Universität Bremen
Fachbereich 3
Studiengang Informatik

Diplomarbeit

Ein System zur Übersetzung von Java nach CSP mit dem Ziel der automatischen Analyse nebenläufiger Java Programme

Manfred Hein

1. Gutachter: Dr. Hui Shi
2. Gutachter: Prof. Dr. Jan Peleska

Oktober 1999

Danksagung

Ich möchte mich an dieser Stelle zunächst einmal für die Ermöglichung meines Studiums sowie für die kontinuierliche Unterstützung bei meinen Eltern bedanken.

Für die Betreuung und nützliche Ratschläge danke ich Dr. Shi Hui und Prof. Dr. Jan Peleska, sowie Dr. Berthold Hoffmann für die Durchsicht des Entwurfs.

Für die gute Betreuung während des Studiums geht ein besonderer Dank an Dr. habil. Zhenju Qian, Dr. Heino Gärtner sowie speziell an Dipl. Ing. Günter Feldmann für die Unmengen an Kaffee, die ich während unzähliger nächtlicher Programmiersessions in den Rechnerräumen des Fachbereichs zu mir nehmen durfte und die wertvollen Anregungen zur "Philosophie" der Systementwicklung.

Ein weiterer Dank geht an die Firmen ATB (Institut für angewandte Systemtechnik Bremen, GmbH) und Freiheit.com Technologieberatung GmbH, Hamburg, die mir die Möglichkeit gaben, die betriebseigene Infrastruktur für die Erstellung dieser Diplomarbeit zu nutzen.

Erläuterung der verwendeten Notation

Um eine leicht und verständlich lesbare Arbeit zu erreichen, wurden einige Konventionen im Bezug auf die textuelle Darstellung dieses Dokuments getroffen, die an dieser Stelle eingeführt werden.

Notation im Fließtext

- Zur besseren Übersicht werden Absatzüberschriften benutzt, die nicht im Inhaltsverzeichnis erscheinen und dazu dienen, die einzelnen Abschnitte eines Fließtextes logisch zu verknüpfen und deren Inhalt zu umreißen.
- Quelltext ist in `kleiner nicht proportionaler Schrift` gesetzt. Schlüsselwörter der jeweiligen Programmiersprachen sind zusätzlich **`fett`** hervorgehoben.
- Bezeichner, die sich direkt auf Quelltext oder Elemente einer Abbildung beziehen sind `nicht proportional` gesetzt.
- Zur Betonung von Worten oder Satzfragmenten sind diese *kursiv* gesetzt.
- Bildunterschriften sind zur besseren Unterscheidung vom Fließtext in *kursiven Lettern* gesetzt.

Notation kontextfreier Grammatiken

In dieser Arbeit werden diverse Sprachstrukturen dargestellt, die durch kontextfreie Grammatiken beschrieben werden.
Eine kontextfreie Grammatik besteht aus einer Menge von Produktionen. Jede Produktion besitzt ein abstraktes Symbol (Nicht- terminal) auf der linken Seite und einer Sequenz von einem oder mehreren nicht- terminalen und terminalen Symbolen auf der rechten Seite. Für jede Grammatik werden die einzelnen Terminalsymbole aus einem festgelegten Alphabet zusammengesetzt. Ausgehend von einem speziellen Nicht- terminal –dem Startsymbol– ergibt eine kontextfreie Grammatik eine Sprache, die als Menge aller aus dem Startsymbol hervorgehenden Ableitungen definiert wird. Eine Ableitung ist hierbei das wiederholte Ersetzen der nicht- terminalen Symbole durch die jeweilige rechte Seite der entsprechenden Produktion (vgl. [Naur60]).
Die Darstellung der kontextfreien Grammatiken erfolgt hierbei in der in [GoJoSt96] festgelegten Form.
Terminalsymbole sind in **`kleiner, fetter nicht proportionaler Schrift`** dargestellt. Nicht- terminale Symbole sind durch *eine kleine kursive Schrift* gekennzeichnet. Die Definition eines Nicht- Terminalen erfolgt durch den Namen des Symbols mit nachfolgendem Doppelpunkt (:). Eine oder mehrere alternative rechte Seiten werden in separaten aufeinanderfolgenden Zeilen dargestellt. Ein nicht- terminales Symbol auf der rechten Seite, welchem ein Stern (*) folgt, repräsentiert null oder mehr aneinandergereihte Terme, die das Nicht- terminal produzieren kann. Ein oder mehrere Symbole, welche in eckigen Klammern ([,]) eingeschlossen sind, repräsentieren null oder einen Term, den das nicht- Terminal produzieren kann.

