabe oder ist suspendiert

Ein Thread-Objekt bleibt so lange erhalten wie es referenziert wird. Während der Ausführung existiert eine "Eigenreferenz".

isAlive() testet, ob der aktuelle Thread noch läuft.

Beenden von Threadausführungen

1. Wenn die Run-Methode durchgelaufen ist, wird der Thread beendet
2. Die Run-Methode kann abgebrochen werden
3. Aufruf der `destroy()`-Methode

1. und 2. geben alle Locks des Threads wieder frei, bei 3. wird der Thread einfach "abgeschossen", ohne jedwede Aufräumarbeit durchzuführen. Zudem ist 3. deprecated.

Unterbrechen von Threads Jeder Thread besitzt die Methode `interrupt()`, welche ein Flag innerhalb des Threads setzt. Ist der Thread aktuell suspendiert (also während `sleep` oder `wait()`), wird eine InterruptedException geworfen. Um zu testen, ob ein Thread unterbrochen wurde, existieren zwei Methoden:

interrupted() Liest das Interrupted-Flag des Threads aus und löscht dieses. Soll also noch weiter auf das Flag reagiert werden, muss dieses selbst neu gesetzt werden.

isInterrupted() Liest das Interrupted-Flag des Threads aus, ohne es zu verändern.

Warten auf Terminierung eines Threads Die `join()` Methode eines Threads sorgt dafür, dass der ausführende Thread so lange wartet, bis der Thread, auf dem die Methode aufgerufen wurde terminiert.

Die Implementierung von `join()` kann man sich so vorstellen, dass ähnlich der MVar-Implementierung bei Thread-Terminierung alle Threads aufgeweckt werden, die vorher auf dem Threadobjekt suspendiert sind, jedoch auf *isAlive()* statt auf *empty* geprüft wird.

2.1.2 Haskell

In Haskell können neue Threads mit der Funktion `forkIO` gestartet werden, der ein abzuarbeitender Programmverweis mitgegeben wird.

2.2 Kommunikation

Die Kommunikation zwischen Threads finden größtenteils über geteilte Variablen statt. Daher muss bei nebenläufiger Programmierung geachtet werden, dass das Scheduling keine Auswirkungen auf das Ergebnis von Berechnungen hat. Standardbeispiel mit den möglichen Ausgaben 0, 1 und 2:

```
1  int i = 0;
2  par { i := i + 1 }
3      { i := i * 2 }
4  write(i);
```

2.2.1 Java

Kommunikation über geteilte Objekte Problematisch ist hier vor allem, festzustellen, wann ein erwartetes Ereignis eingetreten ist (wann z.B. ein Wert verändert wurde).

Zwei Möglichkeiten:

2.2.1.1 Wait/Notify

Verhalten von wait() und notify():

wait() Suspendiert den ausführenden Thread und gibt den Objektlock frei

notify() Erweckt einen beliebigen Thread des Lock-Objektes und fährt mit der eigenen Berechnung fort.

2.2.2 Erlang

Nachrichtenabarbeitung in Erlang Nachrichten werden in Erlang durch `receive` folgendermaßen abgearbeitet:

- Erste Nachricht in der Mailbox (zeitlich gesehen) wird auf alle Pattern getestet.
  - Passt ein Pattern, wird die Nachricht verarbeitet
  - Passt kein Pattern, wird die Nachricht in der Mailbox belassen und mit der nächsten Nachricht weitergemacht.

2.3 Synchronisierung

Semaphoren bestehen aus einem Integer und einer zugeordneten Prozesswarteschlange. Sie können genutzt werden, um die Anzahl der Prozesse innerhalb eines bestimmten Programmabschnitts zu beschränken.

Nachteile von Semaphorennutzung:

- P / V Operatoren sind schwer zuordbar, da sie sich zwar wie Klammern verhalten, dies jedoch im Code selbst nicht ersichtlich ist.
- Codestrukturierung ist erschwert
- Fehler sind leicht durch falsche Reihenfolge der Operationen möglich (Deadlocks, etc)

2.3.1 Java

Synchronisierung über Objekt-Locks

- Methoden eines Threads können als "synchronized" markiert werden.
  - In allen "synchronized"-Methoden eines Objekts darf sich maximal ein Thread aufhalten
  - Durch "sleep" oder "yield" wird die "synchronized"-Methode nicht verlassen
  - Jedes Objekt besitzt ein Lock, bei "synchronized"-Methoden wird automatisch "this" als Lock-Objekt verwendet.
  - Beim betreten einer "synchronized-Methode" wird wie folgt vorgegangen:
    * Lock verfügbar: Lock nehmen und fortfahren
    * eigener Thread hat Lock bereits: fortfahren
    * lock nicht verfügbar: auf Lock suspendieren.
- Bei Verwendung von `synchronized(myObject)` können Blöcke über beliebige Lock-Objekte synchronisiert werden.

2.4 Gefahren bei der Programmierung mit Locks

- Zu wenig Locks -> Inkonsistenzen
- Zu viele Locks -> übermäßige Sequentialisierung oder sogar Deadlocks
- Nehmen von Locks in falscher Reihenfolge -> Deadlocks
- Robustheit -> Nimmt ein Prozess einen Lock und stürzt dann ab, kann es zu inkosistenten zuständen kommen
- Vergessen von Lockfreigaben -> Deadlock

2.5 STM (Software Transactional Memory)

Für STM werden statt MVars TVars benutzt. Eine TVar besteht aus zwei Kompontenten:

1. Einem Wert. Dieser muss vorhanden sein, eine TVar darf nicht leer sein.
2. Einer Versionsnummer. Diese wird immer dann erhöht, wenn die TVar beschrieben wurde

Innerhalb einer STM-Aktion werden Werte noch nicht endgültig in TVars geschrieben, sondern in einem internen Speicher gehalten. Erst, wenn die gesamte Funktion ordnungsgemäß durchgelaufen ist, werden die Werte tatsächlich in die TVars geschrieben (vorausgesetzt, die Validierung war erfolgreich). Der Ablauf einer STM-Aktion ist folgendermaßen:

