Brandenburgische
Technische Universität
Cottbus

Fakultät für Mathematik, Naturwissenschaften und Informatik

Diplomarbeit

Entwicklung eines Präprozessors für JavaFC

Matthias Brattig
Matrikelnummer: 2307885
E-Mail: brattmat@tu-cottbus.de

Cottbus, den 15. Februar 2010

vorgelegt bei
Prof. Dr.-Ing. Monika Heiner
Prof. Dr.-Ing. Jörg Nolte
M.Sc. Martin Schwarick

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Cottbus, den 15. Februar 2010

Matthias Brattig

Inhaltsverzeichnis

1 Einführung **1**

2 Pascal-FC **3**
- 2.1 Bedeutung . . . 3
- 2.2 Anwendung . . . 3
- 2.3 Beispiele . . . 4

3 JavaCC **7**
- 3.1 Grammatik . . . 7
- 3.2 Bestandteile des Parsergenerators . . . 9
- 3.3 Arbeitsweise des Parsergenerators . . . 10

4 Prozesse **13**
- 4.1 Überblick . . . 13
- 4.2 JavaFC-Syntax . . . 13
  - 4.2.1 Process . . . 14
  - 4.2.2 Cobegin . . . 14
  - 4.2.3 Program . . . 15
  - 4.2.4 Beispiel . . . 15
- 4.3 Überführung in Java-Syntax . . . 16
  - 4.3.1 Die Prozessklasse . . . 16
  - 4.3.2 Cobegin und Program . . . 17
  - 4.3.3 Ende-Synchronisation der Threads . . . 19
- 4.4 Zusammenfassung . . . 19

5 Semaphor **21**
- 5.1 Überblick . . . 21
- 5.2 JavaFC-Syntax . . . 21
  - 5.2.1 Semaphor . . . 21
  - 5.2.2 Beispiel . . . 22
- 5.3 Überführung in Java-Syntax . . . 22
  - 5.3.1 Ersetzen der Methodennamen . . . 23
- 5.4 Zusammenfassung . . . 24

6 Kritische Region 25
6.1 Überblick . . . 25
6.2 JavaFC-Syntax . . . 25
6.2.1 Shared . . . 25
6.2.2 Region . . . 25
6.2.3 Beispiel . . . 26
6.3 Überführung in Java-Syntax . . . 28
6.3.1 SharedObject . . . 29
6.4 Zusammenfassung . . . 30

7 Monitor 31
7.1 Überblick . . . 31
7.2 JavaFC-Syntax . . . 31
7.2.1 Monitor . . . 31
7.2.2 Condition . . . 32
7.2.3 Beispiel . . . 33
7.3 Überführung in Java-Syntax . . . 33
7.3.1 Monitorrumpf . . . 34
7.3.2 Conditions . . . 34
7.3.3 Export-Methoden . . . 35
7.3.4 MonitorObject . . . 35
7.3.5 Lock und Unlock . . . 36
7.3.6 Delay und Resume . . . 37
7.4 Zusammenfassung . . . 39

8 Ada-Style-Rendevous 41
8.1 Überblick . . . 41
8.2 JavaFC-Syntax . . . 41
8.2.1 Entry . . . 41
8.2.2 Accept . . . 42
8.2.3 Beispiel . . . 42
8.3 Überführung in Java-Syntax . . . 43
8.3.1 Benötigte Datenstrukturen . . . 43
8.3.2 Anpassung des entry- und des accept-Statements . . . 47
8.4 Zusammenfassung . . . 49

9 Kanal 51
9.1 Überblick . . . 51
9.2 JavaFC-Syntax . . . 52
9.2.1 Channel . . . 52
9.2.2 Beispiel . . . 53
9.3 Überführung in Java-Syntax . . . 55
9.3.1 Channel-Klasse . . . 56
9.3.2 Channel-Exception . . . 58

9.4 Zusammenfassung 59

10 Selektives Warten **61**
10.1 Überblick 61
10.2 JavaFC-Syntax 62
10.2.1 Select 62
10.2.2 Beispiel 63
10.2.3 Sematik des select-Statements 64
10.2.4 Anmerkungen zur replicate-Alternative 64
10.2.5 Anmerkung zur Verwendung von Kanälen 65
10.2.6 Anmerkungen zur terminate-Alternative 65
10.2.7 Anmerkung zur timeout-Alternative 65
10.3 Überführung in Java-Syntax 66
10.3.1 Anpassung des select-Statements 66
10.3.2 Benötigte Datenstrukturen 67
10.3.3 Überprüfung der Alternativen 68
10.3.4 Zusammenfassen der Daten 69
10.3.5 Auswählen einer Alternative 70
10.3.6 Die Umwandlung einer replicate-Alternative 70
10.4 Zusammenfassung 71

11 Einschätzung und zukünftige Entwicklung **75**

12 Zusammenfassung **79**

A Zusammenfassung der JavaFC-Syntax **81**
A.1 Allgemeine Statements 81
A.2 Process, Cobegin und Program 81
A.3 Semaphor 81
A.4 Shared und Region 82
A.5 Monitor und Condition 82
A.6 Entry und Accept 83
A.7 Channel 83
A.8 Select 83

Literaturverzeichnis **85**

1 Einführung

Die folgende Arbeit zur Entwicklung eines Präprozessors für JavaFC behandelt eine syntaktische Erweiterterung für Java, die die angebotenen Synchronisationskonzepte Javas vergrößert. Folgende Mechanismen unterstützt JavaFC:

- Semaphor [4],
- kritische Region [5],
- Monitor [6],
- Ada-Style-Rendevous [8],
- Kanal [7],
- selektives Warten [8]

Dabei werden alle neuen Konzepte als festes Element der Sprachsyntax integriert, dass heißt, es werden in JavaFC keine Bibliotheken importiert oder vorgefertigte Klassen benutzt, sondern neue Schlüsselworte und syntaktische Regeln in Java eingeführt.
JavaFC orientiert sich dabei an der Programmiersprache Pascal-FC [1, 2, 3]. Diese wurde an der Universität "University of Bradford, UK" für Lehrzwecke entwickelt. Pascal-FC vereint alle wichtigen programmiersprachlichen Konzepte der Nebenläufigkeit. Das ermöglicht die Lehre aller Synchronisationsmechanismen im Rahmen einer einzigen Programmiersprache. Die meisten Programmiersprachen bieten nur ein bestimmtes Konzept oder Grundmechamismen zur Synchronisation an. Im Falle von Java bildet der *synchronized*-Block, welcher auch auf Methoden anwendbar ist, die Grundlage für die Synchronisation [9]. JavaFC erweitert die Auswahl der Synchronisationsmechanismen von Java und bildet diese auf neue sprachliche Elemente ab.
Zur Umsetzung dieses Vorhabens wird der Java-Compiler-Compiler (JavaCC) eingesetzt [10]. Dieser Parser-Generator bietet die Möglichkeit die Java-Grammatik zu erweitern und neue Schlüsselworte einzuführen.
JavaFC ist die Sprache, deren Übersetzung über einen Präprozessor auf Java zurückgeführt wird und anschließend mittels Java-Compiler in Bytecode übersetzbar und auf einer Java-Virtual-Machine ausführbar ist.
Der Vorgang der Syntaxanpassung soll für den Anwender völlig transparent verlaufen. Ziel ist es, eine Quelldatei an JavaFC zu übergeben, eine Zieldatei zu generieren und diese im Anschluss zu kompilieren.

In den folgenden Kapiteln werden die Grundzüge von Pascal-FC, sowie JavaCC erläutert. Pascal-FC stellt dabei die fachliche Grundlage von JavaFC dar, auf eine genaue Beleuchtung der einzelnen Elemente Pascal-FCs wird an dieser Stelle jedoch verzichtet, da die einzelnen Kapitel des Hauptteils der Arbeit jedes Element genauer betrachten.
JavaCC bildet das Werkzeug zur Umsetzung von JavaFC und wird im Anschluss an Pascal-FC vorgestellt.
Den Hauptteil der Arbeit bildet die Vorstellung aller neuen Synchronisationsmechanismen. Jedes Kapitel stellt zunächst das Konzept vor. Dabei werden allgemeine Arbeitsweise und Bestandteile des jeweiligen Elements, sowie ein Beispiel für die Verwendung vorgestellt. Im Anschluss wird für jedes Statement, bzw. Synchronisationsmittel eine neue Syntax für JavaFC eingeführt, welche sich meist am Vorbild von Pascal-FC orientiert. Da JavaFC keinen eigenständigen Compiler darstellt, wird im Anschluss die Überführung des Statements in regulären Java-Code diskutiert.
Das gewählte Beispiel, an dem die JavaFC-Syntax vorgestellt wird, ist ein typische Anwendungsbeispiel in der Lehre nebenläufiger Systeme. Dabei handelt es sich um einen Garten, der zwei oder mehr Eintrittstore (oder Drehkreuze) besitzt. Die Tore besitzen einen gemeinsamen Zähler, der die Anzahl der Besucher im Garten speichert. Die verschiedenen Synchronisationsmittel dieser Arbeit dienen dabei als Schutz.

2 Pascal-FC

JavaFC orientiert sich an der Programmiersprache Pascal-FC. Das Ziel ist es, die von Pascal-FC angebotenen Synchronisationsmechanismen auf Java zu übertragen. Dabei soll JavaFC den lehrsprachlichen Charakter von Pascal-FC erhalten.

2.1 Bedeutung

Pascal-FC ist eine Programmiersprache, die den Umgang mit nebenläufigen Programmierkonzepten vermitteln soll. Dabei werden verschiedene Ansätze der Prozesssynchronisation umgesetzt. Das Semaphor und der Monitor inklusive *Condition*-Variablen [6] stellen dabei die klassischen Konzepte zur Synchronisation dar. Darüber hinaus werden synchrones *message passing* mit Hilfe von Kanälen[1] [7], sowie *remote invocations* und das selektive Warten[2] angeboten [8]. Diese Elemente finden vorzugsweise in nachrichten-basierten Programmiersprachen wie Occam und Ada Verwendung.

2.2 Anwendung

Der große Vorteil der Sprache ist der häufige und zufällige Wechsel zwischen verschiedenen Prozessen. Das bietet eine gute Grundlage für die Simulation echter Nebenläufigkeit, was *Deadlock*-, *Starvation*- oder *Lost-Update*-Szenarios sehr schnell zu Tage fördern kann [13].
Dabei erhält Pascal-FC die einfache Lesbarkeit von Pascal und bettet die verschiedenen Konzepte in eine Pascal-typische Syntax ein. Die verschiedenen Synchronisationsmechanismen können in Pascal-FC sowohl für sich, als auch gemeinsam bzw. verschachtelt genutzt werden. Um die Syntax von Pascal-FC zu verdeutlichen, werden im Folgenden zwei Beispiele vorgestellt.

[1] Kanäle werden in den Programmiersprachen *Occam* [7] als Synchronisationsmittel eingesetzt.
[2] Die *remote invocation* ist ein Bestandteil von Ada'95 und wird beim *Ada-Style-Rendevous* genutzt.

2.3 Beispiele

Die Syntax von Pascal-FC Programmen orientiert sich an der klassischen Pascal-Syntax und zeichnet sich durch gute Lesbarkeit und Strukturiertheit aus.

Beispiel 1 - Channels
Das erste Beispiel zeigt eine einfache Kommunikation zweier Prozesse über Kanäle.

```
1  program ProduceConsume;
2    type IntChan = channel of integer;
3    var ch1: IntChan;
4
5    process Receive; /* receive process*/
6      var item: integer;
7      begin
8        ch1 ? item; //consume
9      end;
10
11   process Send; /* send process*/
12     begin
13       ch1 ! 42; //produce
14     end;
15
16   begin
17     cobegin /* start processes*/
18       Receive; Send
19     coend
20   end.
```

Erklärung:

- Der Datentyp `IntChan` ist vom Typ Kanal und überträgt *Integer*-Daten.
- Eine Variable des definierten Typs wird angelegt
- Ein Sende- und Empfangsprozess wird angelegt.
- Das Senden wird durch `!` und das Empfangen durch `?` dargestellt.
- Gestartet werden die Prozess mittels *cobegin-coend*-Statement.

Beispiel 2 - Ada-Style-Rendevous
Das zweite Beispiel zeigt ein Lese- und Schreibszenario mittels *Ada-Style-Rendevous*. Das *Select*-Statement sorgt dafür, dass zwischen Senden und Empfangen zufällig ausgewählt werden kann.

```
1   process Share;
2     entry read(var i: integer); // read entry
3     entry write(i: integer); // write entry
4
5     var
6       value: integer; // shared variable
7
8     begin
9       accept write(i: integer) do
10        value := i; //write initial value
11
12      repeat
13        select
14          accept write(i: integer) do
15            value := i; //write value
16        or
17          accept read(var i:integer) do
18            i := value; //read value
19        end
20      forever
21    end.
```

Erklärung:

- Der Prozess `Share` bietet zwei *Entries* an (↗ siehe Kapitel 8.1); einen zum Schreiben und einen zum Lesen einer Variable.
- Die Variable wird initial beschrieben.
- In einer Endlosschleife ruft das *select*-Statement zufällig die Lese- oder Schreiboperation auf, die auf die geteilte Variable zugreift.

Pascal-FC wurde viele Jahre erfolgreich in der Lehre zum Einführen programmiersprachlicher Konzepte der Nebenläufigkeit eingesetzt. Mittlerweile ist jedoch Pascal als sequentielle Programmiersprache nicht mehr so verbreitet wie vor 10 Jahren; sie wurde in der Lehre weitestgehend von Java verdrängt. Deshalb soll eine zeitgemäße Adaption für Pascal-FC gefunden werden.
JavaFC übernimmt sämtliche Elemente von Pascal-FC, erweitert sie um die kritische Region und bringt sie in die objektorientierte Welt von Java.

3 JavaCC

Der Java-Compiler-Compiler (JavaCC) ist ein in Java implementierter Parser-Generator [12]. Um einen Parser generieren zu können, benötigt JavaCC eine Grammatik. Um den Sprachumfang von Java zu erweitern, muss die Java-Grammatik um neue Schlüsselworte sowie Regeln ergänzt werden.
Für eine erweiterte Grammatik müsste nun auch ein neuer Compiler entwickelt werden, der den Quellcode der neuen Grammatik in Bytecode übersetzt. Für den neuen Bytecode müsste ein entsprechender Bytecode-Interpreter - sprich Virtual Machine - erstellt werden. Um diesen Weg zu vermeiden, wird JavaFC in Richtung eines C++ Präprozessors entwickelt. Der C++ Präprozessor ist dem eigentlichen Compilevorgang vorgeschaltet und behandelt vordefinierte Makros wie "♯include", "♯define" oder "♯if". JavaFC durchläuft ebenfalls den Quelltext und nimmt an vordefinierten Schlüsselworten Anpassungen vor.
Die erweiterte Grammatik wird durch den JavaFC-Parser erkannt und alle neuen Schlüsselworte bzw. Ableitungsregeln werden durch bekannten Java-konformen Code ersetzt, welcher im Anschluss mittels Java-Compiler übersetzbar und auf jeder Java Virtual Machine ausführbar ist.

Abbildung 3.1: Arbeitsweise von JavaFC

3.1 Grammatik

An die neue JavaFC-Grammatik werden folgende Bedingungen geknüpft:

- sie muss vollständig abwärtskompatibel zu Java 6.0 sein,
- den klassischen Java-Sprachumfang nicht erweitern,
- oder ihn einschränken (die Mächtigkeit verändern) und
- die Erweiterungen sollen sich an der Java-typischen Grammatik orientieren.

Um das Aussehen einer JavaCC-Grammatik etwas zu verdeutlichen, folgt ein einfaches Beispiel für das *if*-Statement. Das Beispiel ist dabei in einer Backus-Naur-Form formuliert, wobei Nichtterminalsymbole durch einen Methodenaufruf und Terminale durch - in Hochkommas geschriebene - Schlüsselworte dargestellt werden. Runde und eckige Klammern zeigen die Häufigkeit einer Ableitung an [11].
Folgendes Beispiel zeigt ein *if*-Statement in JavaCC-Syntax.

```
1  IfStatement(){
2    "if" "(" Expression() ")" Statement()
3    [ "else" Statement() ]
4  }
```

Erklärung:

- Das Statement wird vom Schlüsselwort `if` eingeleitet, gefolgt von einem Klammerausdruck, in dem ein beliebiger (bool'scher) Ausdruck formuliert sein kann.
- Nach der schließenden Klammer folgt ein `Statement`, was wiederum zu jedem Element der Sprache abgeleitet werden kann.
- Im Anschluss kann ein optionales *else*-Statement verwendet werden.