Das Startsymbolwird innerhalb dieser Arbeit in einer Grammatik als erstes Symbol definiert.
Folgende Beispiele sollen hierbei zur Veranschaulichung dienen:

IfThenStatement: **if (** *Expression* **)** *Statement*	*BooleanLiteral:* **true** **false**
MethodDescriptor: **(** *[ParameterDescriptor]* **)** *ReturnDescriptor*	*BreakStatement* **break ;** **break** *Identifier* **;**

Abbildung 0-1: Notation verwendeter Grammatiken

Notation für Klassendiagramme

Die in dieser Arbeit verwendeten Diagramme zur Beschreibung von Klassen und deren Beziehungen entsprechen der grafischen Darstellung des objektorientierten Modellierungswerkzeuges Rational Rose im OMT- Format (vgl. [Rational98]). Hierbei wird aus Gründen der Übersichtlichkeit auf die Angabe von Attributen und Methoden in den Klassen verzichtet.
Einen Überblick über die verwendeten Beziehungen gibt folgende Abbildung:

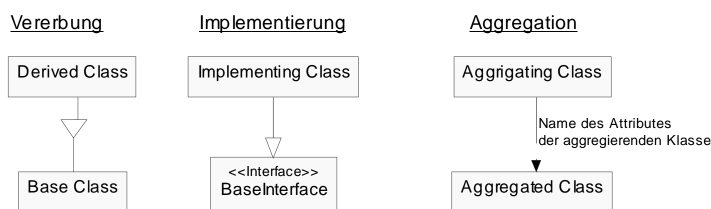

Abbildung 0-2: Notation verwendeter Klassendiagramme

Inhaltsverzeichnis

Kapitel 1: Einleitung ... 1
1.1 Motivation und Zielsetzung ... 1
1.2 Grundlegende Entwurfsentscheidungen ... 2
    1.2.1 Implementierungssprache Java ... 2
    1.2.2 Quellsprache und Quellcode ... 3
    1.2.3 Die Zielsprache CSP und das Auswertungswerkzeug FDR ... 4
1.3 Struktur dieser Arbeit ... 5

Kapitel 2: Grundlagen ... 7
2.1 Nebenläufigkeit ... 7
    2.1.1 Multitasking, Multiprocessing und Multithreading ... 7
    2.1.2 Einsatzmöglichkeiten für Nebenläufigkeit ... 9
    2.1.3 Konkurrierender Zugriff ... 11
    2.1.4 Synchronisation ... 12
    2.1.5 Lebendigkeitsausfälle ... 13
2.2 Nebenläufigkeit in Java ... 15
    2.2.1 Die Java Thread- API ... 16
    2.2.2 Synchronisation in Java ... 21
    2.2.3 Beispiele ... 22
2.3 CSP und FDR ... 26
    2.3.1 Sprachelemente von CSP ... 26
    2.3.2 CSP Refinement ... 30
    2.3.3 Der Model Checker FDR ... 31
2.4 Verifizierbares Verhalten von Threads ... 38
    2.4.1 Threadaufbau und -abbau ... 38
    2.4.2 Threadsuspendierung ... 39
    2.4.3 Monitorverhalten ... 39
    2.4.4 Synchronisationsverhalten ... 40
    2.4.5 Ausnahmen (Exceptions) ... 40
2.5 Auswertung paralleler Java Programme ... 40
    2.5.1 Ermittlung von Kontrollfäden ... 40
    2.5.2 Ermittlung benötigter Synchronisationsvariablen ... 41
    2.5.3 Ermittlung benötigter Strukturvariablen ... 42
    2.5.4 Ermittlung benötigter Algorithmen ... 42
2.6 Abstraktionsmodell ... 43
    2.6.1 Nachbildung des Java- Objektmodells ... 43
    2.6.2 Abbildung von Variablen auf Kanalprotokolle ... 44
    2.6.3 Abbildung von Java Methoden auf CSP Funktionen ... 45