1. Die eigentliche Funktion läuft durch
   (a) Wenn TVars gelesen werden, wird die aktuelle Versionsnummer intern vermerkt
   (b) Wird eine TVar beschrieben, wird der Wert vorerst in einen internen Speicher geschrieben.
   Wichtig: wird eine TVar gelesen, für die bereits ein Schreibzugriff erfolgte, wird ihr aktueller Wert aus dem internen Speicher gelesen.
2. Alle TVars, die im Laufe der Abarbeitung der Funktion gelesen oder geschrieben wurde werden gelockt.
3. Die Versionsnummern aller TVars, die bereits gelesen wurden werden abgerufen.
   (a) Hat sich in der Zwischenzeit eine Versionsnummer verändert, wird ein Rollback gestartet und alle TVars wieder freigegeben.
   (b) Sind alle Versionsnummer unverändert, werden die Werte des WriteSets in die entsprechenden TVars geschrieben.

Es gibt zudem eine alternative Vorgehensweise zum obigen *optimistic locking* (also der Annahme, dass schon niemand anders die TVars verändert haben wird), das *pessimistic locking*. Hier wird eine TVar sofort gelockt, wenn sie lesend oder schreibend verwendet wird.

Zudem gibt es zwei weitere Methoden im Zusammenhang mit STM:

retry Retry kann innerhalb einer STM-Aktion verwendet werden und entspricht mehr oder weniger einem Rollback.

orElse Zwei STM-Aktionen können durch diesen Operator miteinander verknüft werden, schlägt die erste Aktion fehl, wird die zweite ausgeführt. Schlägt auch diese fehl, wird die gesamte Ausführung neu gestartet.

Kapitel 3

Wiederkehrende Probleme

3.1 Setzen und Ausgeben eines States

Eine Funktion setzt einen neuen Zustand, eine andere gibt ihn aus.

```
1  class C {
2    private int state = 0;
3    private boolean modified = false;
4
5    public synchronized void printNewState() {
6     while (!modified) {};
7     System.out.println(state);
8     modified = false;
9    }
10
11   public synchronized void setValue(int v) {
12     state = v;
13     modified = true;
14     System.out.println("value set");
15   }
16 }
```

Dieser Code kann zu den Ausgaben “42 value set” oder “value set 42” auswerten, da nicht gewährleistet ist, dass “setValue” zu Ende verarbeitet wird. Zudem wird in der printNewState()-Methode unnötig Prozessorzeit durch Busy Waiting vergeudet.

```
1  class C {
2    private int state = 0;
3
4    public synchronized void printNewState() {
5     wait();
6     System.out.println(state);
7    }
8
9    public synchronized void setValue(int v) {
10     state = v;
11   notify();
12     System.out.println("value set");
13   }
14 }
```

Diese Lösung führt zu einem Problem, wenn zunächst der schreibende Thread und dann der lesende Thread ausgeführt wird, da ein “notify” nicht gespeichert wird. Das “wait()” des Lese-Threads wird also niemals beendet. Anders ausgedrückt: Alle Nachrichten (Zustandsveränderungen) vor dem Aufruf von “wait()” gehen einfach verloren.

```
class C {
  private int state = 0;
  private boolean modified = false;

  public synchronized void printNewState() {
   if (!modified)
     wait();
   System.out.println(state);
   modified = false;
   notify();
  }

  public synchronized void setValue(int v) {
    if (modified)
      wait();
    state = v;
  notify();
    modified = true;
    System.out.println("value set");
  }
}
```

Es kann passieren, dass das notify() des schreibenden Threads einen weiteren schreibenden Thread aufweckt und somit das System einfriert.

```
class C {
  private int state = 0;
  private boolean modified = false;

  public synchronized void printNewState() {
   while (!modified)
     wait();
   System.out.println(state);
   modified = false;
   notifyAll();
  }

  public synchronized void setValue(int v) {
    while (modified)
      wait();
    state = v;
  notifyAll();
    modified = true;
    System.out.println("value set");
  }
}
```

Alle Threads werden jeweils aufgeweckt und ggf. wieder schlafen gelegt.

3.2 Einelementiger Puffer (MVar)

Die einfachste Variante des einelementige Puffers entspricht exakt dem Aufbau in 3.1, nur dass die Lesemethode den aktuell gespeicherten Wert zurückgibt. Dieser Aufbau ist jedoch für einen Synchronisationspunkt, der u.U. viele Threads gleichzeitig verwalten muss, ungünstig, da bei jeder Aktion alle Threads erweckt werden.

```
public class MVar<T> {
  private T content = null;
  private boolean full = false;
```

```
5     public MVar(){};
6     public MVar(T content) {
7       this.content = content;
8       this.full = true;
9     }

11    public synchronized T take() throws InterruptedException {
12      while(!full) this.wait();
13      full = false;
14      this.notify();
15      return content;
16    }

18    public synchronized void put(T entry) throws InterruptedException {
19      while(full) this.wait();
20      full = true;
21      content = entry;
22      this.notify();
23    }
24  }
```

Diese Implementierung ist fehlerhaft, da nur ein Lock-Objekt benutzt wird. Es kann somit vorkommen, dasss ein falscher Thread aufgeweckt wird (also ein lesender Thread einen weiteren lesenden Thread aufweckt).