Für die folgenden Kapitel wird eine Vereinfachung der Schreibweise vereinbart. Statt eines Methodenaufrufs werden Nichtterminalsymbole durch groß geschriebene und Terminalsymbole oder Schlüsselworte durch klein geschriebene und in Hochkommas gesetzte Worte ersetzt. Das vorangegangene Beispiel sieht an die veränderte Schreibweise folgendermaßen aus:

- IfStatement := if (Expression) Statement [else Statement]

Die Grammatik wird von links nach rechts gelesen. Da die folgenden Kapitel auch kompliziertere Statements behandeln, kann es passieren, dass sich die Grammatik eines einzelnen Statements über mehrere Zeilen erstreckt. In einem solchen Fall beginnt die Grammatik mit dem Nichtterminalsymbol der ersten Zeile.
Da die gesamte JavaFC-Grammatik zu umfangreich ist um vollständig abgebildet werden zu können, wird jedes neue Statement isoliert eingeführt (so wie in diesem Beispiel das `IfStatement`). Im Wesentlichen gibt es nur drei unterschiedliche Arten von neuen Statements.
Die erste Kategorie beinhaltet neue Klassentypen, die auf die gleiche Art verwendet werden können wie die Java-Schlüsselworte *class* und *interface*. In diese Rubrik wird unter anderem der Prozess oder der Monitor eingeordnet (↗ siehe Kapitel 4 und 7), welche überall dort verwendet werden können, wo die standardisierte Java-Grammatik auch eine Klasse oder ein Interface zuläßt.
In die zweite Kategorie werden sämtliche neuen Datentypen eingeordnet. Ähnlich wie der Datentyp *int* oder *char* können diese an jeder Stelle verwendet werden, an der die Java-Syntax eine Variablendeklaration zuläßt[1]. Zu dieser Rubrik wird das

[1] Das können Klassenvariablen aber auch lokale Variablen innerhalb einer Methode sein.

Semaphor oder die *entry*-Deklaration des *Ada-Style-Rendevous* gezählt (↗ siehe Kapitel 5 und 8).
Die dritte Kategorie umfasst alle neuen Statements, welche genauso verwendet werden, wie ein *if*- oder ein *switch-case*-Statement. In diese Rubrik wird unter anderem das *region*-, das *accept*- oder das *select*-Statement eingeordnet (↗ siehe Kapitel 6, 8 und 10).

Besonderheiten

Eine weitere Vereinbarung, die für die neuen Grammatiken getroffen werden, ist die Festlegung von Bezeichnern:

- Process := process NAME { (ProcessBody)* }

Dieses Beispiel ist ein Ausschnitt aus der Grammatik des neuen Prozesstyps (↗ siehe Kapitel 4). Dabei wird das in Großbuchstaben geschriebene `NAME` nicht weiter abgeleitet. Hier wird der Name des Prozesstyps festgelegt. Eine weitere Variable ist `TYPE`, die vom Nutzer das Festlegen eines Datentyps verlangt.
Eine weitere Besonderheit ist folgende:

- ProcessBody := BodyMethod | ...

Dieses Beispiel stammt ebenfalls aus der Grammatik des neuen Prozesses. Die drei Punkte auf der rechten Seite der Ableitung stehen für beliebige weitere Elemente innerhalb des Prozessrumpfes. Hier können Variablen deklariert, Methoden definiert oder eine interne Klasse angelegt werden.
Innerhalb der Grammatik eines neuen Statements können wiederum Nichtterminalsymbole zu finden sein, die nicht weiter abgeleitet werden, wie in folgendem Beispiel:

- IfStatement := if (Expression) Statement [else Statement]

Die Nichtterminalsymbole `Expression` und `Statement` müssen an dieser Stelle erneut abgeleitet werden, wobei auf die genaue Ableitung verzichtet wird.
Das `Statement` steht dabei für sämtliche Elemente der Sprache, die den Kontrollfluss des Programms beeinflussen können, wie zum Beispiel das *if*-, *switch-case*-Statement oder die *while*-, *do-while*- oder *for*-Schleifen.
Das Nichtterminalsymbol `Expression` kann dabei für verschiedenste Bestandteile der Java-Syntax stehen, zum Beispiel Zuweisungen oder bool'sche Ausdrücke. JavaFC verwendet `Expression` jedoch meist in einem Zusammenhang, in dem ein bool'scher Ausdruck verlangt wird.

3.2 Bestandteile des Parsergenerators

Mit JavaCC ist es möglich, jedes Token der Grammatik zu markieren. Diese Markierung dient als zusätzliche Information für die Weiterverarbeitung, hat jedoch keinen Einfluss auf die Grammatik. Alle neuen Elemente der Sprache müssen markiert

werden, damit diese später in eine konforme Java-Syntax überführt werden können. Unmarkierte Token entsprechen bereits der Java Syntax und können unbearbeitet in die Zieldatei übernommen werden.
Der erste Schritt ist die Markierung aller neuen Elemente der Grammatik. Damit jedes markierte Token bei der Auswertung der Syntax identifiziert werden kann, muss jede Markierung mit einem individuellen Namen versehen werden. JavaCC schreibt außerdem vor, dass es für jeden dieser Namen eine Java-Klasse geben muss. In diesen Klassen wird die Verarbeitung jedes Tokens festgehalten.
Die Weiterverarbeitung jedes Tokens ist der zweite Schritt bei der Erstellung des JavaFC-Präprozessors. Jede Klammer oder jedes Schlüsselwort, das neu in die Grammatik eingefügt wurde, kann hier verändert werden. Auf diese Weise ist es möglich, Schlüsselworte zu löschen, zu ersetzen oder neue Codeteile einzufügen. Nachdem die markierten Token nachbearbeitet wurden, werden diese in eine Zieldatei geschrieben und sollten nun den syntaktischen Vorgaben von Java entsprechen. Eine neue Regel und ihre Nachbearbeitung könnte folgendermaßen aussehen:

```
1  WhenStatement(){
2    "when" (Mark1) "(" Expression() ")" Statement
3  }
```

Das `when`-Statement soll durch ein Java-konformes *if*-Statement ersetzt werden. Wenn der Parser beim Durchlaufen des Quelltextes auf ein `when` stößt, wird dieses markiert und nachbearbeitet.
Parser-intern ist das `when`-Token nichts weiter als ein Blatt in einem Syntaxbaum, was die Zeichenkette `when` enthält. Nachdem das "when" durch ein "if" ausgetauscht wurde und das "if" in die Zieldatei geschrieben wurde, entspricht der umgewandelte Quelltext der Java-Syntax.

3.3 Arbeitsweise des Parsergenerators

Der Java-Compiler-Compiler erstellt aus der Grammatik ein Java-Programm, das die in der Grammatik formulierte Syntax einer Sprache erkennt. Das Endergebnis des Entwicklungsprozesses ist ein automatisch generiertes Java-Programm. Dieses Programm überprüft die Quelldatei auf syntaktische Korrektheit.
Nachdem die Quelldatei eingelesen wurde, erstellt der Parser einen Syntaxbaum. Dabei entspricht jede Ableitungsregel einem Knoten und jedes Schlüsselwort einem Blatt im Baum. Erreicht der Parser ein spezielles Token, wird es für die Nachbearbeitung markiert.
Nachdem der Quelltext eingelesen ist, wobei er den syntaktischen Vorgaben der Grammatik entsprechen muss und jedes Token in einer Baumstruktur abgelegt wurde, folgt die Traversierung des Baums. Unmarkierte Token werden unverändert in die Zieldatei geschrieben. Bevor auch die markierte Token in die Zieldatei geschrieben werden, erfahren diese meist eine Anpassung, um der Standard-Java-Syntax zu entsprechen.

Nachdem auch alle markierten Token bearbeitet und ggf. in die Zieldatei geschrieben wurden, kann diese mit Hilfe des Java-Compilers in Bytecode übersetzt werden und auf jeder beliebigen Java-Virtual-Machine ausgeführt werden.

4 Prozesse

Prozesse und Threads spielen eine wichtige Rolle für das Erstellen nebenläufiger Systeme[1]. Das Java *Runnable*-Interface und die *Thread*-Klasse bilden eine komplette Umgebung für das Bearbeiten beliebig vieler nebenläufiger Kontrollflüsse im selben Programm.

4.1 Überblick

Pascal-FC verwendet für nebenläufige Kontrollflüsse das Schlüsselwort *process*. Dieses Schlüsselwort wird auch in JavaFC eingeführt und nimmt den gleichen Stellenwert ein, den bereits die beiden Schlüsselworte *Runnable* und *Thread* haben. Im Folgenden werden die Begriffe "Prozess", sowie "Thread", als Synonyme füreinander verwendet.
Mit Hilfe von *process* lassen sich beliebige Prozess-Typen definieren, die über das *cobegin*-Statement gestartet werden [2]. Dieses Statement findet ebenfalls Verwendung in Pascal-FC und soll das Starten von Prozessen in ein gemeinsames Statement zusammenfassen. Außerdem stellt Pascal-FC einige Bedingungen an das *cobegin*-Statement, z.B. dass alle Prozesse, die innerhalb des Statements angelegt wurden, gleichzeitig gestartet werden.
In JavaFC sollen Prozesse nur zu Begin des Programms erzeugt und gestartet werden. Desweiteren ist es ratsam, auch die verwendeten Datenstrukturen zu Beginn des Programms zu erstellen. Aus diesem Grund wird neben dem Prozess und dem *cobegin*-Statement ein weiteres Element eingeführt. Das neue *program*-Konstrukt soll vor allem die Lesbarkeit des Programms erhöhen, indem es das Anlegen von Datenstrukturen und Prozessen sowie deren Starten gesondert kapselt.

4.2 JavaFC-Syntax

Im Folgenden werden die Grammatiken des Prozesstyps, sowie das *cobegin*- und *program*-Statement eingeführt.

[1] Im Folgenden werden Thread und Prozess an Synonyme füreinander verwendet, obwohl sie auf Betriebssystemebene etwas anderes bedeuten.

4.2.1 Process

Neben den bekannten Schlüsselworten, wie *class* und *interface*, führt JavaFC das Schlüsselwort *process* für Klassen ein.

- Process := process NAME { (ProcessBody)* }
- ProcessBody := BodyMethod | ...
- BodyMethod := public void body ( ) { (Statement)* }

Erklärung:

- Prozesstypen werden wie gewöhnliche Java-Klassen angelegt, nur dass das Schlüsselwort *process* das Java-eigene Schlüsselwort *class* ersetzt.
- Der Prozesstypenrumpf kann völlig beliebig aussehen, schreibt jedoch eine `body`-Methode vor, die den nebenläufig auszuführenden Code enthält, welcher ausgeführt wird, wenn der Prozesstyp über das *cobegin*-Statement gestartet wird.

4.2.2 Cobegin

Das zweite Instrument zum Starten nebenläufiger Prozesse ist das *cobegin*-Statement. Diesen Statement besitzt in Pascal-FC ein besonderes Verhalten, was im Unterkapitel Ende-Synchronisation der Threads (↗ Kapitel 4.3.3) erklärt wird.

- Cobegin := cobegin { (CobeginEntry | For)* }
- CobeginEntry := [NAME =] new NAME ([Expression]) ;
- For := for (ForBody) (CobeginEntry)*
- ForBody := ForInit ; ForCompare ; ForUpdate

Erklärung:

- Das Statement wird mit dem Schlüsselwort `cobegin` eingeleitet.
- Innerhalb des Statements können nur Objekte von Prozesstypen angelegt und gestartet werden. Andere Objekte oder Datenstrukturen können auf Grund der internen Funktionsweise nicht erstellt werden.
- Um eine größere Anzahl Threads starten zu können, steht innerhalb von *cobegin* eine *for*-Schleife zur Verfügung. Andere Statements sind nicht zugelassen.

4.2.3 Program

JavaFC soll den Umgang mit nebenläufigen Aktivitäten vermitteln. Zu diesem Zweck werden Prozesse, sowie gemeinsam verwendete Datenstrukturen, bzw. Objekte benötigt. Das Schlüsselwort *program* hat die Aufgabe, die komplette Initialisierung des nebenläufigen Programms zu kapseln.

- Program := program NAME { (ProgramBody)* }
- ProgramBody := VariableDeclaration | CobeginStatement | ...

Erklärung:

- Das Schlüsselwort *program* leitet die Klasse ein.
- Im Körper des Statements können Datenstrukturen angelegt werden. Außerdem ist es vorgesehen, dass das *cobegin*-Statement hier verwendet wird.

4.2.4 Beispiel

Folgender Code zeigt die Verwendung des Prozesses am Beispiel des in der Einleitung erwähnten Gartenbeispiels. Allerdings kommen hier keine Synchronisationsmechanismen zum Einsatz, was bei der Ausführung zu einem falschen Zählerwert führen wird:

```
1  process Turnstile{
2    Counter counter;
3
4    Turnstile(Counter c){
5      counter = c;
6    }
7    public void body(){
8      for (int i=0; i<20; i++)
9        counter.increment();
10   }
11 }//end turnstile
```

Der Zähler wird durch die Klasse `Counter` dargestellt:

```
1  class Counter{
2    private int value = 0;
3
4    public void increment(){
5      value++;
6    }
7
```

```
8     public int getValue(){
9       return value;
10    }
11  }//end counter
```

Das *program*-Statement dient als Ausgangspunkt für das Programm, wobei *cobegin* die Prozesse startet:

```
1   program TurnstileMain{
2     Counter c = new Counter();
3
4     cobegin{
5       for(int i=0; i<2; i++)
6         new Turnstile(c);
7     }
8     System.out.println("counter: "+c.getValue());
9   }
```

Jedes Java-Programm läßt beim Starten ein String-Array mit Programm-Parametern zu. Um auf diese Parameter zugreifen zu können, wird das Array mit dem Namen `args` auch an das *program*-Konstrukt weitergegeben.

4.3 Überführung in Java-Syntax

Die Aufgabe des Präprozessors ist die Überführung der JavaFC-Konstrukte in regulären Java-Code.

4.3.1 Die Prozessklasse

Ein Prozess oder Thread stellt in Java eine Klasse dar, die das Interface *Runnable* implementiert oder die Klasse *Thread* erweitert. Zum Starten des nebenläufigen Kontrollflusses wird ein neues Thread-Objekt erstellt und die *start*-Methode auf diesem Objekt aufgerufen. Der nebenläufig auszuführende Code wird in der durch das Interface *Runnable* vorgegebenen *run*-Methode formuliert [9].
Bei der Implementierung muss zunächst die Entscheidung darüber getroffen werden, ob die Prozesse in JavaFC das *Runnable*-Interface implementieren oder die *Thread*-Klasse erweitern. Da Mehrfachvererbung in Java nicht direkt möglich ist, aber beliebig viele Interfaces implementiert werden können, nutzt JavaFC das *Runnable*-Interface für Threads, um die Möglichkeit der Vererbung nicht einzuschränken.
Das Schlüsselwort *process* leitet Klassen ein, die einen nebenläufigen Kontrollfluss darstellen. Die Anpassung der folgender Codezeilen:

```
1   process Turnstile {
```

nimmt der Präprozessor folgendermaßen vor:

```
class Turnstile implements Runnable, ProcessI {
  private Semaphore endSema;

  public Turnstile setSema(Semaphore s){
    endSema = s;
    return this;
  }

  public void run(){
    body();
    endSema.release();
  }
```

Erklärung:

- Das Schlüsselwort `process` wird durch `class` ersetzt.
- Vor die öffnende Klammer der Klasse wird die Zeichenkette `implements Runnable` eingefügt[2]. Außerdem implementiert jeder Prozess das Interface `ProcessI`. Auf diese Weise ist sichergestellt, dass der Anwender die `body`-Methode implementiert.
- JavaFC erzeugt für jeden Prozess die `setSemaphore`-Methode, welche die Referenz eines Semaphors speichert, deren Bedeutung im Unterkapitel "Ende-Synchronisation der Threads" (↗ Kapitel 4.3.3).
- Außerdem wird die `run`-Methode vom Präprozessor generiert. In ihr wird die `body`-Methode aufgerufen und nachdem diese beendet ist, eine Ressource auf dem Semaphor freigegeben.
- Die `body`-Methode des Beispiels wird unverändert übernommen.

4.3.2 Cobegin und Program

Das *program*-Statement bietet einen einfachen Weg Prozesse, Datenstrukturen, sowie Variablen anzulegen. Ein Grund für das Einführen dieses Konstrukts ist Javas statische *main*-Methode.
Das *cobegin*-Statement benötigt einige Datenstrukturen. Je nachdem wo das *cobegin*-Statement verwendet wird, müssen diese Strukturen statisch oder dynamisch sein. Da jedem Prozess, der im *cobegin*-Statement gestartet wird, ein Semaphor mitgegeben und gespeichert wird (↗ Kapitel 4.3.3), muss auch die Referenz des Semaphors statisch oder dynamisch sein, je nach Verwendung des *cobegin*-Statements.

[2] Sollte der Prozess ein anderes Interface implementieren oder eine andere Klasse erweitern, wird das an dieser Stelle berücksichtigt.

Um einer zu komplizierten Fallunterscheidung vorzubeugen, wird das *program*-Konstrukt eingeführt. Folgender Quelltext zeigt die Verwendung des *program*- und des *cobegin*-Statements:

```
program TurnstileMain{
  Counter c = new Counter();

  cobegin{
    for(int i=0; i<2; i++)
      new Turnstile(c);
  }
}
```

Dieser Code erfährt folgende Anpassung:

```
class TurnstileMain{
  public static void main(String[] args) {
    new TurnstileMain(args);
  }//end main

  public TurnstileMain(String[] args){
    Semaphore s = new Semaphore(0); //barrier semaphore
    List list = new LinkedList(); //stores Threads

    Counter c = new Counter();
    for (int i=0; i<2; i++)
      list.add(new Thread(new Turnstile(c).setSema(s)));

    for (int i=0; i<list.size(); i++)
      list.get(i).start(); //start Threads
    s.acquire(list.size()); //barrier
  }
}
```

Erklärung:

- Das Schlüsselwort `program` wird durch `class` ersetzt. Außerdem wird eine *main*-Methode in die Klasse eingefügt. In ihr wird der Konstruktor der eigenen Klasse aufgerufen.
- Der komplette Quelltext innerhalb der *program*-Klasse wird in den Konstruktor der Klasse eingefügt.
- Innerhalb des Konstruktors werden zwei Variablen angelegt, ein Semaphor für die Endsynchronisation der Threads, eine `LinkedList`, die alle angelegten Threads speichert.