Kapitel 3: Implementierung ... 49
3.1 Struktur des Systems ... 49
3.2 Java Class File Parser ... 50
3.2.1 Das Java Class File Format ... 50
3.2.2 Laufzeitstrukturen ... 56
3.2.3 Code- Segmente ... 57
3.2.4 Arbeitsweise des Bytecode Parsers ... 63
3.2.5 Erzeugung von Analysestrukturen ... 65
3.3 Arbeitsweise der Codesegmentanalyse ... 67
3.3.1 Methoden, spezielle Methoden und lokale Funktionen ... 67
3.3.2 Aufbau von Syntaxbäumen aus Codesegmenten ... 68
3.3.3 Syntaxbaumanalyse ... 77
3.4 Codegenerierung ... 78
3.4.1 Erweiterung des Syntaxbaumes um CSP Elemente ... 78
3.4.2 Abkürzen von Bezeichnern ... 79
3.4.3 Nachbildung arithmetischer Funktionen ... 80
3.4.4 Erzeugung einer textuellen Repräsentation ... 81
Kapitel 4: Schlußbetrachtung ... 87
4.1 Verwandte Arbeiten ... 87
4.2 Kritik & Ausblick ... 88
4.2.1 Fließkommawerte ... 88
4.2.2 Zeichenketten ... 88
4.2.3 Synchronisation mit Timeout ... 89
4.2.4 Rekursion ... 89
4.2.5 Schnittstelle zu Isabelle/HOL ... 90
4.2.6 Interprozeßkommunikation ... 90
4.2.7 Qualität erzeugter CSP/FDR Abstraktionen ... 90
4.3 Erfahrungen ... 92
Literaturverzeichnis ... 93
Anhang A: Benutzung des Übersetzers ... 97
Anhang B: Verwendete Bezeichner ... 101
Anhang C: Java2CSP Bibliothek ... 105
Anhang D: FDR Grammatik ... 111
Anhang E: Klassendiagramme ... 113
Anhang F: Beispiele ... 117

Abbildungsverzeichnis

0-1: Notation verwendeter Grammatiken ... ii
0-2: Notation verwendeter Klassendiagramme ... ii
2-1: Prozesse in einer Multprozeßumgebung ... 8
2-2: Prozesse und Threads in einer Multithreading Umgebung ... 9
2-3: Das Bankiersproblem ... 11
2-4: Beispiel eines Deadlocks ... 15
2-5: Abhängigkeiten der threadrelevanten Klassen ... 16
2-6: Threadklasse, die das Runnable Interface implementiert ... 22
2-7: Threadklasse, die java/lang/Thread erweitert ... 22
2-8: Beispiel für den Einsatz von suspend, resume und yield ... 23
2-9: Beispiel für den Einsatz des synchronized Schlüsselworts ... 24
2-10: Beispiel für den Einsatz von wait und notify zur Synchronisation ... 25
2-11: Verbindungsdiagramm der speisenden Philosophen ... 29
2-12: Boolsche Operationen in FDR ... 32
2-13: Arithmetische Operationen in FDR ... 32
2-14: Vergleichsoperationen in FDR ... 32
2-15: Strukturierungsausdrücke in FDR ... 34
2-16: Vordefinierte Prozesse in FDR ... 34
2-17: Kommunikation in FDR ... 34
2-18: Nebenläufigkeitsdefinition in FDR ... 35
2-19: Lokale Definitionen in FDR ... 35
2-20: Problem der speisenden Philosophen in FDR ... 36
2-21: FDR Session: Problem der speisenden Philosophen ... 37
2-22: Angestrebtes Start- und Stoppverhalten von Threads ... 38
2-23: Angestrebtes Suspendierungsverhalten von Threads ... 39
2-24: Angestrebtes Monitorverhalten ... 39
2-25: Angestrebtes Synchronisationsverhalten von Threads ... 40
2-26: Ermittlung von Threadinstanzen ... 41
2-27: Beispiel für Strukturvariablen ... 42
2-28: Beispiel für die Ermittlung benötigter Algorithmen ... 42
2-29: Beispiel für die Abbildung des Java Objektmodells ... 43
2-30: Verteilung von Objekt- IDs ... 44
3-1: Struktur des Gesamtsystems ... 49
3-2: Arten von Konstanten im Konstanten Pool ... 52
3-3: Modifizierer in Class Files ... 53
3-4: Grammatik für Feldsignaturen ... 54
3-5: Grammatik für Methodensignaturen ... 55
3-6: Nicht- Terminale für Signaturengrammatiken ... 55
3-7: Class- File- Attribute und deren Bedeutung ... 56
3-8: Limitierungen der Elemente einer Klassendatei ... 56