Puffer mit zwei Locks Durch die Verwendung eines Lese- und eines Schreiblocks lassen sich gezielt Threads erwecken und so viel Busy-Waiting sparen. Der Grundaufbau der Methoden sieht folgendermaßen aus (mit Generic Types):

```
1   public T take() throws InterruptedException {
2     //Nur ein Reader darf gleichzeitig auf den Puffer zugreifen
3     synchronized(readObject) {
4       while (empty)
5         readObject.wait();
6       //Während des Lesevorgangs darf kein Schreiber auf den Puffer zugreifen
7       synchronized(writeObject) {
8         empty = false;
9         writeObject.notify();
10        return content;
11      }
12    }

14  public void put(T o) throws InterruptedException {
15    synchronized(writeObject) {
16      if(!empty)
17        writeObject.wait();
18      synchronized(readObject) {
19        empty = false;
20        content = 0;
21        readObject.notify();
22      }
23    }
24  }
```

Obwohl die zwei Funktionen die Locks in unterschiedlicher Reihenfolge nehmen, kann **kein Deadlock entstehen**, da die Überprüfung auf "empty" das beidseitige Aufnehmen der Locks verhindert (die andere Funktion kann niemals beide Locks anfordern).

Die `while`-Schleifen können nicht durch `if`-Blöcke ersetzt werden, da folgendes Szenario möglich wäre:

1. Der Puffer ist gefüllt, ein schreibender Thread suspendiert bei `writeObject.wait();`
2. Ein lesender Thread durchläuft die `take()`-Methode und erweckt den suspendierten Thread. Dieser steht nun hinter `writeObject.wait();`, muss sich jedoch vor der Weiterbearbeitung erst einmal erneut um den Lock für `writeObject` bewerben (Semantik von notify()).

3. Ein neuer schreibender Thread bewirbt sich in der Zwischenzeit um den Lock für `writeObject`, erhält ihn und durchläuft die gesamte `put()`-Funktion, setzt *empty* auf *false*.

4. Der schreibende Thread erhält nun den Write-Lock und überprüft nicht erneut, ob die MVar in der Zwischenzeit neu gefüllt wurde (`if` statt `while`). Somit beschreibt der Thread die MVar neu und der Wert, der inzwischen geändert wurde, wird einfach überschrieben.

Dies funktioniert analog natürlich auch beim Lesen der MVar (der Wert wird zweimal gelesen).

3.2.1 MVar in Erlang

Die MVar wird in Erlang als eigenständiger Prozess umgesetzt, der je nachdem, ob die MVar gefüllt ist oder nicht, zwischen zwei Zuständen hin- und herwechselt (FSM).

```
1  mVar(empty) ->
2    receive
3      {put, V, Pid} -> Pid!put, mVar({full, V}
4    end;
5  mVar({full, V}) ->
6    receive
7      {take, Pid} -> Pid!{took, V}, mVar(empty)
8    end.

10 take(MVar) ->
11   MVar!{take, self()},
12   receive
13     {took, V} -> V
14   end.

16 put(MVar, V) ->
17   MVar!{put, V, self()},
18   receive
19     put -> ok
20   end.
```

3.2.2 MVars mit STM

Eine MVar kann durch eine TVar vom Typ Maybe dargestellt werden. Die MVar-Funktionen können anschließend ganz naiv aufgebaut werden:

newMVar erzeugt eine neue TVar mit dem Wert *Nothing*

takeMVar Schreibt *Nothing* in die TVar und gibt ihren Wert zurück. War kein Wert vorhanden, wird einfach ein *retry* ausgeführt

putMVar Ist bereits ein Wert vorhanden, wird ein *retry* ausgeführt. Ansonsten wird der neue Wert in die TVar geschrieben

readMVar Da alles innerhalb einer Transaktion ausgeführt wird, kann für diese Aktion einfach der aktuelle Wert aus der TVar genommen und gleich wieder zurückgeschrieben werden (über *takeMVar* und *putMVar*). Schlägt eine der beiden fehl, wird automatisch die Transaktion neu gestartet.

3.2.3 MVar-Implementierung durch Channels

Durch die Verwendung von Channels kann das Message-Passing-Konzept von Erlang (teilweise) simuliert werden, um eine MVar in anderen Sprachen analog aufzubauen. Jede MVar verfügt über zwei Channel:

1. Einen Channel, in den sich lesende Threads eintragen und einen Channel für die Rückantwort mitschicken

2. Einen Channel für die schreibenden Threads, in den sie den neuen Wert und einen Antwortchannel schreiben (in den die MVar dann im Prinzip das *ok* aus Erlang schreibt)

```
data MVar a = MVar (Chan (Chan a)) (Chan (a,Chan ()))
```

newEmptyMVar Erzeugt die beiden Channels und einen neuen Thread, der - je nach aktuellem Zustand - einen der beiden Channels bearbeitet.

takeMVar Ein Channel für die Antwort der MVar wird erzeugt und dann in den Lese-Channel der MVar eingetragen. Danach wird einfach nur darauf gewartet, dass die MVar den aktuellen Wert in den neuen Channel schreibt.

putMVar Ein neuer Channel wird die für Antwort erzeugt und der neue Wert zusammen mit diesem Channel in den Schreib-Channel der MVar geschrieben. Danach wird auf das *ok* für den erfolgreichen Schreibzugriff gewartet

3.3 Producer / Consumer

3.3.1 Allgemein

Produzenten und Konsumenten sind Threads und tauschen ihre "Waren" über einen geteilten Speicherpunkt, für gewöhnlich einen Puffer aus.

Benötigte Semaphoren:

1. Ein "Zähler" für die aktuell bereitstehenden Waren. Konsumenten dekrementieren und Produzenten inkrementieren.
2. Ein Synchronisationspunkt für den Buffer-Access

```
semaphor num     := 0;
sempahor baccess := 1;

//Producer
while (true) {
  produce product;
  P(baccess);
  push product to buffer;
  V(baccess);
  V(num);
}

//Consumer
while (true) {
  P(num);
  P(baccess);
  //Hier wird nicht direkt konsumiert um den
  //Bereich nicht unnötig lange zu sperren.
  pull production from buffer;
  V(baccess)
  consume product;
}
```

3.4 Die dinierenden Philosophen

5 Philosophen sitzen um einen Tisch, auf dem sich 5 Stäbchen befinden. Wenn ein Philosoph hungrig ist, nimmt er nacheinander das Stäbchen links und rechts von ihm, isst und legt beide Stäbchen zurück.