- Für das *cobegin*-Statement wird das Schlüsselwort *cobegin* entfernt und jeder Thread in der *LinkedList* gespeichert. Die *for*-Schleife, die im *cobegin*-Statement verwendet wird, wird entsprechend abgewandelt. Eine *for*-Schleife startet im Anschluss alle Threads.

- Auf dem Semaphor `sema` allokiert der Hauptkontrollfluss im Anschluss genau soviele Ressourcen, wie Threads gestartet wurden.

4.3.3 Ende-Synchronisation der Threads

Das *cobegin*-Statement von Pascal-FC ist wie eine Barriere aufgebaut. Das bedeutet, dass alle Prozesse, die im *cobegin* gestartet wurden, beendet sein müssen, damit das dem *cobegin* folgende Statement ausgeführt werden kann. Ein nicht terminierender Prozess führt demnach dazu, dass das *cobegin*-Statement nie verlassen wird.
Um in JavaFC ein identisches Verhalten zu erzielen, wird jedem Thread ein Semaphor übergeben, auf dem eine Ressource freigegeben wird, nachdem die *body*-Methode ausgeführt wurde. Das *cobegin*-Statement fordert auf dem Semaphor so viele Ressourcen an, wie Threads gestartet wurden. Erst nachdem jeder gestartete Thread beendet ist kann das Statement verlassen werden.
Bei diesem Semaphor handelt es sich nicht um das im nächsten Kapitel eingeführte Semaphor in JavaFC, sondern um eine interne Variable, die der Präprozessor erzeugt, um das Verhalten des *cobegin*-Statements von Pascal-FC nachzubilden. Hierbei wird auf die *concurrent*-Bibiothek[3] von Java zurück gegriffen, die das Konzept des Semaphors bereits in Java implementiert.

4.4 Zusammenfassung

In diesem Kapitel wurden die grundlegenden Mechanismen zum Erstellen von nebenläufigen Programmen vorgestellt. Dabei stützt sich JavaFC auf bereits vorhandene Java-Bibliotheken. Das Starten neuer Prozesse wird dabei vom neuen *cobegin*-Statement übernommen. Für die Deklaration von Prozessen wurde das neue Schlüsselwort *process* eingeführt.
Da JavaFC kompatibel zu Java bleiben soll, kann weiterhin auch das Interface *Runnable* implementiert oder *Thread* erweitert werden. In diesem Fall muss jedoch die Synchronisation auf das Ende anderer Threads und das zeitgleiche Starten selbstständig nachgebildet werden.

[3] Die *concurrent*-Bibliothek ist seit der Java-Version 1.5 fester Bestandteil der Sprache und stellt u.a. eine fertige Semaphor-Implementierung zur Verfügung.

5 Semaphor

Ein Semaphor ist ein Konstrukt zum Synchronisieren von nebenläufigen Prozessen. Dabei handelt es sich um ein feingranulares Synchronisationsmittel, bei dem der Programmierer selbstständig die kritischen Stellen des Programms identifizieren und schützen muss.

5.1 Überblick

Ein Semaphor besteht aus vier wesentlichen Teilen [4]. Einer Initialisierungsmethode, einer Ein- und einer Austrittsmethode und einer Warteschlange. Pascal-FC stellt dafür die Methoden *initial*, *wait* und *signal* bereit, außerdem ist das Schlüsselwort *semaphore* Teil des Sprachumfangs.
Seit der Version 5.0 stellt Java mit der *"concurrent"*-Bibliothek eine vollständige Semaphor-Implementierung bereit. Die Initialisierungsmethode wird dabei durch den Konstruktor ersetzt und die Ein- und Austrittsmethoden heißen *acquire* und *release*. In der ursprünglichen Definition des Semaphors durch Dijkstra, werden die beiden Funktionen *p* und *v* genannt. Da diese Methodennamen zu abstrakt sind, werden die englischen Äquivalente *up* und *down* für JavaFC verwendet.
Eine Verwendung der Pascal-FC Methodennamen ist nicht möglich, da jedes Objekt in Java bereits eine *wait*-Methode implementiert. Diese Methode ist als *final* ausgezeichnet, was ein Überschreiben unmöglich macht.

5.2 JavaFC-Syntax

In JavaFC kann das Semaphor wie ein primitiver Datentyp verwendet werden, mit der Ausnahme, dass auf dem Semaphor - anders als in Java üblich - Methodenaufrufe möglich sind.

5.2.1 Semaphor

Die Backus-Naur-Form der Semaphor-Deklaration unter JavaFC sieht wie folgt aus:

- Semaphore := semaphore SemaphoreDecl (, SemaphoreDecl)* ;
- SemaphoreDecl := NAME [(Expression)]

Erklärung:

- Die Semaphordeklaration wird mit dem Schlüsselwort *semaphore* eingeleitet, gefolgt von einem Bezeichner.
- Danach folgt die optionale Initialisierung des Semaphors.
- Wenn auf eine Initialisierung verzichtet wird, kann das Semaphor nachträglich über die `init`-Methode initialisiert werden.

5.2.2 Beispiel

Das folgende Beispiel wandelt die Zähler-Klasse des Gartenbeispiels um, die `Turnstile`-, sowie die `TurnstileMain`-Klassen bleiben dabei unverändert (↗ siehe Kapitel 4.2.4):

```
class Counter{
  private semaphore mutex(1);
  private int value = 0;

  public void increment(){
    mutex.down(); //lock
    value++;
    mutex.up(); //unlock
  }//end increment

  public int getValue(){
    try{
      mutex.down(); //lock
      return counter;
    }finally{
      mutex.up(); //unlock mutex after return
    }
  }//end getValue
}
```

Die `increment`- und die `getValue`-Methode werden um einen Semaphor-Schutz erweitert, wobei die `getValue`-Methode den Rückgabewert über ein *try-finally*-Block schützt. Der Semaphorschutz wird auf diese Weise erst nach der erfolgreichen Rückgabe frei gegeben.

5.3 Überführung in Java-Syntax

Bei der Implementierung des Semaphors stützt sich JavaFC auf die Implementierung der *concurrent*-Bibliothek. Die einfachste Art der Umsetzung des JavaFC-Semaphors wäre die direkte Abbildung auf diese Bibliothek. Diese bietet mehrere

Methoden an, um sich eine Ressource des Semaphors zu sichern. Die einfachste Methode *acquire* hat jedoch den Nachteil, dass sie eine Exception werfen kann, was dazu führen würde, dass der Nutzer bei jedem Aufruf einen *try-catch*-Block setzen müsste.
Eine weitere Methode, die keine Exception wirft, heißt *acquireUninterruptibly*. Dieser Methodenname ist jedoch zu kompliziert, weshalb die beiden Methoden *up* und *down* von JavaFC verwendet werden und die Namen intern durch die Methodennamen der Bibliothek ersetzt werden.
Aus diesem Grund fügt der Präprozessor in jede Klasse, die ein Semaphor verwendet, ein Import der *concurrent*-Bibliothek ein. Folgendes Beispiel:

```
1 class Counter{
2   private semaphore mutex(1);
```

wird nach dem Parsen:

```
1 import java.util.concurrent.Semaphore;
2
3 class Counter {
4   private Semaphore mutex = new Semaphore(1);
```

Erklärung:

- Zunächst wird vor die Klasse das benötigte *import* gesetzt.
- Das Schlüsselwort *semaphore* wird durch *Semaphore* ersetzt, der Semaphorbezeichner eins zu eins übernommen.
- Vor das Klammerpaar wird `= new Semaphore` eingefügt und der Ausdruck innerhalb der Klammer unverändert übernommen.

5.3.1 Ersetzen der Methodennamen

JavaFC verwendet für die Ein- und Austrittsmethoden *up* und *down*, die Java-Bibliothek jedoch nutzt die Methodennamen *acquire* und *signal*. Beim Parsen des Quelltextes werden die unterschiedlichen Methodennamen ausgetauscht.
Beim Parsen des JavaFC Quelltextes wird jeder Semaphorbezeichner gespeichert. Wenn nun auf einem dieser Semaphoren die Ein- oder Austrittsmethode aufgerufen wird, ersetzt JavaFC sie durch den entsprechenden *concurrent*-Bibliotheksaufruf. Das folgende Beispiel:

```
1 public void increment(){
2   mutex.down(); //lock
3   value++;
4   mutex.up(); //unlock
5 }//end increment
```

wird folgendermaßen transformiert:

```
1   public void increment(){
2     mutex.acquire(); //lock
3     value++;
4     mutex.release(); //unlock
5   }//end increment
```

Der Präprozessor erkennt in diesem Beispiel das Schlüsselwort *semaphore* und speichert alle folgenden Semaphornamen. Trifft JavaFC an einer späteren Stelle des Quelltextes auf den Bezeichner `mutex`, auf dem die Methode `up` aufgerufen wird, ersetzt er diesen Aufruf durch `signal`.

5.4 Zusammenfassung

Das Semaphor bildet das erste Synchronisationsmittel von JavaFC. Es stützt sich dabei vollständig auf eine existierende Java-Bibliothek.
Neben der Einführung eines neuen Schlüsselworts besteht die Hauptaufgabe des Parsers beim Umgang mit dem Semaphorkonstrukt darin, die Ein- und Austrittsmethoden der JavaFC-Syntax an die Java-Bibliothek anzupassen.

6 Kritische Region

Die (bedingte) kritische Region ist eine Übergangsform zwischen Semaphor und Monitor (↗ Kapitel 7), bei der der Compiler für den Schutz einer Variablen sorgt [5].

6.1 Überblick

Das Konstrukt der kritischen Region besteht aus zwei Teilen. Zunächst muss eine Variable als *shared* deklariert werden. Von nun an ist es die Aufgabe des Compilers - in diesem Fall des Präprozessors - dafür zu sorgen, dass bei jedem Zugriff auf die Variable gegenseitiger Ausschluss garantiert ist.
Diese Garantie erfüllt der Präprozessor indem er darauf achtet, dass jeder Zugriff auf eine solche Variable innerhalb eines entsprechenden *region*-Statement geschieht.

6.2 JavaFC-Syntax

Die kritische Region besteht aus zwei Teilen. Zum einen die Vereinbarung einer *shared* Variablen und zum anderen aus dem eigentlichen *region*-Statement.

6.2.1 Shared

Die Grammatik einer *shared*-Variablendeklaration sieht wie folgt aus:

- SharedDeclaration := shared TYPE NAME (, TYPE NAME)* ;

Erklärung:

- Die zu schützende Variable wird mit dem Schlüsselwort *shared* plus Datentyp, gefolgt vom Bezeichner angelegt.

6.2.2 Region

Nach der Deklaration des *shared*-Statements folgt das dazugehörige *region*-Statement sowie das bedingte *region*-Statement.

- CriticalRegionStatement := region NAME { (Statement)* }
- ConditionalRegionStatement := region NAME Guard { (Statement)* }
- Guard := when (Expression)

Erklärung:

- Eine kritische Region ist immer an eine *shared*-Variable gebunden. Das Schlüsselwort *region* eröffnet das Statement, gefolgt vom Namen der *shared*-Variablen, auf die zugegriffen wird.
- Innerhalb einer kritsichen Region können beliebige weitere *region*-Statements formuliert werden. Allerdings führt die Verschachtelung von mehreren *region*-Statements hinsichtlich der gleichen *shared*-Variable zu einem Deadlock und wird durch einen Fehler vom Präprozessor gemeldet.
- Wenn der Zutritt in eine kritische Region an eine Bedingung geknüpft werden soll, kann dieses mittels *when*, gefolgt von einem bool'schen Ausdruck angezeigt werden. Der Zugriff auf die *shared*-Variable ist neben dem *region*-Statement auch im bool'schen Ausdruck des *Guards* zulässig.

6.2.3 Beispiel

Um das Gartenbeispiel mit Hilfe einer kritschen Region umzusetzen, bieten sich zwei verschiedene Möglichkeiten an. Zum einen könnte man den Zähler in der `Counter`-Klasse als *shared* deklarieren, zum anderen könnte die Instanz der `Counter`-Klasse, welche in der `TurnstileMain`-Klasse angelegt wird, als *shared* deklariert werden. JavaFC unterstützt beide Varianten.
Die erste Variante zeigt die angepasste `Counter`-Klasse, die Klassen `Turnstile` und `TurnstileMain` bleiben dabei unverändert (↗ siehe Kapitel 4.2.4):

```
1  class Counter{
2    private shared int value;
3
4    Counter(){
5      region value{
6        value = 0;
7      }
8    }
9
10   public void increment(){
11     region value{
12       value++;
```

```
13        }
14      }//end increment
15
16      public int getValue(){
17        region value{
18          return value;
19        }
20      }//end getValue
21    }//end Counter
```

Die zweite Variante ist das Anlegen einer *shared*-Instanz der `Counter`-Klasse. Dabei kann die ursprüngliche Implementierungsvariante der `Counter`-Klasse vom Kapitel 4.2.4 übernommen werden.

```
1   program TurnstileMain{
2     shared Counter c = new Counter();
3
4     cobegin{
5       for(int i=0; i<2; i++)
6         new Turnstile(c);
7     }
8     region c{
9       System.out.println("counter: "+c.getValue());
10    }
11  }//end TurnstileMain
```

Die `Turnstile`-Klasse verlangen folgende Änderungen:

```
1   process Turnstile{
2     shared Counter counter;
3
4     Turnstile(shared Counter c){
5       counter = c;
6     }
7     public void body(){
8       for (int i=0; i<20; i++){
9         region counter{
10          counter.increment();
11        }
12      }
13    }
14  }//end turnstile
```

6.3 Überführung in Java-Syntax

Bei der Anpassung des *shared*- und *region*-Statements verfährt der Präprozessor immer gleich, egal ob eine Instanz eines komplexen Datentyps oder nur eine Membervariable als *shared* deklariert wurde (siehe Beispiele). Die Anpassung wird an einer *shared*-Instanz eines komplexen Datentyps demonstriert.
Sämtliche zu schützenden Objekte werden in eine *Container*-Klasse verpackt, die den Schutz des Objekts bereitstellen. Folgenden Ausdruck:

```
1 shared Counter c = new Counter();
```

wird folgendermaßen angepasst:

```
1 SharedObject<Counter> c = new SharedObject<Counter>(new
      Counter());
```

Erklärung:

- Das Schlüsselwort *shared* wird durch die Zeichenkette `SharedObject<` ersetzt. Der geteilte Datentyp wird übernommen, allerdings wird zusätzlich eine schließende Klammer `>` eingefügt.
- Zwischen das `=` und das `new` des ursprünglichen Statements fügt der Präprozessor `new SharedObject<Counter>(` ein. Der Rest des Statements wird übernommen.
- *SharedObject* stellt eine Container-Klasse dar, die die geteilten Objekte schützen, ähnlich einem Monitor. Der eigentliche `Counter` ist nur ein Bestandteil des Containers und die Variable `c` bezeichnet nicht mehr den *Counter*, sondern eine Instanz eines *SharedObjects*.

Den zweiten Bestandteil der kritischen Region bildet das *region*-Statement, welches in einer geschützten und einer ungeschützten Variante verwendbar ist. Bei der Verwendung eines Schutzes, wird das *region*-Statement erst betreten, wenn der bool'sche Ausdruck erfüllt ist.

```
1 region c{
2   System.out.println("counter: "+c.getValue());
3 }
```

Aus dem Code generiert der JavaFC-Präprozessor folgenden Quelltext:

```
1 try{
2   c.enterRegion();
3   c.content.increment();
4 }finally{
5   c.enterRegion();
6 }
```

Erklärung:

- JavaFC ersetzt das Schlüsselwort und die Klammern des *region*-Statements durch einen `try-finally`-Block. Außerdem werden eine Ein- und Austrittsmethode für den Schutz der *shared* Variablen aufgerufen[1].
- Da der eigentliche *Counter* nur eine Membervariable des *SharedObjects* mit dem Namen *content* ist, muss JavaFC die Zeichenkette `.content` einfügen.
- Beim Einsatz eines *Guards* wird der bool'sche Ausdruck zyklisch überprüft und wenn dieser erfüllt ist, die kritische Region betreten.

6.3.1 SharedObject

Die Klasse *SharedObject* ist eine Container-Klasse für jede *shared*-Variablen, deren Aufgabe der Schutz der Variable ist. Sie implementiert Ein- und Austrittsfunktionen, die auf die Variable des zugreifenden Threads synchronisiert. Die Klasse ist wie folgt aufgebaut:

```
1  class SharedObject<T>{
2    private Lock lock = new ReentrantLock();
3    public T content;
4
5    public SharedObject(T obj){
6      content = obj;
7    }
8    public void enterRegion(){
9      lock.lock();
10   }
11   public void leaveRegion(){
12     lock.unlock();
13   }
14 }//end SharedObject
```

Erklärung:

- Bei *SharedObject* handelt es sich um eine generische Klasse, die jeden komplexen Datentyp verarbeiten kann. Ein *lock*-Objekt der Bibliothek *concurrent* sorgt für den Schutz des geteilten Objekts.
- Neben dem Konstruktor, dem das zu schützende Objekt übergeben und als `content`-Variable gespeichert wird, bietet die *SharedObject*-Klasse eine Ein- und Austrittsmethode an, die immer beim Betreten und Verlassen der kritschen Region aufgerufen werden.

[1] Das *try-finally* wird verwendet, da innerhalb des Statements ein *return* verwendet werden kann und der Schutz des Objekts wieder aufgehoben werden muss.