3-9: Arithmethische Befehle im Java Class File Format 58
3-10: Sprünge 59
3-11: Bedingte Verzweigungen 59
3-12: Operationen zum Legen von Kostanten auf den Stack 60
3-13: Operation zur Stackmanipulation 60
3-14: Operationen auf lokalen Variablen 61
3-15: Operationen auf Feldern 61
3-16: Beispiel für Array Klassen 61
3-17: Operationen auf Arrays 62
3-18: Ablauf eines Methodenaufrufs 62
3-19: Instruktionen zum Aufruf von Methoden 62
3-20: Arbeitsweise des Bytecode Parsers 64
3-21: Stapelparsierungsvorgang 64
3-22: Aktionen beim Parsieren einer Klasse 65
3-23: Aktionen beim Parsieren von Attributen 65
3-24: Bildung von Analysestrukturen 66
3-25: Elemente einer Methode in Analyseform 66
3-26: Elemente eines Feldes in Analyseform 66
3-27: Elemente einer Klasse in Analyseform 66
3-28: Fragment eines Codesegments als Syntaxbaum 69
3-29: Blockstruktur 70
3-30: Getypte und nicht- getypte Stacks 71
3-31: Klassenabhängigkeiten der abstrakten Syntax 72
3-32: Elemente der abstrakten Syntax 73
3-33: Abhängigkeiten der abstrakten Syntax als Grammatik 73
3-34: Syntaxbaum vor der Syntaxbaumoptimierung 76
3-35: Algorithmus zur Ermittlung von Kontrollfäden im Syntaxbaum 77
3-36: CSP Variablen in Zuweisungen 79
3-37: CSP Variablen bei dyadischen Operationen 79
3-38: CSP Variablen bei Methodenaufrufen 79
3-39: Algorithmus zur Kürzung von Bezeichnern 80
3-40: Qualifizierung von Bezeichnerklassen 80
3-41: Struktur der Codegenerierung 81
3-42: Strukturvariablentypen und zugeordnete Kanäle 82
3-43: Kanäle für Thread-, Monitor- und Strukurvariablentypen 83
A-1: Einstellbare Konstanten in erzeugten Spezifikationen 99
A-2: Das JAVA2CSP Frontend 99
B-1: Suffixlose Bezeichner 103
B-2: Suffixbehaftete Bezeichner 104
C-1: JAVA2CSP Bibliothek 109
D-1: FDR- Grammatik 112
E-1: Klassendiagramm Parser 113
E-2: Klassendiagramm Analyse und Codegenerierung 114
E-3: Aggregationsdiagramm der abstrakten Syntax 115
F-1: JAVA2CSP- Übersetzung des Leser-/Schreiber- Problems 120
F-2: Java Umsetzung eines unsicheren Schloßalgorithmusses 122
F-3: JAVA2CSP- Übersetzung des unsicheren Schloßalgorithmusses 126
F-4: Java Umsetzung eines sicheren Schloßalgorithmusses 127
F-5: JAVA2CSP- Übersetzung des sicheren Schloßalgorithmusses 133