3.4.1 Implementierung mit Semaphoren

Naive Implementierung

```
while (true) {
  think();
  get_hungry();
  P(stick[i]);
  P(stick[(i+1)%n]);
  eat();
  V(stick[i]);
  V(stick[(i+1)%n]);
}
```

Problem: Deadlock Nehmen alle Philosophen gleichzeitig ihr Stäbchen auf fährt sich das System fest, niemand kann essen.

Implementierung mit Zurücklegen eines Stäbchens

```
while (true) {
  think();
  get_hungry();
  P(stick[i]);
  if (stick[(i+1)%n] == 0)
    V(stick[i]);
  else {
    P(stick[(i+1)%n]);
    eat();
    V(stick[i]);
    V(stick[(i+1)%n]);
  }
}
```

Problem: Livelock Legt ein Philosoph sein Stäbchen zurück, wenn das andere Stäbchen nicht verfügbar ist, kann es passieren, dass er ebenfalls niemals zum Essen kommt, da er nur mit Aufnehmen/Ablegen beschäftigt ist.

Andere Methoden zur Deadlockvermeidung

- Ein Philosoph nimmt immer zuerst das linke Stäbchen, alle anderen zuerst das rechte
- Die Philosophen nehmen immer abwechselnd das linke oder rechte Stäbchen zuerst (der erste Philosoph nimmt zuerst das rechte, der zweite Philosoph zuerst das linke, ...)

3.4.2 Implementierung in Erlang

Die Stäbchen werden in der Erlang durch eigene Prozesse realisiert, die jeweils zwei Zustände besitzen, zwischen denen sie wechseln, wenn das Stäbchen aufgenommen oder abgelegt wird.

```
stickDown() ->
  receive
    {take, P} -> P!took, stickUp();
    _         -> stickDown()
  end.

stickUp() ->
  receive
    put -> stickDown()
  end.

take(St) ->
  St!{take, self()},
  receive took -> ok end.

put(St) -> St!put.
```

Dadurch, dass in den beiden Zuständen unterschiedliche Nachrichten bearbeitet werden, ist gewährleistet, dass ein Stäbchen immer nur von einem Philosophen gleichzeitig benutzt werden kann. Zudem garantiert der `receive`-Befehl in der *take*-Methode, dass der entsprechende Philosoph suspendiert, bis das von ihm gewünschte Stäbchen wieder verfügbar ist.

3.4.3 Implementierung mit STM

Die Stäbchen entsprechen jeweils einer TVar vom Typ Bool. Ist sie mit True gefüllt, liegt das Stäbchen auf dem Tisch.

Die Aktionen zum Aufnehmen und Ablegen der Stäbchen können ganz naiv implementiert werden:

putStick Der Wert *True* wird in die TVar geschrieben. Eine Überprüfung ist nicht nötig, da der Philosoph das Stäbchen ja bereits in der Hand hat.

takeStick Der aktuelle Wert der TVar wird geprüft. Ist das Stäbchen bereits belegt, wird ein *retry* ausgeführt, wenn nicht, wird *False* in die TVar geschrieben.

Nun müssen in der *phil()*-Methode lediglich die Aktionen zum Aufnehmen beider Stäbchen zu einer atomaren Aktion zusammengefasst werden, so dass es zu keinen Deadlocks mehr kommen kann: `atomically (takeStick l; takeStick r)`

3.5 Channel

Ein Channel (also eine Warteschlange, bei der am einen Ende gelesen und am anderen geschrieben wird) kann durch Verwendung von MVars als einfach verkettete Liste realisiert werden. Folgendes wird benötigt:

- Eine ChanElem-Klasse, die die einzelnen Kettenglieder repräsentiert. Die Instanzen benötigen lediglich einen Wert und einen Pointer auf das nächste Kettenglied. In Java:

```
1 private class ChanElem<T> {
2   private T value;
3   private MVar<ChanElem<T>> next;
4   ...
5 }
```

- Eine MVar für jedes Chan-Element, so dass lesende Threads suspendieren, wenn das aktuelle Kettenglied (noch) keinen Wert enthält. In Java: `MVar<ChanElem<T>>`
- Eine MVar für das leere Chan-Element (*hole*). Das Schreibende zeigt immer auf dieses Element: `MVar<ChanElem<T>> hole = new`
- Zwei MVars, die auf das Lese- und Schreibende des Chans zeigen. Diese werden hauptsächlich genutzt, um lesende und schreibende Threads zu synchronisieren (nur ein Leser/Schreiber gleichzeitig). Zu Anfang zeigen beide Enden auf das leere Kettenglied (leerer Chan):

```
1 read  = new MVar<MVar<ChanElem<T>>>(hole);
2 write = new MVar<MVar<ChanElem<T>>>(hole);
```

3.5.1 Lesen vom Channel

1. *take()* auf dem Leseende. Dies führt dazu, dass immer nur ein Thread gleichzeitig Lesen kann
2. *take()* auf dem Element, auf das das Leseende zeigt. Sollte dies leer sein, suspendiert der Leser
3. Das Leseende wird auf das nächste Kettenglied gesetzt
4. Der gelesene Wert wird zurückgegeben

3.5.2 Schreiben auf den Channel

1. Ein neues *hole* wird erstellt, also ein leeres Chan-Element, welches das neue Schreibende darstellt
2. *take()* auf dem aktuellen Schreibende. Dies führt dazu, dass immer nur ein Thread gleichzeitig schreiben kann.
3. Das aktuelle *hole* wird mit dem zu schreibenden Wert und einem Pointer auf das neue *hole* gefüllt
4. Das Schreibende wird auf das neue *hole* gesetzt.