Einschränkung des SharedObjects

Der Aufbau des *SharedObjects* und die Arbeitsweise von Java ziehen eine wichtige Einschränkung für *shared*-Variablen nach sich.
Der *SharedObject*-Klasse wird bei der Instanzierung ein beliebiger generischer Datentyp übergeben. Dieser Datentyp wird anschließend als die `content`-Variable gespeichert und stellt seinerseits ebenfalls einen generischen Datentyp dar.
Java's *Generics* erfahren zur Compilezeit eine Typenlöschung und werden intern auf die *Object*-Klasse abgebildet. Das bedeutet, dass es zur Laufzeit keinerlei Typinfomationen mehr gibt und auch der *instance of*-Operator nicht zur Verfügung steht [9]. Die Abbildung sämtlicher Informationen auf den allgemeinen *Object*-Datentyp führt dazu, dass es nicht möglich ist, primitive Datentypen - wie *int* oder *boolean* - im Zusammenhang mit *shared*-Variablen zu verwenden.
Allerdings bietet Java für jeden primitiven Datentyp eine Wrapper-Klasse an, die sich in der Regel genauso verhält, wie die primitiven Typen. Als Beispiel sei der `++`-Operator für den *int*-Typ genannt, welcher genauso für die *Wrapper*-Klasse *Integer* zur Verfügung steht.

6.4 Zusammenfassung

Das Konstrukt der kritische Region besteht aus zwei wesentlichen Teilen, einer Variablenvereinbarung und einem Statement. Jede Variable, die mit dem Schlüsselwort *shared* gekennzeichnet ist, darf nur innerhalb eines *region*-Statements gelesen oder verändert werden. Da der Compiler die Überwachung der Variablen übernimmt, kann es nicht passieren, dass der Schutz einer *shared*-Variablen vergessen wird.

7 Monitor

Ein Monitor ist ein grobgranulares Konstrukt zum Synchronisieren von kritischen Codebereichen. Es ist möglich, mit Hilfe eines Monitor für komplette Klassen, Methoden oder Datenstrukturen gegenseitigen Ausschluss zu garantieren [6].

7.1 Überblick

Das Konzept des Monitors in Pascal-FC umfasst drei wesentliche Bestandteile. Jede öffentliche Methode des Monitors wird in Pascal-FC als *Export*-Methode bezeichnet. Diese Methoden garantieren gegenseitigen Ausschluss. Innerhalb einer *Export*-Methode können weitere Untermethoden aufgerufen werden, wobei der Schutz des Monitors auch rekursive Aufrufe von *Export*-Methoden zulassen soll (was zu keiner Verklemmung führen darf).
Sollte ein Prozess in einen besetzten Monitor eintreten wollen, wird dieser aufgehalten und auf einer Warteschlange abgelegt. Neben der normalen Warteschlange kennt der Monitor Pascal-FCs eine weitere Warteschlange. Auf der zweiten Warteschlange werden Prozesse vermerkt, die die Kontrolle des Monitors freiwillig abgegeben haben. Diesen Prozessen wird beim Eintritt in den Monitor Vorrang gewährt, wenn der Schutz des Monitors wieder verfügbar ist. Diese Liste wird im Folgenden als *UrgentList* bezeichnet.
Das dritte Element des Monitors ist die *Condition*. Dabei handelt es sich um ein Element, das es Prozessen erlaubt, sich innerhalb des Monitors schlafen zu legen und auf das Eintreffen eines Ereignisses zu warten. Der entsprechende Prozess gibt in dem Fall die Kontrolle über den Monitor ab, bekommt diese jedoch wieder, wenn die erwartete Bedingung eintritt.

7.2 JavaFC-Syntax

Neben dem Schlüsselwort *class*, *interface* und *process* wird an dieser Stelle das Schlüsselwort *monitor* eingeführt, welches eine Monitortyp-Klasse einleitet.

7.2.1 Monitor

Die folgende Grammatik zeigt die Monitor-Deklaration in BNF. Das Anlegen von *Export*-Methoden oder von *Conditions* ist nur im Zusammenhang mit einem Mo-

nitor möglich, d.h. die Schlüsselworte *condition* und *export* sind außerhalb eines Monitors nicht zulässig.

- Monitor := monitor NAME { (MonitorBody)* }
- MonitorBody := ExportMethod | CoditionDeclaration | ...
- ExportMethod := export ExportSignature { (Statement)* }
- ExportSignature := TYPE NAME ((Parameter)*)

Erklärung:

- Der Monitor wird mit dem neuen Schlüsselwort *monitor* eingeleitet. Danach folgt der Bezeichner des Monitortyps.
- Eine *Export*-Methode wird mit dem Schlüsselwort *export* eingeleitet und ersetzt das Java-typische Schlüsselwort *public*.
- Um eine Bedingungsvariable anzulegen, wird das Schlüsselwort *condition* verwendet. Allerdings ist neben dem Bezeichner keine weitere Initialisierung möglich. Auf einer *Condition* sind lediglich die beiden Methodenaufrufe - *delay* und *resume* - zulässig.

7.2.2 Condition

Die Grammatik der *Condition*-Deklaration sieht wie folgt aus:

- ConditionDeclaration := condition NAME (, NAME)* ;
- ConditionStatement := NAME . delay ( ) | NAME . resume ( )
  | NAME . size ( )

Erklärung:

- Um eine Bedingungsvariable anzulegen, wird das Schlüsselwort *condition* verwendet. Allerdings ist neben dem Bezeichner keine weitere Initialisierung möglich.
- Auf einer *Condition* sind die Methodenaufrufe - *delay*, *resume* und *size* - zulässig.

7.2.3 Beispiel

Bei der Anpassung des Gartenbeispiels an den Monitor wird die `Counter`-Klasse in einen Monitor umgewandelt, die beiden Klassen `Turnstile` und `TurnstileMain` bleiben unverändert:

```
monitor Counter{
  private int value = 0;

  export void increment(){
    value++;
  }//end increment

  export int getValue(){
    return value;
  }//end getValue
}//end counter
```

Im Unterschied zu Pascal-FC bietet JavaFC keine getrennte *Export*-Methoden-Deklaration und Methoden-Implementierung an, sondern vereint beide Funktionen[1]. Da für das Beispiel keine *Conditions* benötigt werden, folgt ein kurzes Beispiel, was die Verwendung einer *Condition* zeigt:

```
monitor Cond{
  condition c1;

  export void method1(){
    c1.delay();
  }//end method1

  export void method2(){
    c1.resume();
  }//end method2
}//end Cond
```

Eine wichtige Einschränkung für den Monitor ist, dass der Präprozessor beim Parsen des Monitors überprüft, ob der Anwender das Schlüsselwort *public* verwendet hat. Ist das der Fall, wird der Übersetzungsvorgang abgebrochen, da dieser Monitor gegen die Prinzipien des von Hoare formulierten Monitorkonzept widersprechen würde.

7.3 Überführung in Java-Syntax

Die JavaFC-Implementierung des Monitors nutzt zur Umsetzung die Semaphor-Bibliothek Javas. Die *"concurrent.locks"*-Bibliothek bietet eine vollständige Moni-

[1] Pascal-FC formuliert zunächst nur die *Export*-Methoden-Signatur, ähnlich einem Interface in Java und setzt erst im Monitorrumpf die *Export*-Methode um.

torimplementierung - samt *Conditions* - an, allerdings läßt diese Implementierung keine *UrgentList* zu, wie sie Pascal-FC benutzt.
Aus diesem Grund muss eine eigene Monitorimplementierung entwickelt werden, die eine zweistufige Warteliste unterstützt.
Im Folgenden wird die Umwandlung des JavaFC-Monitors in Java-Code beschrieben.

7.3.1 Monitorrumpf

Die JavaFC Grammatik stellt den Monitor auf die selbe Stufe wie reguläre Klassen, folglich wird der folgende Ausdruck:

```
monitor Counter{
```

folgendermaßen transformiert:

```
class Counter{
  Semaphore secure = new Semaphore(1,true);
  Semaphore lock = new Semaphore(1,true);
  LinkedList urgent = new LinkedList();
```

Erklärung:

- Das Schlüsselwort `monitor` wird durch `class` ersetzt.
- Zusätzlich werden zwei Semaphoren eingefügt, die den Zutritt zum Monitor und die Manipulation an der internen Datenstruktur des Monitors sichern.
- Auf der *LinkedList* werden alle Prozesse abgelegt, die die Kontrolle des Monitors freiwillig abgegeben haben.

7.3.2 Conditions

Nachdem die benötigten Datenstrukturen angelegt sind, werden die *Conditions* angepasst.

```
condition c1;
```

wird entsprechend angepasst.

```
LinkedList c1 = new LinkedList();
```

Erklärung:

- Das Schlüsselwort *condition* wird durch eine `LinkedList` ersetzt.
- Nach dem Bezeichner der *Condition* wird der Konstruktor der Liste eingefügt.

7.3.3 Export-Methoden

Die exportierten Methoden werden vom Monitor nach Außen bekannt gegeben und garantieren gegenseitigen Ausschluss. Dabei können diese Methoden geschachtelt aufgerufen werden. Die folgenden Code:

```
export void increment(){
  value++;
}//end increment
```

erfährt folgende Anpassung:

```
public void increment(){
  try{
    this.lock(); //lock mutex
    value++;
  }finally{
    this.unlock(); //unlock mutex
  }
}//end increment
```

Erklärung:

- Das Schlüsselwort *export* wird durch *public* ersetzt.
- JavaFC fügt nach dem Betreten der Methode einen *try-finally*-Block ein. Dieses Vorgehen ermöglicht innerhalb der Methode die Verwendung eines *return*-Statements, der `finally`-Block gibt nach dem Verlassen der Methode jedoch noch den Monitor frei.

7.3.4 MonitorObject

Die Klasse `MonitorObject` speichert einige interne Daten des Monitors.

```
class MonitorObject{
  private int counter = 0;
  private Semaphore sema = new Semaphore(0);
}
```

Erklärung:

- Die Variable `counter` zählt die Schachtelungstiefe.
- Auf dem Semaphor `sema` legt sich der Prozess schlafen, wenn er auf die `urgentList` kommt.

7.3.5 Lock und Unlock

Die Methode `lock` verwaltet geschachtelte *Export*-Methodenaufrufe und schützt den Zutritt zum Monitor.

```
private void lock(){
  if(active == null
      || !Thread.currentThread().equals(active.thread)){
    lock.acquire(); //monitor lock
    active = new MonitorObject(); //monitor owner
    active.counter = 1;
    active.thread = Thread.currentThread();
  }else{
    active.counter++; //nested monitor lock
  }
}//end lock
```

Erklärung:

- Die Methode überprüft zunächst, ob der Monitor leer ist oder sich der aktuelle Thread noch nicht im Monitor befindet. In diesem Fall wird das *lock*-Objekt des Monitors angefordert. Ist der Monitor leer, kann der aktuelle Thread in den Monitor eintreten, ansonsten muss er auf den `lock`-Semaphor warten.
- Nachdem der Monitor betreten wurde, wird überprüft, ob der aktuelle Thread der Monitorbesitzer ist. In diesem Fall handelt es sich um einen geschachtelten Methodenaufruf und ein Zähler wird erhöht, der die Schachtelungstiefe speichert.
- Handelt es sich nicht um einen geschachtelten *Export*-Methodenaufruf, der aktuelle Prozess in die Warteschlange des Monitors eingereiht.

  Wenn der Monitor nicht belegt ist, darf der aktuell Prozess eintreten und wird als Monitorbesitzer festgelegt.

Die `unlock`-Methode realisiert die Freigabe des Monitors.

```
public void unlock(){
  active.counter--;
  if(active.counter > 0){
    return; //handle nested monitor lock
  }
  if(urgentList.size() > 0){
    active = urgentList.removeFirst(); //change owner
    active.sema.release();
  }else{
    active = null;
    lock.release(); //unlock monitor
```

```
12      }
13  }//end unlock
```

Erklärung:

- Die `unlock`-Methode dekrementiert zunächst den Zähler für die Schachtelungstiefe. Bei einem Wert von größer Null, kehrt die `unlock`-Methode ohne Änderungen am Monitorzustand zurück.
- Ist der Wert gleich Null, wird überprüft, ob ein Thread auf der `urgentList` wartet. In diesem Fall wird dem ersten Thread auf der Liste die Kontrolle des Monitors übertragen.
- Wartet kein Thread auf der `urgentList`, wird der Monitor als frei markiert und das `lock`-Semaphor freigegeben.

7.3.6 Delay und Resume

JavaFC macht aus jeder *Condition* eine *LinkedList*. Bei einem Aufruf von *delay* und *resume* wird diese *LinkedList* als Warteschlange genutzt. Dabei muss beim Aufruf von *resume* oder *delay* folgende Anpassung gemacht werden. In der JavaFC-Syntax wird auf dem *Condition*-Objekt die Methode *delay*, bzw. *resume* aufgerufen.

```
1  c1.delay();
```

da jede *Condition* eine *LinkedList* ist, die weder *delay* noch *resume* anbietet, muss der Aufruf folgendermaßen abgewandelt werden:

```
1  this.delay(c1);
```

Jeder Monitor bietet die Methoden `resume` und `delay` an, denen eine *LinkedList* übergeben wird.

```
1   private void delay(LinkedList con){
2     MonitorObject mo = active;
3     con.add(mo);
4     if(urgentList.size() > 0){
5       active = urgentList.removeFirst();
6       active.sema.release();
7     }else{
8       active = null;
9       lock.release(); //unlock monitor
10    }
11    mo.sema.acquire(); //current thread sleeps
12  }//end delay
```

Erklärung:

- Zunächst wird der aktuelle Prozess in die *LinkedList* eingetragen.

- Der nächste Schritt ist die Überprüfung, ob es einen Prozess auf der `urgentList` gibt, um diesen ggf. aufzuwecken.
- Befindet sich kein Prozess auf der `urgentList`, wird der Monitor als frei markiert.
- Der Thread legt sich, nachdem die Verwaltung des Monitors abgeschlossen ist, auf einem Semaphor schlafen.

Die *resume*-Methode bildet das Gegenstück zur *delay*-Methode und ist folgendermaßen aufgebaut.

```
1  private void resume(LinkedList con){
2    if(con.size() > 0){
3      MonitorObject mo = active;
4      urgentList.add(mo);
5      active = con.removeFirst(); //change owner
6      active.sema.release();
7     mo.sema.acquire(); //current thread sleeps
8    }
9  }//end resume
```

Erklärung:

- Die Methode `resume` überprüft, ob sich ein Prozess auf der übergebenen *LinkedList* befindet. Wenn dies nicht der Fall ist, wird die Methode verlassen - das `resume` wäre in diesem Fall ohne Effekt.
- Sollte sich ein Prozess auf der Warteschlange befinden, wird dieser von der Liste genommen und aufgeweckt. Außerdem trägt sich der aktuelle Prozess in die `urgentList` ein und legt sich im Anschluss schlafen.

Schlafen mit Hilfe eines Semaphors

Jeder Thread legt sich in der JavaFC-Monitorvariante auf einem Semaphor "schlafen", anstatt sich über die *Object*-Methoden *wait* und *notify* schlafen zu legen bzw. aufgeweckt zu werden.
Semaphoren bieten die Möglichkeit festzustellen, ob ein Prozess vor dem eigentlichen Schlafenlegen (mittels *wait*) bereits aufgeweckt worden ist. Das passiert immer dann, wenn ein Prozess einen Anderen aufweckt und der Java-Scheduler diesem neuen Prozess sofort die Kontrolle über den Prozessor übergibt, noch bevor der erste Thread sich schlafen legen konnte. Der neue Prozess weckt nun mittels *notify* den ersten wieder, obwohl dieser nicht schläft - das *notify* bleibt wirkungslos und der erste Thread würde nie wieder aufgeweckt werden, wenn dieser sich im Anschluss schlafen legt. Ein Semaphor verhindert dieses Scenario, denn es "merkt" sich im internen Zähler das Aufwecken.

7.4 Zusammenfassung

Der Monitor stellt eine komplexe Datenstruktur zum Schützen kritischer Programmteile dar. Die klassischen *synchronized* und *wait*, zbw. *notify* Aufrufe werden dabei vollständig durch eine eigene Monitorimplementierung ersetzt, die sich an der Verhaltensweise Pascal-FCs orientiert. Der Präprozessor fügt für jeden Monitor selbstständig Quellcode ein, der die Allokierung des Monitors, sowie deren Freigabe vornimmt.
Die Aufgabe des Monitorschutzes, sowie das Schlafenlegen auf den *Conditions* werden dabei vollständig von Semaphoren übernommen. Auf die Verwendung der *Lock*-Bibliothek zur Umsetzung eines Monitors wurde verzichetet, da das Prozessmanagement der *Lock*-Implementierungen - vor allem in Bezug auf die *UrgentList* - nicht dem Pascal-FC Verhalten entspricht.

8 Ada-Style-Rendevous

In diesem Kapitel soll ein Kommunikationskonzept eingeführt werden, welches in der Programmiersprache *Ada* verwendet wird und ein so genanntes *many-to-one*-Kommunikationsmuster erlaubt.
Bei einem *Ada-Style-Rendevous* wird ein entfernter Methodenaufruf durchgeführt, bei dem der aufgerufene Prozess selbstständig entscheidet, ob und wann der Aufruf behandelt wird (*remote method invocation*). Der Aufrufer blockiert so lange, bis seine Anfrage bearbeitet wurde und nimmt einen möglichen Rückgabewert entgegen.