Kapitel 1

Einleitung

In den letzten zwei Jahrzehnten ist der Trend zu erkennen, daß Software in immer größerem Maße bei sicherheitsrelevanten Anwendungen zum Einsatz kommt. Das angewendete Vorgehen zur Qualitätssicherung von Software, beschränkt sich zumeist auf die Begutachtung des Quellcodes und das Testen von erstellten Systemen. Dies ist bei der Entwicklung sensibler Softwaresysteme nicht unbedingt dazu geeignet, deren Sicherheit zu gewährleisten. Diese Diplomarbeit soll dazu beitragen, die formale Verifikation von Systemen zu vereinfachen und zu beschleunigen, um damit deren Zuverlässigkeit und Sicherheit zu fördern. Hierzu wird ein exemplarisches System entwickelt, mit dem sich die Nebenläufigkeitsproblematik von Programmen, die in einer weit verbreiteten Programmiersprache entwickelt wurden, überprüfen läßt.

1.1 Motivation und Zielsetzung

Der heutige Stand

Der Einsatz formaler Methoden zum Beweis der Korrektheit gegenüber einer gesetzten Spezifikation ist bis zum heutigen Tage nicht sehr verbreitet, und wird zumeist lediglich für explizit fehlertolerante Systeme eingesetzt.

Die Verbreitung von Rechnersystemen innerhalb von technischen Systemen hat in den letzten zwei Jahrzehnten stark zugenommen, so daß kaum eine technische Anlage ohne eine speziell für die entsprechende Aufgabe geschaffene Software funktioniert. Bisher wurden solche Systeme ausgiebig getestet, um sicherzustellen, daß sie der Aufgabe angemessen korrekt funktionieren. Es ist jedoch nicht davon auszugehen, daß alle möglichen Konfigurationen einer komplexen Software –besonders in Grenzbereichen der Anwendung– auf einwandfreie Funktion überprüft werden können.

Sicherheitsrelevante Systeme

In vielen Bereichen übernimmt Software sicherheitsrelevante Aufgaben. Dies ist zum Beispiel in Airbagsteuerungssystemen, Flugzeug- und Gleissteuerungsanlagen sowie Überwachungssystemen in Kraftwerken oder anderen industriellen Anlagen der Fall. Ein Fehler, der die korrekte Funktion einer solchen Software außer Kraft setzt, kann fatale Folgen haben.

Daß die Sicherheit solcher Systeme durch das bisherige Vorgehen bei den verschiedenen Softwareentwicklungs- und Testphasen nicht zu erreichen ist, zeigen immer wieder Ausfälle von Sicherungssystemen und Steuerungsanlagen, die eindeutig auf fehlerhafte Software zurückzuführen sind.

Besonders im Bereich der Massentransportmittel (vgl. [HaPe98]) sowie der Luft- und Raumfahrt (vgl. [BKPS97]) wird immer mehr dazu übergegangen Software zu verifizieren. Bisher wurden relevante Teile existierender Programme von Hand in Spezifikationen übertragen, und diese dann verifiziert. Hierbei ist jedoch nicht gewährleistet, daß die manuelle Übersetzung, trotz aller Sorgfalt, fehlerfrei ist. Außerdem steht immer noch ein

Compiler zwischen der entwickelten Software und der lauffähigen Binärversion, wobei fraglich bleibt, ob der Compiler wirklich jede Anweisung korrekt in Maschinensprache umsetzt.

Formale Verifikation in Industrieprojekten

Ein kompletter Verifikationsprozeß ist in der Regel sehr aufwendig und langwierig, so daß vor allem große industrielle Entwicklungen hiervon Abstand nehmen. Dies mag zum einen daran liegen, daß viele Entwickler die diesbezüglichen Möglichkeiten nicht kennen, da sie selbst im Vorgehensmodell (vgl. [BMI92]), dem Standard des Bundesministeriums des Innern der Bundesrepublik Deutschland für die Planung und Durchführung von informationstechnischen Vorhaben, nicht erwähnt werden, zum anderen daran, daß es bislang nur wenige Werkzeuge gibt, die die formale Verifikation von Softwaresystemen wirksam unterstützen.