3.5.3 Prüfung, ob der Channel leer ist

```
1 public boolean isEmpty() {
2   MVar<ChanElem<T>> rEnd = read.take();
3   MVar<ChanElem<T>> wEnd = write.take();
4   write.put(wEnd);
5   read.put(rEnd);
6   return (wEnd == rEnd);
7 }
```

Im Falle eines leeren Channels kann es dazu kommen, das isEmpty() nicht sofort den gewünschten Wert zurückliefert, sondern suspendiert. Dies passiert, wenn ein anderer Thread gleichzeitig versucht, vom leeren Channel zu lesen und somit das Readlock an sich nimmt. isEmpty() muss nun so lange warten, bis dieser lesende Thread seinen Lesevorgang abgeschlossen hat.

Lösung mit Semaphore Eine funktionierende Lösung wäre es, im Channel einen Counter in Form einer Semaphore zu halten. Dieser wird bei Schreibvorgängen erhöht und bei Lesevorgängen erniedrigt. Bei *isEmpty()* muss dann lediglich abgefragt werden, ob der aktuelle Zählerstand gleich 0 ist.

3.5.4 Größenbeschränkter Channel

Ein Channel kann analog zur Leerheitsprüfung (siehe 3.5.3) einfach größenbeschränkt werden, indem eine weitere Semaphore genutzt wird. Im Gegensatz zum Counter wird sie jedoch mit der maximalen Element-Anzahl initialisiert. Nun muss diese Semaphore nur noch genau gegenläufig zum Counter genutzt werden, also erhöht werden, wenn ein Element entfernt wurde.

3.5.5 Über TCP beschreibbarer Channel (RemoteChan, Erlang)

Es gibt (mindestens) zwei Möglichkeiten, Channels über Remote beschreibbar zu machen:

1. Jeder Channel öffnet einen Port über ein ListenSocket und kann somit Anfragen auch übers Netzwerk beantworten (bzw. er startet einen Hilfsthread, der Netzwerkanfragen verwaltet)
2. Es gibt einen Channel-Manager pro Knoten, der alle Anfragen für beliebige Channels entgegennimmt und an die entsprechenden Channels weiterleitet.

In den Übungsaufgaben haben wir die zweite Möglichkeit gewählt. Der Channel-Manager ist für folgende Aufgaben zuständig:

- Starten des ListenSockets und speichern der Socket-Informationen
- Starten neuer Channels und Vergabe von einzigartigen IDs an diese (so dass ein Channel pro Knoten mit einem eindeutigen Namen angesprochen werden kann).
- Eintragen von Channels in eine globale Registry
- Entgegennehmen neuer Schreibaufträge in einem Extra-Thread und Weiterleitung dieser an die lokalen Chans

In der write-Funktion eines Channels muss danach nur noch unterschieden werden, ob es sich um einen lokalen oder einen remote-Channel handelt.

```
write({chan, _ID, _Manager, MVar}, Value) -> mVar:put(MVar, Value);
write({remote_chan, ID, Host, Port}, Value) ->
  {ok, Sock} = gen_tcp:connect(Host, Port, [binary, {packet, 0}, {active, false}]),
  ok = gen_tcp:send(Sock, term_to_binary({write, ID, Value})),
  ok = gen_tcp:close(Sock).
```

Soll der Channel auch über TCP lesbar sein muss zusätzlich auf dem Knoten, auf dem der lesende Prozess läuft, ein weiterer TCP-Listener laufen, damit der lesende Prozess korrekt suspendiert.

3.6 Verbindung über TCP

Server

1. Ein ListenSocket wird unter Angabe eines Ports erstellt. Dieser Port wird danach verwendet, um den Kontakt mit dem Server herzustellen. Ein Beispiel hierfür ist Port 80 bei Web-Servern, obwohl die eigentliche Kommunikation später über einen anderen Port abläuft
2. Verbindet sich ein Client mit dem Server (unter Nutzung des in 1. gesetzten Ports) wird ein neues Socket erstellt (*listenSocket.accept()*), welches einen automatisch vergebenen Port benutzt.
3. Das unter 2. erstellte Socket stellt die direkte Verbindung zum Client dar und kann (je nach Einstellung) bidirektional genutzt werden, während (wiederum je nach Implementierung) das ListenSocket bereits die nächste Verbindung herstellen kann
4. Die Verbindung kann Serverseitig geschlossen werden

Client

1. Ein Socket wird erstellt, welches eine Verbindung zum ListenPort des Servers herstellt und von diesem automatisch den zu nutzenden Port erhält
2. Das Socket kann nun bidirektional genutzt werden
3. Die Verbindung kann Clientseitig geschlossen werden.

3.6.1 TCP in Erlangs

Vorgehen für Server

1. Beim Start des Servers wird ein ListenSocket erstellt.

   ```
   {ok, LSock} = gen_tcp:listen(Port, [binary, {packet, 0}, {active, false}])
   ```

2. Ein Thread (z.B. der Hauptthread) des Servers befindet sich dauerhaft in einer Schleife, die lediglich dazu genutzt wird, einkommende Verbindungen auf dem ListenSocket anzunehmen und neue Threads für deren Verarbeitung zu spawnen.

   ```
   database_listener(LSock, Database) ->
     case gen_tcp:accept(LSock) of
       {ok, Sock} -> spawn(database, network_database, [Sock, Database]),
                     database_listener(LSock, Database);
       Other -> throw("Accept returned " ++ base:show(Other))
     end.
   ```

3. Der neu erstellte Thread nimmt so lange Daten entgegen, bis die Verbindung geschlossen wurde. Im folgenden Beispiel sind Empfang und Verarbeitung der Daten aus Gründen der Übersichtlichkeit getrennt.
   Die Daten werden in Erlang als Binärdaten empfangen, die erst durch Hilfsmethoden wieder verarbeitet werden können.

```
network_database(Sock, Database) ->
  case gen_tcp:recv(Sock, 0) of
    {ok, Binary} -> handle_binary_request(Sock, Database, Binary);
    {error, closed} -> ok
  end.

handle_binary_request(Sock, Database, Binary) ->
  case binary_to_term(Binary) of
    {allocate, Key, Value} -> Database!{allocate, Key, Value},
                              gen_tcp:send(Sock, term_to_binary(ok)),
                              network_database(Sock, Database);
    {lookup, Key} -> Database!{lookup, Key, self()},
                     receive Ans -> gen_tcp:send(Sock, term_to_binary(Ans)) end,
                     network_database(Sock, Database)
  end.
```

In `handle_binary_request` sieht man zudem, dass die Verbindung bidirektional ist - der Server sendet ein *ok* zum Client.