8.1 Überblick

Das *Ada-Style-Rendevous* besteht aus drei wesentlichen Teilen. Zunächst muss ein Prozess, der ein *Ada-Style-Rendevous* anbietet, ein *Entry* definieren. Dabei handelt es sich um die Bekanntgabe einer Methodensignatur. Die *Entries* werden wie Klassenvariablen angelegt und durch das Schlüsselwort *entry* eingeleitet.
Das *accept*-Statement stellt eine konkrete Implementierung des *Entries* dar. Dabei ist es möglich, dass ein *Entry* in beliebig vielen *accept*-Statements auf unterschiedlichste Weise realisiert werden kann.
Das dritte Element des *Ada-Style-Rendevous* ist eine Warteschlange. Jedes *Entry* kann von beliebig vielen Prozessen aufgerufen werden. Somit kann es passieren, dass mehrere Bewerbungen für ein *Entry* existieren. Damit jede Anfrage bearbeitet werden kann, werden die Prozesse auf einer Warteschlange abgelegt, sodaß jedes *Entry* seine eigene Warteschlange besitzt.

8.2 JavaFC-Syntax

Das *Ada-Style-Rendevous* besitzt zwei Elemente zum einen die *entry*-Deklaration und zum anderen das *accept*-Statement.

8.2.1 Entry

Die folgende Grammatik zeigt die *entry*-Deklaration in BNF:

- EntryDeclaration := entry MethodSignature ;
- MethodSignature := TYPE NAME ((Parameter)*)

Erklärung:

- Die *entry*-Deklarierung wird durch das Schlüsselwort *entry* eingeleitet, gefolgt von einer Methodensignatur[1].
- JavaFC unterstützt keine überladenen *Entries*.

8.2.2 Accept

Ein *Entry* wird durch ein *accept*-Statement umgesetzt. Die Grammatik in BNF sieht wie folgt aus:

- AcceptStatement := accept MethodSignature { (Statement)* }

Erklärung:

- Das *Accept*-Statement wird durch *accept* eingeleitet, gefolgt von einer Methodensignatur. Innerhalb des Statements kann beliebiger Code formuliert werden.

8.2.3 Beispiel

Die `Counter`-Klasse wird für das *Ada-Style-Rendevous* zu einem aktiven Prozess umgewandelt. Außerdem wird für das Beispiel das noch nicht eingeführte *select*-Statement benötigt (↗ siehe Kapitel 10).

```
1 program TurnstileMain{
2   Counter c;
3   cobegin{
4     c = new Counter();
5     for (int i=0; i<2; i++)
6       new Turnstile(c);
7   }
8 }//end TurnstileMain
```

Die Klasse `Turnstile` bleibt unverändert. Die `Turnstile`-Prozesse rufen weiterhin auf einer Instanz der `Counter`-Klasse die Methode `increment` auf.
Die Klasse `Counter` wird folgendermaßen angepasst:

[1] JavaFC kann in der Signatur keine *Exceptions* verarbeiten.

```
process Counter{
  entry void increment();
  entry int getValue();
  private int value = 0;

  public void body(){
    repeat{
      select{
        accept void increment(){
          value++;
        }or accept int getValue(){
          return value;
        }terminate{
          System.out.println("counter: "+value);
        }
      }
    }forever;
  }//end body
}//end Counter
```

8.3 Überführung in Java-Syntax

Die Umsetzung des *Ada-Style-Rendevous* in Java-konformen Code benötigt eine dynamische Datenstruktur, welche die Anfragen für die einzelnen *Entries* speichert, die aufrufenden Prozesse schlafen legt und wieder aufwecken nachdem ihr Auftrag vom aufgerufenen Prozess ausgeführt wurde.

8.3.1 Benötigte Datenstrukturen

Die wichtigste Datenstruktur für das *Ada-Style-Rendevous* ist eine zweidimensionale Warteschlange, in der die Anfragen für jedes *Entry* gespeichert werden. Der Aufbau der Warteschlange ist mit einer "Flatter-Matrix" zu vergleichen, wobei die x-Richtung durch die Anzahl der *Entries* fest vorgegeben ist. Die Ausdehnung in y-Richtung wird durch die Anzahl der Anfragen für jedes *Entry* festgelegt.
Neben dem Schlafenlegen und Aufwecken des Aufrufers hat die Warteschlange auch die Aufgabe, die Übergabeparameter und die Rückgabewerte des *accept*-Statements zu behandeln.
Jede Anfrage wird durch einen *Container* dargestellt, der in die Liste eingefügt wird. Der *Container* speichert alle nötigen Daten einer Anfrage sowie nötige Variablen zum synchronisierten Zugriff auf den *Container*.

```
public class Container{
  public Container next; //next job
```

```
3    public Condition thread;
4    public Object[] items; //parameters
5    public boolean done = false;
6
7    Container(int i){
8      items = new Object[i];
9    }
10 }
```

Erklärung:

- Da die Anzahl an Aufträgen pro *Entry* variieren kann, sind die *Container* in einer einfach-verketteten Liste angelegt, wobei jeder *Container* seinen Nachfolger in der Variablen `next` abspeichert.
- Die *Condition*-Variable `thread` dient als Bedingungsvariable für das Schlafen des aufrufenden Prozesses. Sie ist Bestandteil der "*concurrent.lock*-Bibliothek" und steht nicht in Verbindung mit der *Condition*, die von JavaFC eingeführt wird.
- Das *Object*-Array `items` speichert neben den Übergabeparametern auch den Rückgabewert des *Entries*.
- Die letzte Membervariable `done` verhindert die mehrfache Ausführung des selben Auftrags.

Nachdem der *Container* vorgestellt wurde, folgt nun die eigentliche Liste. Sie verwaltet die *Container*, sowie die Übergabeparameter und Rückgabewerte. Dass die Liste, wie zu Begin des Kapitels erwähnt, zweidimensional ist, spielt hier keine Rolle. Jede Instanz des Datentyps `ParameterList` verwaltet ein eigenes *Entry*. Die Gesamtheit aller *Entries* wird in einem `ParameterList`-Array gespeichert.

```
1 import java.util.concurrent.*;
2
3 public class ParameterList{
4   private Container head, tail, work;
5   private Semaphore read = new Semaphore(0,true),
6                     sync = new Semaphore(1,true);
7   private Lock lock = new ReentrantLock(true);
```

Erklärung:

- Die Referenzen `head` und `tail` speichern den Anfang und das Ende der verketteten Liste. In `work` wird der aktuell bearbeitete Auftrag vermerkt.
- Das Semaphor `read` verzögert den konsumierenden Teil des Rendevous, wenn keine Anfrage für ein *Ada-Style-Rendevous* vorhanden ist.

  Das Semaphor `sync` schützt die Datenstruktur vor nebenläufigem Zugriff.
- Das `lock`-Objekt stellt *Conditions* zur Verfügung, auf denen sich die wartenden Prozesse schlafen legen.

Hinzufügen von Aufträgen

Im Folgenden werden zwei wichtige Methoden der Liste vorgestellt. Die erste Methode setzt neue Aufträge auf die Liste und legt den aufrufenden Prozess schlafen, die zweite Methode liest die entsprechenden Aufträge.

```
1   public Condition add(Object... os){
2     sync.acquire(); //lock data structure
3     Container newC = new Container(os.length+1);
4     int i = 1;
5     for ( Object o: os){
6       newC.items[i++] = o;
7     }
8     //insert Container into list
9
10    lock.lock(); //enter monitor
11    read.release(); //send 'new job'-signal
12    sync.release(); //unlock data structure
13
14    newC.thread = lock.newCondition();
15    newC.thread.await(); //sleep
16    lock.unlock();
17    return newC.thread;
18  }//end add
```

Erklärung:

- Zu Beginn wird die Datenstruktur mittels `sync` geschützt und ein neuer Container erstellt, in den die Übergabeparameter hineingeschrieben werden. Dabei bleibt die erste Position frei, denn in ihr wird später der Rückgabewert gespeichert.
- Im Anschluss wird der Container an das Ende der Liste eingefügt.

  Außerdem wird eine Ressource auf dem `read`-Semaphor freigegeben, um anzuzeigen, dass ein neuer Auftrag vorliegt.

- Nachdem eine *Condition* des Monitors erstellt wurde, legt sich der Aufrufer auf dieser schlafen, bis der Auftrag erledigt ist.

Auslesen von Aufträgen

Die nächste Methode beschreibt das Auslesen eines unbearbeiteten Auftrags aus der Liste.

```
1  public Object[] read(){
2    read.acquire(); //wait for new job
3    try{
4      sync.acquire();
5      Container c = head;
6      while(c.done == true){
7        c = c.next; //search next valid job
8      }
9      work = c;
10     return c.items;
11   }finally{
12     sync.release();
13   }
14 }//end read
```

Erklärung:

- Die Methode kann nur betreten werden, wenn vorher ein Element auf die Liste gesetzt wurde und eine Ressource des `read`-Semaphors verfügbar ist.

- Im zweiten Schritt wird die Datenstruktur mittels des `sync`-Semaphors geschützt und die Liste über eine Schleife durchlaufen. Die Aufträge, die bereits erledigt sind, werden dabei übersprungen[2].

- Danach wird das Array mit den Übergabeparametern an den Prozess zurück gegeben, der das *Entry* anbietet.

Neben dem Anlegen und dem Auslesen von Aufträgen bietet die *ParameterList* noch weitere Methoden an. Zum einen `putRetValue`, welche den Rückgabewert des *accept*-Statements in die *ParameterList* schreibt, sowie die `readRetValue`-Methode, welche diesen ausliest, zum anderen eine `remove`-Methode, welche den Auftrag, nachdem der Rückgabewert ausgelesen ist, von der *ParameterList* entfernt.

[2]Gelöscht werden die Aufträge erst, wenn der aufrufende Prozess das Endergebniss entgegen genommen hat und seinerseits den Auftrag von der Liste entfernt. Die Ursache ist die zeitliche Verschwänkung der Prozesse. Ein aufgeweckter Prozess muss möglicherweise noch auf die Zuteilung des Hauptprozessors warten, bis er wirklich weiterarbeiten kann

8.3.2 Anpassung des entry- und des accept-Statements

Der Präprozessor entfernt alle *entry*-Deklarationen aus dem Quelltext. Allerdings speichert er die Methodensignatur in einer internen Datenstruktur. Nachdem das Löschen der *entry*-Deklaration abgeschlossen ist, werden Stellvertretermethoden erstellt, die immer aufgerufen werden, wenn ein Prozess einen *Entry* benutzen möchte. Die Stellvertreter speichern die Übergabeparameter und hängen den Auftrag in die Parameterliste ein. Das folgende Beispiel:

```
process Counter{
  entry int getValue();
  //...

  public void body(){
    //...
    accept int getValue(){
      return value;
    }
  }//end body
}//end Counter
```

wird in zwei Teilanalysen aufgeteilt.

Entry

Das *entry*-Statement wird vollständig aus dem Quellcode entfernt und an dessen Stelle folgender Code eingefügt.

```
private ParameterList[] list = new ParameterList[1];

public int getValue(){
  Condition con = null;
  try{
    Object[] oa = new Object[1]; //array for parameters
    //process the parameters
    con = list[1].add(oa); //add job to list
    int retVal = list[0].readRetValue(con);
    return retVal;
  }finally{
    list[0].remove(con);
  }
}
```

Erklärung:

- Im ersten Schritt wirde die *entry*-Deklaration aus dem Quelltext entfernt und ein Array aus `ParameterLists` angelegt. Die Größe dieses Arrays richtet sich

dabei nach der Anzahl der angebotenen *Entries*.

- Der Präprozessor generiert die Stellvertretermethode `getValue`, welche die gleiche Signatur wie das *Entry* besitzt. Innerhalb der Methode wird zunächst ein *Object*-Array angelegt, welches die Übergabeparameter speichert.

  Das Speichern der Parameter im Array wird dynamisch vom Präprozessor erzeugt, dabei spielen die Anzahl der Parameter sowie deren Name eine Rolle.

- Im Anschluss ruft der Stellvertreter die `add`-Methode der `ParameterList` auf. Diese Methode registriert den Auftrag in der Liste und legt den aufrufenden Prozess schlafen, bis der Auftrag abgearbeitet wurde.

- Nachdem der Auftrag abgearbeitet wurde und der Thread aufgeweckt wurde, wird diesem eine *Condition* zurückgegeben, welche als Identifikationsmerkmal des entsprechenden Auftrags dient.

- Sollte das *accept*-Statement einen Rückgabewert liefern, wird dieser im Anschluss mit Hilfe der `readRetValue`-Methode abgefragt und an den Aufrufer zurückgegeben.

- Als letztes wird der Auftrag aus der Liste über die `remove`-Methode entfernt.

Accept

Das *accept*-Statement wird in JavaFC mit dem Schlüsselwort *accept* eröffnet, gefolgt von der Methodensignatur des *Entries*, der an dieser Stelle bearbeitet werden soll.

```
1  public void body(){
2    //...
3
4    Object[] ret = list[0].read();
5    //process parameters
6    //implementation
7    list[0].putRetValue(value);
8  }
```

Erklärung:

- Das *accept*-Statement wird zunächst entfernt und die `read`-Methode der *ParameterList* aufgerufen.

  Nachdem `read`-Methode zurückgekehrt ist, steht ein *Object*-Array zur Verfügung, in dem sämtliche Übergabeparameter abgelegt sind.

- Im Anschluss werden Variablen angelegt, die vom Typ und dem Namen her der *entry*-Deklaration entsprechen. Es ist wichtig, dass die Typen sowie die Namen genau der *entry*-Vorgabe entsprechen, da sich die *accept*-Implementierung auf die Variablen bezieht.

- Nachdem das Auspacken der Variablen abgeschlossen ist, folgt die Implementierung des eigentlichen Statements.
- Sollte das *accept*-Statement über eine *return*-Anweisung verfügen, wird das *return* gelöscht und die `putRetValue`-Methode mit dem Inhalt des *return*-Statements aufgerufen.

8.4 Zusammenfassung

Das Konstrukt der *remote method invocation* bietet dem Programmierer eine gute Möglichkeit, Methoden bzw. Dienstleistungen öffentlich zur Verfügung zu stellen. Dabei wird eine Client-Server-Architektur nachempfunden. Allerdings werden alle syntaktischen Unannehmlichkeiten vor dem Anwender versteckt, so dass die Kommunikation über *remote invocation* wie ein lokaler Funktionsaufruf aussieht.

9 Kanal

Das Konzept des Kanal läßt sich zur Kommunikation und Synchronisation verwenden. Es handelt sich um eine Punkt-zu-Punkt-Kommunikation, bei der immer zwei Kommunikationspartner vorausgesetzt werden. Die Daten werden nicht zwischengespeichert, sondern direkt vom Sender an den Empfänger übermittelt. Der Moment der Datenübertragung wird im Folgenden als Rendevous bezeichnet. Folgende Grafik zeigt einen beispielhaften Ablauf für eine Datenübertragung mit Hilfe eines Kanals:

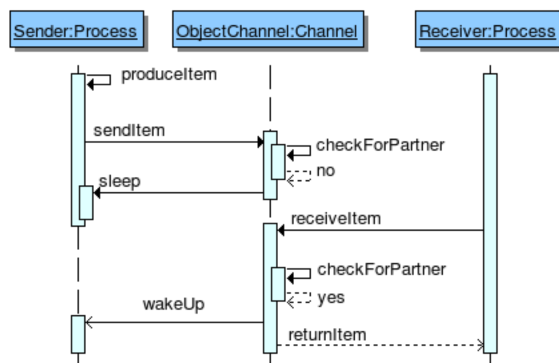

Abbildung 9.1: Datenaustausch über Kanäle.

9.1 Überblick

Der Kanal ist ein synchrones Kommunikationsmittel, denn sowohl Sender als auch Empfänger müssen im Moment des Datenaustauschs innerhalb ihrer jeweiligen Sende- oder Empfangsmethode sein. Ist das nicht der Fall, blockieren die Prozesse, bis der andere Kommunikationspartner am Rendevous angekommen ist.
Eine weitere Eigenschaft des Kanals ist seine Typsicherheit. Das Konzept des Kanals sieht es vor, nur Daten genau eines Datentyps zu übermitteln. Bei einem Verstoß gegen diese Vorgabe wird ein Laufzeitfehler ausgelößt.
Eine weitere Einschränkung des Kanals ist, dass nur jeweils ein Sender und ein Empfänger gleichzeitig in der Sende- oder Empfangsmethode sein dürfen, bis der Datenaustausch stattgefunden hat. Sollte ein Prozess die Sendemethode aufrufen, obwohl sich bereits ein Prozess in der Methode befindet, wird ebenfalls ein Laufzeitfehler ausgelößt.

Kanäle lassen sich neben der Datenübertragung auch für die Synchronisation zweier Prozesse verwenden. Dafür bietet der Kanal den *synchronous*-Datentyp an, welcher eine Synchronisation ohne Datenaustauch erlaubt. Die Datentypen *synchronous* oder kürzer *sync*[1] sind dabei reservierte Schlüsselworte der Sprache ähnlich wie *int* oder *char*.
In Pascal-FC existiert neben dem *synchronous*-Datentyp noch ein Schlüsselwort *any*. Dabei stellt *any* ein Dummy-Datum dar, was bei jeder Kommunikation über einen *synchronous*-Kanal übertragen wird. In JavaFC wird dieses Dummy-Datum nicht benötigt.

9.2 JavaFC-Syntax

In JavaFC kann der Kanal wie ein primitiver Datentyp verwendet werden, mit der Ausnahme, dass auf dem Kanal - anders als in Java üblich - Methodenaufrufe möglich sind.

9.2.1 Channel

Der Kanal hat folgenden syntaktischen Aufbau:

- ChannelDeclaration := channel Modifier Channel (, Channel)* ;
- Channel := NAME [= new channel Modifier]
- Modifier := [ChannelDataType] [[ ]]
- ChannelDataType := (TYPE | sync | synchronous)
- ChannelStatement := NAME . send (Expression) | NAME . receive ( )

Erklärung:

- Die Deklarierung eines Kanals wird mit dem Schlüsselwort *channel* eingeleitet. Optional kann danach angegeben werden, welcher Datentyp vom Kanal übertragen werden soll. Möchte der Anwender ein Array von Kanälen anlegen, folgt auf den Kanaltyp eine entsprechende Klammerung.
- Nach dem Kanalbezeichner kann die Initialisierung folgen.
- Die Initialisierung beginnt Java-typisch mit einem Gleichheitszeichen und dem Schlüsselwort `new`. Danach folgt der Datentyp und die optionale Klammerung eines Arrays.

[1] Sowohl *synchronous* als auch *sync* haben in JavaFC die gleiche Funktion. Der Datentyp *sync* ist lediglich eine verkürzte Schreibweise von *synchronous*.

9.2.2 Beispiel

Zur Umsetzung des Beispiels werden alle drei Klassen - `Turnstile`, `TurnstileMain` und `Counter` angepasst. Folgende Änderungen hat die Klasse `TurnstileMain` erfahren:

```
program TurnstileMain{
  channel chan1 = new channel(int);
  channel chan2 = new channel(int);

  cobegin{
    new Counter(chan1, chan2);
    new Turnstile(chan1);
    new Turnstile(chan2);
  }
}//end TurnstileMain
```

Die Klasse `Counter` ist nun kein passives Element mehr, sondern ein aktiver Prozess, dem zwei Kanäle übergeben werden. Der `Turnstile`-Klasse wird statt der `Counter`-Instanz nun ein Kanal übergeben, was zu folgenden Änderungen innerhalb der Klasse führt:

```
process Turnstile{
  channel(int) chan;

  Turnstile(channel(int) c){
    chan = c;
  }

  public void body(){
    for(int i=0; i<20; i++)
      chan.send(1);
  }//end body
}//end Turnstile
```

Die Klasse `Counter` muss die beiden Kanäle nun verwalten:

```
process Counter{
  channel(int) chan1;
  channel(int) chan2;
  private int value = 0;

  Counter(channel(int) c1, channel(int) c2){
    chan1 = c1;
    chan2 = c2;
  }

```

```
11     public void body(){
12       repeat{
13         select{
14             value += chan1.receive(); //read 1st channel
15           or
16             value += chan2.receive(); //read 2nd channel
17           terminate
18             System.out.println("counter: "+value);
19         }
20       }forever;
21     }//end body
22   }//end Counter
```

Das *select*-Statement liest die beiden Kanäle aus, bis sich die entsprechenden Prozesse beenden. Danach gibt der `Counter` den Zählerstand aus und verläßt die Endlosschleife. Das Auswählen eines der beiden Kanäle geschieht dabei völlig zufällig (insofern beide Kanäle belegt sind), ausschlaggebend dafür ist, dass der entsprechende `Turnstile`-Prozess die Sendemethode aufgerufen hat.
In diesem Beispiel hätte man die Kanäle an den `Counter`-Prozess auch in einem Array übergeben können. Dann muss die `TurnstileMain`-Klasse wie folgt aussehen:

```
1   program TurnstileMain{
2     channel[] channelArray = new channel[2];
3     for (int i=0; i<2; i++)
4       channelArray[i] = new channel(int);
5
6     cobegin{
7       new Counter(channelArray);
8       new Turnstile(channelArray[0]);
9       new Turnstile(channelArray[1]);
10    }
11  }//end TurnstileMain
```

Die `Counter`-Klasse muss nun einer kleinen Änderung unterzogen werden, wohingegen die `Turnstile`-Klasse unverändert bleiben kann.

```
1   process Counter{
2     channel(int)[] channelArray;
3     private int value = 0;
4
5     Counter(channel(int)[] array){
6       channelArray = array;
7     }
8     //skip body method
9   }//end Counter
```

Der Rest der Klasse bleibt nahezu unverändert. Der Zugriff auf die Kanäle muss innerhalb der Alternativen des *selects* an das Array angepasst werden.

9.3 Überführung in Java-Syntax

JavaFC ersetzt jeden Aufruf des Schlüsselworts *channel* durch die entsprechende *Channel*-Bibliothek. Da diese Bibliothek als generische Klasse umgesetzt ist, müssen Kanal-Arrays gesondert behandelt werden, denn eine normale Array-Deklaration mit einem generischen Datentyp ist in Java nicht erlaubt [9]. JavaFC nimmt folgende Änderungen vor:

```
1  channel chan1 = new channel(int);
```

wird umgewandelt in:

```
1  Channel chan1 = new Channel<Integer>();
```

Erklärung:

- Das Schlüsselwort *channel* wird durch einen Bibliotheksaufruf der Klasse `Channel` ersetzt.

  Diese Bibliothek wird im Kapitel 9.3.1 genauer beschrieben.

- Der durch den Kanal zu übertragende Datentyp wird als generische Typinformation an die *Channel*-Bibliothek weitergegeben.

- Sollte der Anwender primitive Datentypen verwenden, ersetzt JavaFC diese automatisch durch die entsprechenden Wrapperklassen.

Im Folgenden wird die Umsetzung von Kanal-Arrays behandelt. Den Quelltext:

```
1  channel[] channelArray = new channel[2];
2
3  for (int i=0; i<2; i++)
4    channelArray[i] = new channel(int);
```

verändert JavaFC folgendermaßen:

```
1  Channel[] channelArray = new Channel[2];
2
3  for(int i=0; i<2; i++)
4    channelArray = new Channel<Integer>();
```

Erklärung:

- Wenn bei einer Referenz auf einen Kanal der zu übertragende Datentyp angegeben wird, wird auch die *Channel*-Klasse mit diesem Datentyp aufgerufen.

- Bei der Verwendung eines Arrays wird die Typinformation hingegen nicht übernommen, da Arrays mit generischen Typen von Java nicht unterstützt werden[2].

9.3.1 Channel-Klasse

Für die Implementierung des Kanals kann nicht auf eine Java-Bibliothek zurückgegriffen werden, weshalb ein neuer Implementierungsansatz diskutiert werden muss. Da ein Kanal unterschiedliche Datentypen transportieren muss, ist eine generische Klasse der naheliegendste Ansatz. Dabei stehen allerdings nur komposite Datentypen zur Verfügung, denn primitive Datentypen werden nicht durch *Generics* unterstützt.
Auf Grund des *synchronous*-Datentyps werden zwei verschiedene Sendemethoden und auch zwei Empfangsmethoden benötigt. Die beiden Sendemethoden sind einfach zu realisieren, während der *synchronous*-Datentyp auf eine Sendemethode ohne Übergabeparameter zurückgreift, nutzen alle anderen Datentypen eine Sendemethode, der ein generischer Datentyp übergeben wird. Dieser Ansatz läßt sich leider nicht auf die Empfangsmethode anwenden.
Während die Empfangsmethode für die normale Kanalkommunikation einen generischen Datentyp T zurückgibt, müsste die Methode für den *synchronous*-Datentyp den Rückgabetyp *void* haben. Eine Unterscheidung im Rückgabetyp, bei ansonsten identischer Methodensignatur ist jedoch nicht erlaubt. Die einfachste Lösung ist für beide Fälle die gleiche Empfangsmethode zu verwenden, allerdings wird bei einem *synchronous*-Kanal einfach *null* zurück gegeben.
Das Grundgerüst der Kanal-Bibliotheksklasse nutzt einige Datenstrukturen für die Verwaltung und Synchronisation der Prozesse. Der Rumpf des Kanal sieht wie folgt aus:

```
1  import java.util.concurrent.Semaphore;
2
3  public class Channel<T>{
4    private Semaphore put = new Semaphore(1),
5                      get = new Semaphore(1),
6                      signalput = new Semaphore(0),
7                      signalget = new Semaphore(0);
8    private T toSend;
9
10   class Synchronous{ }
```

Erklärung:

- Die *Channel*-Klasse nutzt insgesamt vier Semaphoren für die Synchronisation. Dabei dienen `signalput` und `signalget` zur Ereignissychronisation.

[2] Ein Compilieren würde mit dem Fehler "*illegal: generic array creation*" abgebrochen werden.

- Die Variable `toSend` bildet einen Zwischenpuffer für das zu übertragende Datum[3].

- Um den *synchronous*-, bzw. *sync*-Datentyp auf einen existierenden Datentyp abzubilden, wird eine interne `Synchronous`-Klasse in den *Channel* integriert.

Sendemethoden

Wie bereits erwähnt, muss der Kanal zwei unterschiedliche Sendemethoden besitzen. Einen für die normal Kommunikation zweier Kanäle und einen anderen für die Synchronisation zweier Prozesse mittels *synchronous*-Datentyp. Die Sendemethoden haben folgenden Aufbau:

```
1  public void send(){
2    if(! put.tryAcquire() )
3      throw new ChannelException();
4    signalget.release(); //sender ready
5    signalput.acquire(); //wait for receiver
6    put.release();
7  }
8
9  public void send(T o){
10   if(! put.tryAcquire() )
11     throw new ChannelException();
12   toSend = o;
13   signalget.release(); //sender ready
14   signalput.acquire(); //wait for receiver
15   put.release();
16 }
```

Erklärung:

- Die Sendemethoden sichern sich zunächst das `put`-Semaphor. Ist die Ressource des Semaphors vergeben, dann befindet sich bereits ein Prozess im Sendevorgang und die Methode wirft eine *Exception*.

- Im Anschluss wird das zu übertragende Datum in die `toSend`-Variable geschrieben und über das `signalget`-Semaphor dem Empfänger signalisiert, dass das zu übertragende Datum vorliegt.

- Nachdem dem Empfänger ein Signal gesendet wurde, legt sich der Prozess auf dem `signalput`-Semaphor schlafen.

[3] Die Variable `toSend` dient zum Austausch der Daten zwischen der Sende- und Empfangsmethode zum Zeitpunkt des Rendevous.

Empfangsmethoden

Die Empfangsmethode stellt das Gegenstück zur Sendemethode dar.

```
public T receive() {
  try{
    if(! get.tryAcquire() )
      throw new ChannelException();
    signalget.acquire(); //wait for sender
    T t = null;
    if (T != Synchronous)
      t = toSend;
    signalput.release(); //signal receiver ready
    return t;
  }finally{
    get.release();
  }
}
```

Erklärung:

- Zunächst sichert sich der Prozess die einzige Ressource des `get`-Semaphors.
- Danach wartet der Prozess auf dem `signalget`-Semaphor, bis der Sender das entsprechende Signal schickt, dass das zu übertragende Datum abgelegt wurde.
- Im Anschluss wird das Datum auf die lokale Variable `t` geschrieben. Beim *synchronous*-Datentyp bleibt `t` uninitialisiert.
- Schließlich wird ein Signal an den Sender geschickt. Damit wird zum Ausdruck gebracht, dass das Datum erfolgreich übertragen wurde. Danach können beide Prozesse weiterlaufen.

9.3.2 Channel-Exception

Da die Sende- und Empfangsmethode eine *Exception* werfen können, muss diese nun vom Anwender behandelt werden. Eine Lösung wäre, dass JavaFC selbstständig einen *try-catch*-Block um die Kanalaufrufe setzt. Das würde jedoch dem Anwender verschleiern, dass er den Kanal falsch verwendet hat. Eine Lösung für dieses Problem ist die Klasse *RuntimeException*.

```
class ChannelException extends RuntimeException{
  public ChannelException(){
    super();
  }
}
```

Diese Unterklasse der *Exception* verlangt keinen *try-catch*-Block, liefert bei einem Fehler jedoch trotzdem eine *Exception*, vergleichbar mit einer *NullPointer-Exception* (die ebenfalls nicht explizit behandelt werden muss).

9.4 Zusammenfassung

Der Kanal ist ein Konstrukt, das Synchronisation und Kommunikation kombiniert. Durch die erzwungene paarweise Verwendung von *send* und *receive* synchronisieren sich beide Threads automatisch aufeinander.
Der Kanal wird durch eine generische Klasse realisiert, der der zu übertragende Datentyp mitgegeben wird. Die JavaFC-Grammatik erzwingt einen Datentyp für jeden Kanal. Das schließt Kanäle aus, die beliebige Daten übertragen können, was das exakte Verhalten von Pascal-FC wiederspiegelt.
Eine weitere Einschränkung für Kanäle ist, dass immer nur ein Prozess gleichzeitig die Sende- oder Empfangsmethode aufrufen darf. Ein Prozess, der dagegen verstößt, wird eine *Exception* auslösen.

10 Selektives Warten

Mit dem selektiven Warten wird in Java ein Instrument zur Erzeugung eines nichtdeterministischen Kontrollflusses eingeführt. Bei nachrichtenbasierten Programmiersprachen ist dieses *select*-Statement mit dem *if*-Statement einer sequentiellen Programmiersprache zu vergleichen.

10.1 Überblick

Das *select*-Statement ist ähnlich aufgebaut wie das *switch-case*-Statement. Das *select*-Statement besitzt verschiedene Alternativen, von denen nur jeweils eine Alternative ausgewählt und ausgeführt werden kann. Jede Alternative kann sich dabei im Zustand "offen" oder "geschlossen" befinden, was darüber entscheidet, ob eine alternative ausgewählt werden kann. Folgende Bedingungen haben Einfluss auf den Zustand einer Alternative:

- der Wahrheitswert des *bool'schen* Ausdruck eines *Guards*,
- die verfügbaren Ressourcen eines Semaphors,
- ein vorhandener Kommunikationspartner eines Kanals
- oder ein vorhandener Auftrag für ein *Entry*.

Trifft eine Bedingung nicht zu, gilt die Alternative als geschlossen und kann nicht ausgewählt werden.
Nach der Feststellung, welche Alternativen offen und welche geschlossen sind, wird zufällig eine offene Alternative ausgewählt und ausgeführt. Neben den Alternativen stehen drei weitere Schlussalternativen zur Verfügung, von denen pro *select*-Statement genau eine genutzt werden kann. Die Schlussalternativen werden nur ausgeführt, wenn keine andere Alternative des *select*-Statements geöffnet ist.
Die *else*-Alternative wird unmittelbar ausgeführt, wenn alle anderen Alternativen geschlossen sind. Wird auf den Einsatz des *else* verzichtet und alle anderen Alternativen sind geschlossen, wird ein Laufzeitfehler ausgelößt.
Die *timeout*-Alternative, läßt den Prozess eine vorgegebene Zeit warten, bis die Alternative ausgeführt wird.
Die letzte Möglichkeit ist die *terminate*-Schlussalternative, welche so lange wartet, bis sich eine Alternative öffnet oder bis kein anderer Prozess mehr arbeitet, woraufhin sich der Prozess, der das *select*-Statement bearbeitet, ebenfalls beendet.

10.2 JavaFC-Syntax

10.2.1 Select

Im Folgenden wird die Grammatik des *select*-Konstrukts in BNF analysiert.

- Select := select { SelectBody }
- SelectBody := SelectEntry (or SelectEntry)* [FinalAlternative]
- SelectEntry := [Guard =>] EntryBody [: Statement]
- Guard := when (Expression)
- EntryBody := SyncStatement | ReplicateEntry
- SyncStatement := SemaphoreStatement | ChannelStatement
  | AcceptStatement
- ReplicateEntry := replicate (Init ; [Check] ; [Update]) Statement
- FinalAlternative := (timeout INTEGER | else | terminate) Statement

---

Erklärung:

- Das Statement wird vom Schlüsselwort *select* eingeleitet. Der Rumpf des Statements wird von einem Klammernpaar umschlossen.
- Der Rumpf wird von mindestens einer Alternative und einer optionalen Schlussalternative gebildet, wobei Alternativen über das Schlüsselwort *or* von einander abgegrenzt werden.
- Eine Alternative unterscheidet sich noch einmal in eine normale oder eine *replicate*-Alternative[1]. Normale Alternativen können von einem *Guard* mit dem Schlüsselwort *when* eingeleitet werden, gefolgt von einem bool'schen Ausdruck und einem Pfeil.
- Die nächste Anweisung ist der synchronisierende Teil der Alternative; hier müssen Kanal-, Semaphor- oder Rendevousoperationen durchgeführt werden. Dieses Statement ist der einizge vorgeschriebene Teil jeder Alternative.
- Nach der synchronisierenden Anweisung können weitere Anweisungen folgen. Diese Anweisungen sind optional und werden mit einem Doppelpunkt eingeleitet.

[1] Eine *replicate*-Alternative fasst verschiedene Alternativen zusammen, die nur auf unterschiedliche Kanäle zugreifen.

- Neben der normalen Alternative bietet das *select*-Statement auch eine *replicate*-Alternative an, welche vom Schlüsselwort *replicate* eingeleitet wird, gefolgt von einem Klammerausdruck. Der Inhalt des Klammerausdrucks ähnelt dem einer *for*-Schleife, mit der Ausnahme, dass die Initialisierung des Schleifenzählers zwingend innerhalb der Schleife geschehen muss.

  Nach dem Klammerausdruck folgt ein Statement, bei dem der Zugriff auf ein Kanal-Array vorgeschrieben ist.

- Den Abschluss des Statements bildet gegebenenfalls eine von drei Schlussalternativen.