Vorgehen für Clients

1. Ein neues Socket wird durch die Verbindung zum öffentlichen Port des Servers erstellt

   ```
   {ok, Sock} = gen_tcp:connect(Host, Port, [binary, {packet, 0}, {active, false}]),
   ```

   Das *active* Flag im Beispiel muss gesetzt werden, damit eine bidirektionale Verbindung erlaubt wird.

2. Über das neue Socket können Nachrichten gesendet und empfangen werden

   ```
   ok = gen_tcp:send(Sock, term_to_binary({lookup, Key})),
   {ok, Ans} = gen_tcp:recv(Sock, 0)
   ```

3. Nach Beendigung der Kommunikation sollte die Verbindung geschlossen werden

   ```
   ok = gen_tcp:close(Sock)
   ```

3.7 Serverloser Chat in Erlang

Die naive Lösung besteht darin, dass sich ein neuer Client bei jedem beliebigen vorhandenen Client anmelden kann und eine Liste der aktuellen Clients zurückerhält. Danach linkt er sich mit allen anderen Clients, damit er eine Benachrichtigung erhält, wenn einer von ihnen abstürzt oder den Chat verlässt. Abschließend broadcastet er seinen Login an alle existierenden Clients, damit diese ihn in ihre Liste aufnehmen können.

Dieses Vorgehen führt insbesondere zu einem Problem, wenn sich zwei Clients gleichzeitig bei unterschiedlichen Ansprechpartnern anmelden wollen. Sie erhalten dann lediglich eine Liste der aktuellen Clients, werden aber nicht über den jeweils anderen Neuankömmling benachrichtigt.

Dies kann dadurch behoben werden, dass jeweils ein Client aus der aktuellen Liste als Ansprechpartner für neue Clients ausgewiesen wird (z.B. der erste Client in der Liste, die nun natürlich sortiert gespeichert werden muss).

Ein Anmeldevorgang sieht dann folgendermaßen aus:

- Ein neuer Client sendet eine Login-Nachricht an einen beliebigen Client des Chats
  - Ist dieser der erste in der aktuellen Client-Liste, beantwortet er die Anfrage wie in der naiven Lösung
  - Ist er nicht der erste, sendet er eine Antwort mit dem ersten Client der Liste an den neuen Client. Dieser meldet sich dann dort an.

3.8 Linda

Das Linda-Modell (resp. die Sprache Linda) sieht einen unabhängigen Tupelraum vor, in den beliebige Clients Tupel schreiben und lesen können. Die zeitliche Abfolge, in der die Tupel im Tupelraum gelandet sind, ist hierbei egal. Linda sieht die folgenden Operationen vor:

out(t) Fügt das Tupel t in den Tupelraum ein. Berechnungen in t werden vor dem Einfügen ausgewertet (strikt).
Beispiel: `out(("hallo", 4, 1.5))` fügt das Tupel `("hallo", 4, 1.5)` in den Tupelraum ein

in(t) Entfernt das Tupel t aus dem Tupelraum, wenn es dort vorhanden ist. Wenn nicht, wartet es so lange, bis t im Tupelraum vorhanden ist.
Da nicht alle Elemente eines Tupels bekannt sein müssen, können auch Platzhalter verwendet werden, die dann an die aktuellen Werte gebunden werden (*pattern matching*).
Beispiel: `in(("hallo", ?i, ?f))` entfernt das obige Tupel und bindet $i = 4$ und $f = 1.5$

rd(t) wie *in(t)*, löscht das Tupel aber nicht aus dem Tupelraum

eval(stmt) Erzeugt einen neuen Prozess, der *stmt* auswertet.

inp(t), rdp(t) Wie *in(t)* und *rd(t)*, jedoch suspendiert der Thread nicht, wenn t nicht im Tupelraum vorhanden ist, sondern liefert false zurück.

Serveranwendungen mit Linda Bei Serveranwendungen muss darauf geachtet werden, dass Nachrichten beim richtigen Client ankommen. Eine einfache Lösung hierfür ist, dass der Server eindeutige IDs für die Kommunikation mit Clients benutzt. Da Server und Client beide die gleiche ID verwenden, können Tupel eindeutig zugeordnet werden.

1. Client fragt neue ID beim Server an
2. Server generiert neue ID und schreibt sie in den Tupelraum
3. Ein beliebiger Client, der gerade auf eine neue ID wartet, nimmt sie aus dem Tupelraum
4. Client schickt Anfrage an den Server. Diese enthält die vorher "ausgehandelte" ID
5. Server schickt Anfrage an den Client. Diese enthält ebenfalls die ID, so dass der Client Nachrichten, die an ihn gerichtet sind, aus dem Tupelraum entnehmen kann.

3.8.1 Linda in Erlang

Es ist in Erlang nicht möglich, Pattern zu verschicken. Aus diesem Grund werden stattdessen partiell definierte Funkionen in den Tupelraum eingetragen.

Der Tupelserver (resp. die Serverschleife) hält zwei Listen:

1. Die Liste der Tupel im Tupelraum
2. Die Liste der Anfragen (anonyme Funktionen), die noch nicht beantwortet werden können.

Wird eine neue Anfrage an den Server geschickt, wird diese (Funktion) auf alle Tupel angewandt, die sich aktuell im Tupelraum befinden. Ist die Funktion anwendbar (es wird kein Fehler geworfen), wird ihr Rückgabewert an den Anfrager zurückgeschickt.

Analog wird mit neuen Tupeln verfahren, nur dass dieses Tupel dann durch alle aktuell suspendierten Anfragen geschickt wird, bis eine passt. Ist dies nicht der Fall, wird das neue Tupel in die Tupelliste geschrieben.

3.8.2 Konto

Die Modellierung eines Kontos und der Transfer zwischen Konten ist nicht einfach zu modellieren.

Eine Lösung, die rein auf der Nutzung von MVars basiert stellt ein Konto einfach als eine *MVar Int* dar. Die Funktionen sehen folgendermaßen aus:

```
withdraw :: MVar Int -> Int -> IO ()
withdraw acc amount = do
  balance <- takeMVar acc
  putMvar acc (balance - amount)

deposit :: MVar Int -> Int -> IO ()
deposit acc amount = do
  withdraw acc (-amount)

balance :: MVar Int -> IO Int
balance acc = readMVar acc

-- Hebt nur Geld vom Konto ab, wenn genug vorhanden ist
limitedWithdraw :: MVar Int -> Int -> IO Bool
limitedWithdraw acc amount = do
  b <- balance acc
  if b >= amount
    then do
      withdraw acc amount
      return True
    else
      return False
```

Bei der letzten Funktion ist nicht garantiert, dass die Ausführung atomar stattfindet. Es kann also passieren, dass, nachdem der aktuelle Kontostand abgefragt wurde, jemand anderer Geld von diesem Konto abhebt. In diesem Fall kann der Kontostand negativ werden.

Eine Lösung hierfür wäre es, den Lock auf dem Konto über die gesamte Funktion zu halten. Dann aber wäre es nicht mehr möglich, die Funktion *withdraw* wiederzuverwenden (dies wäre in Java möglich, indem alle Methoden als *synchronized* gekennzeichnet werden).

Lösung mit TVars (STM) Diese Lösung sorgt dafür, dass bestimmte Aktionen atomar ausgeführt werden. Im folgenden wird nur die kritische *limitedWithdraw*-Methode betrachtet

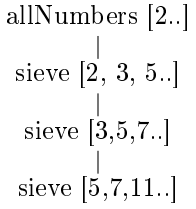
```
limitedWithdraw :: TVar Int -> Int -> STM Bool
limitedWithdraw acc amount = do
  b <- balance acc
  if b >= amount
    then do
      withdraw acc amount
      return True
    else return false
```

3.9 Das Sieb des Erathosthenes

In Erlang kann das Sieb des Eratothenes durch drei Prozessgruppen realisiert werden. Folgende Prozesse werden benötigt:

- Ein Prozess, der auf Anfrage alle natürlichen Zahlen ab 2 liefert
- Ein Prozess für jede Primzahl, die abgefragt wurde
- Ein Collector-Prozess, der die Anfrage nach einer neuen Primzahl beantwortet.

Jeder der Prozesse, die im zweiten Punkt genannt wurden, liefert lediglich eine neue Primzahl. Danach ist er dafür zuständig, alle Vielfachen seiner Primzahl aus den Zahlen herauszufiltern, die das Sieb der vorigen Primzahl ihm liefert. Der Aufbau ist also folgendermaßen:



Jedes Sieb kennt somit seinen Vorgängen und kann ihn nach der jeweils nächsten Zahl fragen. Dieser Vorgang geht jedes Mal zurück bis zum allNumbers-Generator.

Der Collector-Prozess muss immer nur das aktuelle Sieb kennen, welches automatisch angelegt wird, wenn das vorige Sieb eine Anfrage nach seiner Primzahl erhalten hat.

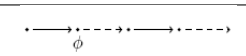
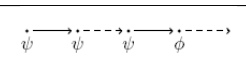
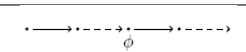
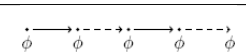
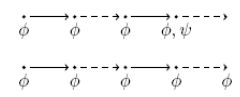
Kapitel 4

Linear Time Logic (LTL)

LTL ermöglicht es, Eigenschaften von Pfaden eines Systems auszudrücken. Ein System erfüllt dann eine LTL-Formel, wenn alle Pfade des Systems diese Formel erfüllen. Als Pfad wird hierbei ein möglicher Durchlauf durch das Programm bezeichnet.

4.1 LTL-Syntax und Abkürzungen

LTL-Formeln enthalten einfache Propositionen, ähnlich Atomen in Erlang. Diese sind letztendlich Bezeichner für einen bestimmten Zustand innerhalb des Programms.

Kurzform	Umformung	Erklärung	
$\neg\varphi$		Negation von LTL-Formeln	
$\varphi \wedge \psi$		Konjunktion	
$X\varphi$		φ muss ab dem nächsten Zustand gelten	
$\varphi U \psi$		φ gilt so lange, bis ψ gilt	
$false$	$\neg P \wedge P$	Boolscher Wert	
$true$	$\neg false$	Boolscher Wert	
$\varphi \vee \psi$	$\neg(\neg\varphi \wedge \psi)$	Disjunktion	
$\varphi \rightarrow \psi$	$\neg\varphi \vee \psi$	Implikation	
$F\varphi$	$true\, U\, \varphi$	Irgendwann gilt φ	
$G\varphi$	$\neg F \neg\varphi$	φ gilt immer (gilt niemals nicht)	
$F^{\infty}\varphi$	$GF\varphi$	φ gilt unendlich oft	
$G^{\infty}\varphi$	$FG\varphi$	φ gilt nur endlich oft nicht	
$\varphi W \psi$	$(\varphi U \psi) \vee (G\varphi)$	Schwaches Until	
$\varphi R \psi$	$\neg(\neg\varphi U \neg\psi)$	ψ gilt bis einschließlich zur ersten Position, an der φ gilt oder für immer, wenn eine solche Position nicht existiert.	

4.2 Kripke-Strukturen

Eine Kripkestruktur $K = (S, Props, \rightarrow, \tau, s_0)$ besteht aus

S Einer menge von Zuständen,

Props einer Menge von Propositionen

$\rightarrow$ einer Transitionsrelation

τ einer Beschriftungsfunktion für Zustände (die aktuells geltenden Propositionen) und

$s_o \in S$ einem Startzustand.