10.2.2 Beispiel

Die Möglichkeiten, das Gartenbeispiel mit Hilfe eines *select*-Statements umzusetzen, wurde bereits in den Kapiteln 8 und 9 vorgestellt. Da die *select*-Anweisung kein eigenständiges Synchronisations- oder Kommunikationselement darstellt, ist sie meist nur ein Steuerkonstrukt, wie beispielsweise ein *if*-Statement.
Um die unterschiedlichen Möglichkeiten der *select*-Anweisung zu demonstrieren, folgt ein abstraktes Beipiel, was keine wirkliche Funktionalität bietet, allerdings viele neue Mechanismen von JavaFC zu einem Beispiel kombiniert:

```
1  entry boolean function1(int i); //entry
2  channel[] ch1 = new channel[2]; //channel array
3  channel(int) c1 = new channel(int);
4  semaphore semaphore1(1); //semaphore
5  channel channel1(int);
6
7  public void body(){
8    select{
9      when (/*boolean Expression*/) => semaphore1.down() : {
10       //alternative1
11     }
12     or c1.send(1);
13     or replicate (int i=0; i<2; i++){
14       ch1[i].send(i);
15     }
16     or accept boolean function1(int i){
17       //accept alternative
18       return true;
19     }
20     timeout 4000{
21       //timeout
22     }
23   }
24 }//end body
```

Im Folgenden wird die Sematik des *select*-Stamtents erläutert, außerdem werden vier Besonderheiten des Statements noch einmal genauer erklärt.

10.2.3 Sematik des select-Statements

Das *select*-Statement verwaltet verschiedene Alternativen, die im gegenseitigen Ausschluss zueinander stehen - vergleichbar mit einem *switch-case*-Statement.
Ob eine Alternative ausgeführt werden kann, entscheidet zunächst der *Guard* jeder Alternative. Besitzt die Alternative keinen *Guard*, wird angenommen, dass der *Guard* immer wahr ist. Zum Beispiel:

```
1  when (true)
```

Nachdem der *Guard* jeder Alternative überprüft wurde, steht fest, welche Alternativen betreten werden können. Allerdings wird in einem zweiten Schritt überprüft, ob die Ressourcen, die benötigt werden um die Alternative auszuführen, auch vorhanden sind. Nur wenn sowohl der *Guard* erfüllt ist und die benötigten Ressourcen vorhanden sind, kann die Alternative ausgeführt werden.
Die benötigte Ressource zum Ausführen der Alternative können ein Kanal, auf dem bereits ein Kommunikationspartner wartet, ein Semaphor mit ausreichenden Ressourcen oder ein *Ada-Style-Rendevous* sein.
Nachdem zur Laufzeit sowohl *Guard* als auch benötigte Ressourcen überprüft wurden, wird zwischen allen offenen Alternativen zufällig eine ausgewählt und ausgeführt.
Wenn der *Guard* einer Alternative offen ist, gilt die Alternative. Sollte diese Alternative ausgewählt und ausgeführt werden, kann sie denoch blockieren, da das Synchronisationsstatement möglicherweise nicht sofort ausgeführt werden kann (z.B. durch fehlende Resourcen eines Semaphors oder wegen eines fehlenden Auftrags eines *Ada-Style-Rendevous*).
Besitzt das *select*-Statement keine Schlussalternative und sind alle *Guards* geschlossen, wird ein Laufzeitfehler ausgelößt.

10.2.4 Anmerkungen zur replicate-Alternative

Die Syntax einer *replicate*-Alternative setzt eine Initialisierung eines Schleifenzählers voraus. Es ist nicht möglich eine Variable zu verwenden, die zu einem früheren Zeitpunkt des Programmverlaufs angelegt wurde. Eine weitere Einschränkung des *replicates* ist die Verwendung eines Kanal-Arrays innerhalb der *replicate*-Alternative. Der Zugriff auf dieses Array muss allerdings nicht zwingend das erste Statement innerhalb der Alternative sein.
Dass ein Fehlen eines Kanal-Arrays einen Fehler liefert, liegt an der besonderen Bedeutung des *replicates*, denn es greift nicht wie eine *for*-Schleife auf alle Elemente des Arrays zu. Es wandelt das folgende Statement:

```
or replicate (int i=0; i<2; i++){
    ch1[i].send(i);
}
```

entsprechend intern um:

```
or
    ch1[0].send(0);
or
    ch1[1].send(1);
```

10.2.5 Anmerkung zur Verwendung von Kanälen

Eine Alternative, die auf einen Kanal zugreift, gilt nur dann als offen, wenn bereits ein Kommunikationspartner im Kanal wartet. Der Grund dafür ist die Verhinderung einer Verklemmung des *select*-Statements, was innerhalb einer Empfangs- oder Sendemethode auf einen nicht mehr vorhandenen Kommunikationspartner wartet.
Dieses Verhalten führt dazu, dass eine Kommunikation zweier *select*-Statements über einen Kanal nicht durchgeführt wird, da der jeweils andere Kommunikationspartner immer auf das Eintreten des anderen Prozesses in den Kanal wartet.

10.2.6 Anmerkungen zur terminate-Alternative

Die Funktionsweise der *terminate*-Alternative kann ebenfalls eine unerwünschte Konsequenz nach sich ziehen. So wartet ein *select*-Statement innerhalb des *terminate's* bis sich eine Alternative öffnet oder sich alle anderen Prozesse beendet haben, bis es sich selbst beendet.
Diese Verhaltensweise entspricht nicht dem Vorbild von Pascal-FC. Hier wird der Prozess, der sich im *terminate*-Zustand befindet erst beendet, wenn sich alle anderen Prozesse entweder beendet oder auch im Zustand *terminate* befinden.
In JavaFC führt das zu dem Fehler, dass zwei Prozesse, die sich mittels *terminate* beenden wollen, nie beendet werden, da jeder Prozess auf die Beendigung des jedweils Anderen wartet. Eine Behebung des Fehlers wird im Kapitel 11 diskutiert.

10.2.7 Anmerkung zur timeout-Alternative

Die *timeout*-Alternative verlangt als Zeitangabe zwingend einen Integer. Dieser Wert - angegeben in Millisekunden - bestimmt die Zeit, die das Statement auf eine sich öffnende Alternative wartet. Sollte diese Zeitspanne verstreichen, wird das Statement beendet.
Aufgrund der internen Funktionsweise der *timeout*-Alternative kann an dieser Stelle keine Variable verwendet werden. Außerdem wird die tatsächliche Wartezeit geringfügig höher sein, als der angegebene Zeitwert.

10.3 Überführung in Java-Syntax

Bei der Überführung des *select*-Statements in regulären Java-Code muss nahezu das komplette *select*-Statement ausgetauscht und Code generiert werden, der die Analyse der einzelnen Alternativen vornimmt, sowie die Entscheidung darüber trifft, welche offene Alternative ausgeführt wird.
Das *select*-Statement wird in ein *switch-case*-Statement umgewandelt. Jede *case*-Alternative stellt dabei eine Alternative des *select's* dar. Die Schlussalternativen werden durch die *default*-Alternative dargestellt.
Ein *switch-case*-Statement entscheidet anhand einer Bedingungsvariablen, welche Alternative betreten wird. Bei der Umwandlung des *select*- in ein *switch-case*-Statement muss nun eine solche Bindungsvariable eingeführt werden. Der Präprozessor nummeriert die Alternativen durch und erzeugt Code, der vor dem Betreten des *switch-case*-Statements entscheidet, welchen Wert die Variable annimmt.
Eine Alternative kann nur betreten werden, wenn diese offen ist. Deshalb muss zunächst überprüft werden, welche Alternativen offen. Im Anschluss wird zufällig eine offene Alternative ausgewählt, deren Wert die Bedingungsvariable des *switch-case*-Statements annimmt.

10.3.1 Anpassung des select-Statements

Der syntaktische Aufbau des *switch-case*-Statements ist dem des *selects* sehr ähnlich. Beide Statements akzeptieren beliebig viele Alternativen, jedoch nur eine Schlussalternative. Bei der Verwendung der *break*-Anweisung führt das *switch-case*-Statement pro Aufruf auch nur eine Anweisung aus. Folgendes Beispiel:

```
1  select{
2    when (/*boolean Expression*/){
3      semaphore1.down();
4      channel1.send(1);
5    }
6    or replicate (int i=0; i<2; i++){
7      ch1[i].send(i);
8    }
9    or accept boolean function1(int i){
10     //accept alternative
11     return true;
12   }
13   timeout 4000
14     //timeout
15 }
```

wird vom Präprozessor entsprechend umgewandelt:

```
1  switch ( selectedIndex ){
```

```
2      case 0:   //alternative1
3                break;
4      case 1:   //replicate alternative
5                break;
6      case 2:   //accept alternative
7                break;
8      default:  //timeout alternative
9                break;
10   }
```

Erklärung:

- Das *select*-Statement wird durch ein *switch-case*-Statement ersetzt. Die Variable entscheidet dabei, welche Alternative ausgewählt wird.
- Die Schlussalternative wird durch den *default-case* dargestellt.
- *Guards*, *replicate*-Statement oder das *accept*-Statment werden vollständig entfernt und nur der auszuführende Code bleibt innerhalb der Alternativen erhalten.

10.3.2 Benötigte Datenstrukturen

Um eine Entscheidung darüber treffen zu können, welchen Wert die Variable `selectedIndex` annimmt, muss bereits vorher geklärt werden, welche Alternativen offen sind und welche der offenen Alternative für die Ausführung ausgewählt worden ist. Dafür werden einige Datenstrukturen benötigt:

```
1  int selectedIndex = -1, openCounter = 0;
2  double timeoutRest = 0.0;
3  boolean ready = false;
4  boolean[] openList = new boolean[3],
5            semaphores = new boolean[3],
6            guards = new boolean[3],
7            channels = new boolean[3],
8            adaRendevous = new boolean[3];
```

Erklärung:

- Die Variable `openCounter` zählt alle offenen Alternativen und `selectedIndex` speichert die auszuführende Alternative.
- Die Größe der fünf Arrays richtet sich nach der Anzahl der Alternativen. Das `guards`-Array speichert beispielsweise für jede Alternative den Zustand des jeweiligen *Guard*'s. Besitzt eine Alternative keinen *Guard*, gilt sie - in Bezug auf den *Guard* - als offen.

  Im Anschluss an die Deklaration werden alle Arrays mit *false* initialisiert.

- Die *double*-Variable speichert die Restzeit, die bis zum *timeout* verbleibt.

10.3.3 Überprüfung der Alternativen

Nachdem alle nötigen Variablen angelegt wurden, wird getestet, welche Alternativen offen sind.

```
1  do{
2    // Guards
3    if (/*booleanExpression*/) guards[0] = true;
4    guards[1] = true;
5    guards[2] = true;
6    // Channels
7    if (channel1.hasWaitingPartner()) channels[0] = true;
8    for(int i=0; i<2; i++){
9      if (ch1[i].hasWaitingPartner()) channels[1] = true;
10   }
11   channels[2] = true;
12   // Ada-Style-Rendevous
13   adaRendevous[0] = true;
14   adaRendevous[1] = true;
15   if ( list[0].hasValidmember() )
16     adaRendevous[2] = true;
17   // Semaphores
18   if ( semaphore1.availablePermits() > 0 )
19     semaphores[0] = true;
20   semaphores[1] = true;
21   semaphores[2] = true;
```

Erklärung:

- Zu Beginn des Auswahlverfahrens wird eine *do-while*-Schleife betreten. Da es sich um ein *select*-Statement mit einem *timeout* handelt, muss in regelmäßigen Abständen überprüft werden, ob sich eine Alternative geöffnet hat (weshalb die Überprüfung der Alternativen mehrfach durchgeführt werden muss).
- Jede Alternative, die keinen *Guard* besitzt, gilt als offen. Weshalb in das `guards`-Array ein *true* eingetragen wird - in diesem Fall trifft das auf Alternative zwei und drei zu.

  Bei jeder Alternative mit einem *Guard* wird der *bool'sche* Ausdruck des *when*-Statements in ein *if*-Statement überführt, welcher an dieser Stelle ausgewertet wird. Das Ergebnis der Überpfüfung des *if*-Ausdrucks wird in das `guards`-Array eingetragen.
- Der zweite Test überprüft die Kanäle. In diesem Beispiel wird ein Kanal in Alternative eins und ein *replicate* in Alternative zwei verwendet. Der Kanal muss

belegt sein, deshalb wird mit der Methode `hasWaitingPartner` überprüft, ob bereits ein Kommunikationspartner auf dem Kanal wartet. Sollte das der Fall sein, wird die Alternative als offen markiert.

Bei der Verwendung einer *replicate*-Alternative wird das komplette Kanal-Array überprüft, auf das die Alternative zugreift. Dafür wird der Inhalt des *replicate*-Ausdrucks in eine *for*-Schleife kopiert, die alle Kanäle überprüft und das Ergebnis in das `channels`-Array an die entsprechende Stelle eingetragen. Dabei wird nur gespeichert, dass ein bliebiger Kanal des Arrays belegt ist. Welcher Kanal ausgewählt werden kann, wird später innerhalb der Alternative entschieden.

- Die Überprüfung eines *Ada-Style-Rendevous* oder eines Semaphors geschieht ähnlich wie der Test eines Kanals. Sowohl das Semaphor als auch die `ParameterList` des *Ada-Style-Rendevous* bieten Methoden an, die überprüfen, ob ein anderen Prozess einen Auftrag für ein Rendevous bzw. ob das Semaphor noch eine Ressource zur Verfügung hat.

  Wie beim *Guard* oder beim Kanal, wird auch bei diesen Tests ein *true* in das jeweilige Array geschrieben, wenn die Alternative kein Semaphor oder *Ada-Style-Rendevous* benutzt.

10.3.4 Zusammenfassen der Daten

Nachdem der Zustand eines *Guards*, *Channels*, *Semaphor* und *Ada-Style-Rendevous* für jede Alternative bekannt ist, müssen die Daten kombiniert werden, um festzustellen, ob eine Alternative offen ist.

```
1  for (int j=0; j<3; j++){
2    if ( guards[j] == false ||
3           semaphores[j] == false ||
4           channels[j] == false ||
5           adaRendevous[j] == false )
6        openList[j] = false; //alternative closed
7    else
8        openCounter++; //increment counter
9  }
```

Erklärung:

- Die *for*-Schleife überprüft die gesammelten Daten für jede Alternative.
- Sollte nur eine der vier Kriterien (für eine offene Alternative) nicht zutreffen, wird die entsprechende Alternative als geschlossen markiert.
- Gilt die Alternative als offen, wird der `openCounter` inkrementiert.

10.3.5 Auswählen einer Alternative

Nach der Überprüfung aller Alternativen, wird das Ergebnis ausgewertet.

```
1      if (openCounter != 0){
2        do{
3          decision = (int)(Math.random()*100) % 3;
4          ready = true;
5        }while (openList[decision] == false);
6      }else{
7        try{Thread.sleep(1);}catch(Exception e){}
8        if ( (timeoutRest -= 1) <= 0){
9          ready = true;
10         decision = Timeout;
11       }
12     }
13   }while(!ready);
```

Erklärung:

- Zunächst wird überprüft, ob die Anzahl der offenen Alternativen größer als Null ist. Ist das der Fall, wird zufällig eine offene Alternative gewählt.

  Der *bool'sche* Wert `ready` wird auf *true* gesetzt, was bedeutet, dass die Auswahl einer Alternative abgeschlossen ist.

- Wurde bei der Überprüfung keine offene Alternative gefunden, legt sich der Thread eine feste Zeitspanne schlafen. Im Anschluss wird diese Spanne von `timeoutRest` abgezogen und überprüft, ob die restliche Zeit kleiner gleich Null ist.

  Sollte der Wert `timeoutRest` gleich Null sein, wurde keine offene Alternative in dem vorgegebenen Zeitfenster gefunden und die *timeout*-Alternative wird ausgeführt. Wenn die Zeit nicht abgelaufen ist, wird die Überprüfung der Alternativen erneut durchgeführt.

Nach der Auswahl einer offenen Alternative, wird das *switch-case*-Statement betreten und die entsprechende Alternative ausgeführt.

10.3.6 Die Umwandlung einer replicate-Alternative

Bei der Analyse der offenen Alternativen wird für eine *replicate*-Alternative lediglich festgestellt, ob ein beliebiger Kanal einen Kommunikationspartner vorweisen kann. Sollte das Auswahlverfahren im Anschluss die *replicate*-Alternative auswählen, muss erneut festgestellt werden, welcher der Kanäle nun kommunizieren soll.
Diese zweite Auswahl wird direkt in der Alternative getroffen. Da bereits klar ist, dass mindestens ein Kanal bereit für eine Kommunikation ist, kann es auch nicht mehr zu einem *deadlock* kommen. Folgender Code wird in die Alternative eingefügt:

```
LinkedList tmpList = new LinkedList();
int i=0;
for ( ; i<2; i++){
  if( ch1[i].hasWaitingPartner() )
    tmpList.add(true);
  else
    tmpList.add(false);
}
do{
  i = (int)(Math.random()*100) % tmpList.size();
}while(tmpList.get(i) == false);

//begin content of replicate-Statement
ch1[i].send(i);
```

Erklärung:

- Zunächst wird eine Liste angelegt, die den Zustand jedes Kanals des Arrays speichert.
- Im zweiten Schritt wird der Inhalt des *replicate*-Statement in eine *for*-Schleife umgewandelt. Dabei wird der Initialisierungsteil des Schleifenzählers vor die Schleife gesetzt.
- Die *for*-Schleife durchläuft jeden Kanal des Arrays und speichert den Zustand des Kanals in der Liste.
- Anschließend wird zufällig ein sende- bzw. empfangsbereiter Kanal ausgewählt. Gespeichert wird der Wert im Schleifenzähler der *for*-Schleife. Die Deklaration des Zählers wurde aus der *for*-Schleife herausgelöst, damit die Variable innerhalb der gesamten Alternative gültig ist.
- Nachdem die Auswahl getroffen wurde, wird der eigentliche Code des *replicate*-Statements ausgeführt.

10.4 Zusammenfassung

Das *select* bietet eine Möglichkeit, einen nichtdeterministischen Kontrollfluss in Java zu implementieren. Das komplexe Verhalten des *selects* macht eine Implementierung recht aufwändig. Sämtliche Mechanismen, die dazu führen könnten, dass eine Alternative als geschlossen gilt, müssen vorher geprüft werden. Die Entscheidung, welche Alternative ausgeführt wird, hängt dabei einzig und allein von einem Zufallsgenerator ab.

Weitere Funktionalitäten von JavaFC

Neben den vorgestellten Konzepten und Statements der vorangegangenen Kapitel bietet JavaFC einige weitere Funktionen an.

Neue Statements

Pascal-FC bietet in seiner Syntax eine Endlosschleife an. Diese Schleife wird auch in JavaFC angeboten. Folgende Syntax besitzt die Schleife.

- RepeatForever := repeat Statement forever ;

Erklärung:

- Die Schleife beginnt mit dem Schlüsselwort *repeat*.
- Innerhalb der Schleife kann ein beliebiges Statement stehen, welches unendlich oft ausgeführt wird. Den Schluss des Statements bildet das Schlüsselwort *forever*, gefolgt von einem Semikolon.
- Sollte die Schleife ein *select*-Statement enthalten, welches eine *terminate*-Schlussalternative besitzt, wird die Schleife eventuell durch das *terminate* verlassen.

JavaFC-Bibliothek

JavaFC bietet die Möglichkeit, oft verwendete Methoden oder Funktionen in einer Bibliothek zu kapseln. Diese Bibliothek umfasst derzeit nur eine Funktion.

SimulateInterrupt

Die Funktion bietet die Möglichkeit einen Interrupt zu simulieren, was zu einem Prozesswechsel führt. So wird ein häufiges Wechseln des Kontrollflusses erreicht, was auf der einen Seite das Gesamtsystem etwas ausbremst, auf der anderen Seite jedoch ein hochgradig nebenläufiges Programm erlaubt.

```
1  public static final void simulateInterrupt(double val){
2    if(Math.random() < val){
3      try {
4        Thread.sleep(1);
5      } catch(InterruptedException e){}
6    }
7  }
```

Erklärung:

- Der Methode wird ein *double*-Wert übergeben, der bestimmt, mit welcher Wahrscheinlichkeit ein Prozesswechsel vollzogen wird. Der Bereich dieser Variable soll zwischen Null und Eins liegen.
- Eine Zufallszahl wird ermittelt und diese mit dem übergebenen *double*-Wert verglichen.
- Ist der Zufallswert kleiner als der Übergabeparameter wird ein Prozesswechsel vollzogen, andernfalls passiert nichts. Allerdings kann der Java-Thread-Scheduler dafür sorgen, dass ein Wechsel des Prozesses bereits kurz nach dem Ausführen der Methode durchgeführt wird.

11 Einschätzung und zukünftige Entwicklung

Bei der Entwicklung von JavaFC mussten einige Einschränkungen für die Verwendung gemacht werden. Diese Einschränkungen ergaben sich meist aus einer Sonderfallbetrachtung, welche oft Konsequenzen für den gesamten Präprozessor hatten.
Andere Einschränkungen resultieren aus fehlender Entwicklungszeit. So ist es möglich, dass die meisten Probleme durch zusätzliche Entwicklungszeit beseitigt werden könnten, wobei an dieser Stelle bereits mögliche Ansätze diskutiert werden.

Eingeschränkte Angabe über Fehlerquellen

Eine sehr störende Nebenerscheinung, für die momentan auch keine Lösung vorliegt, ist die Tatsache, dass beim Parsen des JavaFC-Quelltextes Fehler nicht wie üblich auf der Konsole - mit Zeilennummer und Fehlerursache - angezeigt werden.
Einfache Fehler in der Syntax werden oft mit dem Abbruch des Parsevorgangs und mit einer Ausgabe einer allgemeinen Fehlernachricht quittiert. Der Anwender sollte selbstständig auf korrekte Klammerung oder die richtige Syntax aller Statements achten. Der Präprozessor kann an dieser Stelle nicht behilflich sein.
Eine Hilfestellung beim Lokalisieren möglicher Fehlerquellen ist eine Ausgabe, die sämtliche neuen Konstrukte auf der Konsole ankündigt. Bricht das Parsen des Quelltextes an einer Stelle ab, sollte das letzte erfolgreich geparste Statement gesucht und eine Fehleranalyse durchgeführt werden. Folgende Konsolen-Ausgabe deutet auf einen syntaktischen Fehler im *select*-Statement innerhalb der `body`-Methode hin:

```
1  JavaFC Preprocessor: Reading from file Counter.javaFC...
2  ******************************************
3  All converting steps are following.
4
5  Converting: 1. process class...
6    converting: 1. entry declaration...
7   entering method body...
8    converting: 1. select statement...
9    converting: 1. accept statement...
10 ******************************************
11 JavaFC Preprocessor: Encountered errors during first parse.
```

Sollte ein Fehler während der zweiten Phase des Parse-Vorgangs auftreten und keine falsche Verwendung eines Statements oder neuen Datentyps vorliegen, ist dies weitaus dramatischer und nur durch eine manuelle Fehlerbehebung in den generierten Java-Files zu beheben. Ein solcher Fehler ist meist auf ein fehlerhaftes Arbeiten des Präprozessors zurückzuführen.
Ein manuelles Beseitigen des Fehlers durch den Nutzer ist ohne detailiertes Wissen um die interne Vorgänge und Bearbeitungsschritte des Präprozessors nicht zu bewerkstelligen. Ein solcher Fehler könnte folgendermaßen aussehen[1]:

```
1  JavaFC Preprocessor: Reading from file Counter.javaFC...
2  ******************************************
3  All converting steps are following.
4  
5  Converting: 1. process class...
6    converting: 1. entry declaration...
7   entering method body...
8    converting: 1. select statement...
9    converting: 1. accept statement...
10 ******************************************
11 JavaFC Preprocessor: First transformation complete.
12  Processing the 1. entry statement
13  Processing the 1. accept statement
14  Processing the 1. select statement
15  /*runtime error follows: mostly String operation failure*/
16 ******************************************
17 JavaFC Preprocessor: Encountered errors during second parse.
```

Einschränkungen beim Ada-Style-Rendevous

Im Kapitel *Ada-Style-Rendevous* (↗ siehe Kapitel 8.2.1) wird das Fehlen von überladenen *Entries* diskutiert. Stößt der Präprozessor beim Parsen des Quelltextes auf das Schlüsselwort *accept* vergleicht er den folgenden Methodennamen mit allen vorher eingelesenen *entry*-Deklarationen. Bei einer Erweiterung der Implementierung könnte JavaFC nun sämtliche Übergabeparameter vergleichen und so Überladung von *Entries* ermöglichen.

Einschränkungen der select-Anweisungen

Die meisten Einschränkungen in JavaFC finden sich im Zusammenhang mit dem *select*-Statement.
Bei der Analyse der *replicate*-Alternative wurde festgestellt (↗ siehe Kapitel 10.2.4), dass die Variable, die innerhalb der Alternative auf das Kanal-Array zugreift im Kopf

[1] Die hier auskommentierte Fehlernachricht ist oft eine fehlgeschlagene String-Operation, die sich auf vom Präprozessor generierte Dateien beziehen.

des *replicates* angelegt werden muss. Der Grund liegt in der internen Verarbeitung des *replicates*, eine Lösung für dieses Problem liegt jedoch nicht vor.
Die Kommunikation zweier Prozesse durch einen Kanal, ausgewählt durch ein *select*, kommt in der aktuellen Implementierung nicht zu Stande (↗ siehe Kapitel 10.2.5). Der Grund ist, dass jeweils beide auf das Betreten des Kanals durch den anderen Prozess warten. Dieses Problem könnte gelößt werden, indem der Kanal registriert, dass sowohl Sender, als auch Empfänger über ein *select* kommunizieren wollen.
Die Schlussalternative *terminate* besitzt ebenfalls eine wichtige Einschränkung. Zwei Prozesse, die sich in einem *select*-Statement über eine *terminate*-Alternative beenden wollen, werden nie beendet. Jeder Thread wartet für seine Beendigung auf das Ende des jeweils anderen Threads (↗ siehe Kapitel 10.2.6). Eine Lösung für das Problem ist die Einführung eines neuen Threadtypen, von dem jeder Prozess in JavaFC erbt. Der neue Prozesstyp müsste einen zusätzlichen Zustand des Thread speichern. Dieser Zustand besagt, dass der Prozess in einem *terminate* auf die Beendigung der anderen Prozesse wartet. Wenn alle normalen Prozesse beendet sind und nur noch Prozesse in diesem zusätzlichen Zustand existieren, können diese ebenfalls beendet werden.

12 Zusammenfassung

In dieser Arbeit wurde ein Präprozessor für Java eingeführt, der die syntaktischen Möglichkeit Javas erweitert und bekannte Konzepte der Nebenläufigkeit in den Sprachumfang intergriert.
Der Umfang der Arbeit orientiert sich dabei an der Lehrsprache Pascal-FC, welche auf Grund der geringer werdenen Bedeutung von Pascal-FC durch ein entsprechendes Java-äquivalent abgelößt werden soll.
Die Anpassungen umfassten neben der Einführung neuer Datentypen, neuer Anweisungen und Klassenmodifikatoren. Neben der Umsetzung klassischer Synchronisationskonzepten führt JavaFC auch spezielle Mechanismen anderer Programmiersprachen wie *Occam* oder *Ada95* ein.
Als klassisches Synchronisationsmittel kann das Semaphor, die kritische Region oder der Monitor bezeichnet werden. Als weiteren Datentyp führt JavaFC Kanäle ein, die sowohl Synchronisations-, als auch Kommunikationsmittel darstellen. Das *Ada-Style-Rendevous* und das *select*-Statement komplettieren das Angebot von nebenläufigen Konstrukten. Neben den aus Pascal-FC übernommenen Mechanismen und Datentypen ersetzt JavaFC aber auch Java-eigene Elemente durch neue Konstrukte. Als Beispiel kann hier die Einführung des neuen *process*-Schlüsselwortes für nebenläufige Kontrollflüsse oder das Starten dieser Prozesse mittels *cobegin*-Statement genannt werden.
Bei der Umsetzung von JavaFC wurde auf das Entwickeln eines neuen Compilers verzichtet, da das Werkzeug Java-Compiler-Compiler die Möglichkeit bietet, die Grammatik Javas zu erweitern und diese Erweiterungen, nach dem der Quelltext eingelesen (geparst) wurde, wieder auf regulären Java-Code abzubilden. Die Aufgabe des Präprozessors ist also die syntaktische Korrektheit des JavaFC-Quelltextes zu überprüfen und alle neuen Elemente durch vorhandene Java-Konstrukte zu ersetzen, um diesen im Anschluss mit Hilfe des regulären Java-Compilers in Byte-Code zu überführen.

A Zusammenfassung der JavaFC-Syntax

Im folgenden wird die Syntax aller neuen Statements nocheinmal zusammengefasst.

A.1 Allgemeine Statements

Endlosschleife:

- RepeatForever := repeat Statement forever ;

A.2 Process, Cobegin und Program

Das *process*-Statement:

- Process := process NAME { (ProcessBody)* }
- ProcessBody := BodyMethod | ...
- BodyMethod := public void body ( ) { (Statement)* }

Das *cobegin*-Statement:

- CobeginStatement := cobegin { (CobeginEntry | ForLoop)* }
- CobeginEntry := [NAME =] new NAME ([Expression]) ;
- ForLoop := for (ForInit ; ForCompare ; ForUpdate) (CobeginEntry)*

Das *program*-Statement:

- Program := program NAME { (ProgramBody)* }
- ProgramBody := VariableDeclaration | CobeginStatement | ...

A.3 Semaphor

- SemaphoreDeclaration := semaphore Semaphore (, Semaphore)* ;
- Semaphore := NAME [(Expression)]
- SemaphoreStatement := NAME . up ( ) | NAME . down ( )

A.4 Shared und Region

- SharedDeclaration := shared TYPE NAME (, TYPE NAME)* ;
- CriticalRegionStatement := region NAME { (Statement)* }
- ConditionalRegionStatement := region NAME Guard { (Statement)* }
- Guard := when (Expression)

Anmerkung:

- JavaFC überprüft, ob eine als *shared* deklarierte Variable, außerhalb eines *region*-Statement verwendet wird.
- Bei einer Verwendung einer *shared*-Variable außerhalb einer *region*, bricht der Präprozessor das Parsen mit einem Fehler ab
- Innerhalb eines *guards* kann jedoch auf die *shared*-Variable zugegriffen werden.
- Ein verschachtelter Aufruf eines *region*-Statements auf der selben *shared*-Variable führt zu einem Deadlock. Was durch eine Fehlermeldung durch den Präprozessor gemeldet wird.

A.5 Monitor und Condition

Die *monitor*-Klasse:

- Monitor := monitor NAME { (MonitorBody)* }
- MonitorBody := ExportMethod | CoditionDeclaration | ...
- ExportMethod := export ExportSignature { (Statement)* }
- ExportSignature := TYPE NAME ((Parameter)*)

Die *Condition*-Variable:

- ConditionDeclaration := condition NAME (, NAME)* ;
- ConditionStatement := NAME . delay ( ) | NAME . resume ( )
| NAME . size ( )

Anmerkung:

- Der Präprozessor überprüft, ob innerhalb des Monitors das Schlüsselwort *public* verwendet wird.
- Wird in einem Monitor das Schlüsselwort *public* verwendet, bricht der Präprozessor das Parsen des Quelltextes mit einer Fehlermeldung ab.

A.6 Entry und Accept

- EntryDeclaration := entry MethodSignature ;
- MethodSignature := TYPE NAME ((Parameter)*)
- AcceptStatement := accept MethodSignature { (Statement)* }

Anmerkung:

- Das Überladen von *Entries* ist nicht möglich.

A.7 Channel

- ChannelDeclaration := channel Modifier Channel (, Channel)* ;
- Channel := NAME [= new channel Modifier]
- Modifier := [ChannelDataType] [[ ]]
- ChannelDataType := (TYPE | sync | synchronous)
- ChannelStatement := NAME . send (Expression) | NAME . receive ( )

A.8 Select

- Select := select { SelectBody }
- SelectBody := SelectEntry (or SelectEntry)* [FinalAlternative]
- SelectEntry := [Guard =>] EntryBody [: Statement]
- Guard := when (Expression)
- EntryBody := SyncStatement | ReplicateEntry
- SyncStatement := SemaphoreStatement | ChannelStatement
  | AcceptStatement
- ReplicateEntry := replicate (Init ; [Check] ; [Update]) Statement

- FinalAlternative := (timeout INTEGER | else | terminate) Statement

Anmerkung:

- Eine Alternative gilt dann als offen, wenn sein *Guard* wahr ist.
- Haben verschiedene Alternativen einen offenen *Guard*, wird eine der offenen Alternative ausgewählt.
- Wenn eine Alternative betreten wurde, das Synchronisationsstatement jedoch nicht ausführbar ist, da die entsprechenden Resourcen fehlen (nicht empfangsbereiter Kanal oder ein fehlender Auftrag für ein Ada-Style-Rendevous), blockiert die Alternative.
- Existiert keine offene Alternative (alle *Guards* sind geschlossen) und wurde auch keine Schlussalternative formuliert, bricht das *select*-Statement mit einem Fehler ab.

Literaturverzeichnis

[1] G.L. Davies University of Bradford, UK
"Pascal-FC Version 5 Language Reference Manual".
www.lcc.uma.es/~gallardo/lrm.pdf

[2] G. Davies and A. Burns.
"The Teaching Language Pascal-FC".
The Computer Journal 33(2) (1990).

[3] G. Davies and A. Burns.
"Concurrent Programming".
Addision-Wesley Publishing Company (1993).

[4] E.W. Dijkstra.
"Co-operating sequential processes".
ed. F. Genuys, Academic Press (1968).

[5] C. A. R. Hoare.
"Towards a theory of parallel programming".
International Seminar on Operating System Techniques (1971).

[6] C.A.R. Hoare.
"Monitors: an Operating System Structuring Concept".
Communications of the ACM
Vol. 17 (1974), pp. 549-557.

[7] INMOS Limited.
"Occam Programming Manual".
Prentice Hall (1984).

[8] ANSI.
Reference Manual for the Ada Programming Language (1983).

[9] J. Gosling, B. Joy, G. Steele, G. Bracha.
"The Java Language Specification Third Edition".
Addision-Wesley (2005).

[10] Sun Microsystems
Java Compiler Compiler Reference Manual
https://javacc.dev.java.net/doc/docindex.html.

[11] Donald E. Knuth
"Backus Normal Form versus Backus Naur Form"
Communications of the ACM 7

[12] R. Sethi, M. S. Lam, A. V. Aho
"Compiler. Prinzipien, Techniken und Werkzeuge" (Das Drachenbuch)
PEARSON STUDIUM; 2. Auflage (2008).

[13] Andrew S. Tanenbaum
"Moderne Betriebssysteme"
PEARSON STUDIUM; 2. Auflage (2002).