Ein Zustandspfad von K ist ein unendliches Wort $s_0 s_1 ...$ mit $s_i \rightarrow s_{i+1}$ und s_0 als Startzustand von K. Gelten in jedem Zustand ausschließlich Propositionen aus *Props* (genauer: ist das Ergebnis der Beschriftungsfunktion immer ein Element der Potenzmenge von *Props*), heißt dieser Zustandspfad ein *Pfad von K*.

Eine Kripke-Struktur erfüllt eine LTL-Formel genau dann, wenn alle Pfade der Kripke-Struktur die Formel erfüllen

4.3 Anwendungsgebiete von LTL

Im Wesentlichen können mit LTL drei unterschiedliche Eigenschaften von Programmen definiert werden:

Name	Übliche Form	Beschreibung	Beispiel
Safety	$G(\neg\varphi)$	Sicherheitskritische Situationen sollen programmweit ausgeschlossen werden	Für eine kritische Sektion soll gelten, dass immer nur ein Prozess gleichzeitig Zugriff haben soll. Für zwei Prozesse, die beim Betreten des Bereichs jeweils eine Proposition cs_i setzen, wäre $\phi = cs_1 \wedge cs_2$
Liveness	$G(F\varphi)$ oder $F\varphi$	Ein Programm soll (u.U. auch nur auf Anfrage) in jedem Fall reagieren	Nach einer Anfrage an einen Server soll dieser in endlicher Zeit eine Antwort schicken: $\varphi = G(req \rightarrow F\, answ)$
Fairness	$F^{\infty}\varphi$	Fairnesseigenschaften werden meist als Vorbedingung für eine Implikation benutzt, um unfaire Pfade auszuschließen	Jeder beteiligte Prozess soll unendlich oft (vom Scheduler) gewählt werden.

4.4 Implementierung von LTL in Erlang

Für die Implementierung von in Erlang sind folgende Grundfunktionen nötig:

- Eine Funktion, die die Negationen in einer LTL-Formel bis direkt vor Propositionen verschiebt (also nach innen zieht).
  Beispiele:

$normalize(\neg(\varphi \vee \psi))$	$normalize(\neg\varphi) \wedge normalize(\neg\psi)$
$normalize(\neg X\varphi)$	$X(normalize(\neg\varphi))$
$normalize(\neg F\varphi)$	$G(normalize(\neg\varphi))$
$normalize(\neg G\varphi)$	$F(normalize(\neg\varphi))$

- Eine Funktion, welche die Gültigkeit einer gegebenen LTL-Formel anhand der aktuell im System vorhandenen Propositionen testet. Diese Funktion besitzt nur drei mögliche Rückgabewerte:
  - true
  - false
  - Eine neue Formel, falls die Gültigkeit im aktuellen Schritt noch nicht entscheidbar ist. $X\varphi$ kann bspw. erst im nächsten Schritt entschieden werden.
- Eine Schrittfunktion, die insbesondere $X\varphi$ auflöst. Ein Schritt wird immer dann gemacht, wenn eine Proposition zur Menge hinzugefügt oder aus ihr entfernt wird.
- Eine Funktion *analyze*, die erfüllte und widerlegte Formeln aus der Formelliste löscht.

4.5 Beweisbarkeit von LTL-Formeln

Bestimmte LTL-Formeln können durch Testen nicht bewiesen werden. Hierzu zählen z.B.

- $F\varphi$, da es passieren kann, dass φ sehr spät auftritt
- $G\varphi$, da eine Verletzung erst sehr spät auftreten kann
- F^∞ oder G^∞, diese können weder bewiesen noch widerlegt werden (logisch, da sie aus der Hintereinanderausführung von F und G bestehen)

Eine Möglichkeit, mit diesen Beschränkungen umzugehen, ist, F und G eine maximale Anzahl an Schritten zuzugestehen, in denen die Formel erfüllt, bzw. für die die Formel erfüllt sein muss.

4.5.1 Verifikation

LTL-Formeln sind für endliche Kripke-Strukturen beweisbar. Dies erfordert insbesondere:

- Möglichst wenige konkrete Werte
- Keine Datenstrukturen (da diese unendliche Wertebereiche zulassen)

Ein Weg der Entwicklung eines verifizierten Systems besteht darin, zuerst eine Spezifikation zu erstellen und gegen spezifizierte Eigenschaften zu verifizieren. Danach wird diese schrittweise zum tatsächlichen System verfeinert.

Ein anderer Weg ist die umgekehrte Richtung. Ein bereits bestehendes System wird so weit abstrahiert, dass eine endliche Kripke-Struktur entsteht. Insbesondere bedeutet dies, dass Programmwerte abstrahiert werden müssen (es können bspw. nicht alle natürlichen Zahlen als Werte angenommen werden, wohl aber zwei abstrakte Werte *gerade* und *ungerade*). Zusätzlich muss generell ein Wert *unentschieden* vorhanden sein, wenn eine Entscheidung über den abstrakten Werten nicht möglich ist (IF-Blöcke, es werden dann beide möglichen Pfade weiterbetrachtet).

Wird der letztere Weg gewählt, kann es passieren, dass nicht vorhandene LTL-Verletzungen angezeigt werden. Dies kann auf zwei Arten überprüft werden:

1. Der Pfad, welcher die LTL-Verletzung hervorgerufen hat, existiert auch in der konkreten Semantik (ohne abstrahierte Werte). Ist dies der Fall, verletzt er wirklich die LTL-Formel und ist fehlerhaft.
2. Der Pfad, der die LTL-Verletzung hervorgerufen hat, existiert in der konkreten Sematik gar nicht, sondern ist erst durch den Nichtdeterminismus der abstrakten Werte hervorgerufen worden. In diesem Fall muss man versuchen, die Abstraktion der Werte etwas zu verfeinern.