Aufgabe und Ziel

Ziel dieser Diplomarbeit ist es, ein effektives System zu entwickeln, daß eine automatisierte Übersetzung einer Programmiersprache in eine Spezifikationssprache realisiert, mit der sich Synchronisationseigenschaften eines Systems formal korrekt nachweisen lassen. Die Untersuchung der Synchronisationseigenschaften beziehen sich dabei auf die Kommunikation mehrerer Kontrollfäden eines Programms.

Das Ergebnis dieser Arbeit soll dabei helfen, die Analyse bestehender Systeme zu vereinfachen und damit zu einer höheren Sicherheit und Zuverlässigkeit in Softwaresystemen beizutragen.

1.2 Grundlegende Entwurfsentscheidungen

Zu Beginn einer Softwareentwicklung, die ein gestecktes Ziel innerhalb einer festgelegten Zeit erreichen soll, ist es vor dem Beginn der Arbeit notwendig grundlegende Entwurfsentscheidungen zu treffen. In diesem Fall betrifft das die Implementierungssprache, den zu übersetzende Code und die Zielsprache in die übersetzt werden soll.

1.2.1 Implementierungssprache Java

Als Implementierungssprache für den Softwareanteil dieser Diplomarbeit wurde Java gewählt, da es sich hierbei um eine der modernsten Softwareentwicklungssprachen handelt. Sie weist für diese Arbeit gegenüber anderen Sprachen einige bestechende Vorteile auf, die im folgenden aufgeführt sind [vgl. [GoJoSt96]].

- **Verteiltheit**: Java wurde so entworfen, daß netzwerkbezogene Applikationen einfach und schnell entwickelt werden können. Es unterstützt verschiedene Kommunikationsprotokolle, die den netzweiten Datenaustausch ohne Umwege über externe Bibliotheken ermöglichen.
- **Sicherheit**: Der Begriff der Sicherheit ist eng verknüpft mit der Robustheit. Die Maßnahmen, die zur Robustheit von Java Programmen getroffen wurden, sind für die Sicherheit des Systems unabdingbar, da sie vor undefinierten Systemverhalten schützen. Da Java Bytecode internetweit verteilt wird, muß dieser so gestaltet sein, daß er innerhalb von Systemen keinen mutwillig initiierten Schaden anrichten kann. Aus diesem Grunde benutzt das Java Laufzeitsystem einen *Bytecode Verifizierer*, der überprüft, ob eingehender Bytecode festgelegten Restriktionen im Bezug auf Form und Inhalt genügt. Des weiteren fängt das System bei der Benutzung von Java Applets Zugriffe auf Verzeichnisse und Ressourcen ab, auf die es keinen Zugriff haben soll.
- **Multithreaded**: Die Programmiersprache Java unterstützt inhärent die Verwendung von Threads. Es existieren Konstrukte in der Sprache, die Synchronisationsmechanismen zwischen einzelnen Kontrollflüssen bereitstellen, und so die Entwicklung von Applikationen mit mehreren Threads vereinfachen. Die Java Klassenbibliothek bietet

außerdem mächtige Möglichkeiten zum Aufbau, Abbau und zur Steuerung von Threads.

1.2.2 Quellsprache und Quellcode

1.2.2.1 Quellsprache

Wie bereits erläutert, ist das Ziel dieser Arbeit, die Nebenläufigkeitskonstrukte einer Programmiersprache in eine Verifikationssprache zu übersetzen. Die Wahl der Ausgangssprache hängt dabei von den folgenden Faktoren ab:

- **Threadunterstützung**: Die Sprache, in denen die zu untersuchenden Programme geschrieben wurden, muß die Entwicklung von Programmen mit mehreren Kontrollfäden unterstützen, da dies das wichtigste Element zur Erzeugung nebenläufiger Systeme ist.
- **Festgelegte Threadfunktionalität**: Die Threadfunktionalität muß festgelegt sein. Dies impliziert, daß es eine einheitliche Programmierschnittstelle für Threadfunktionen geben muß. In der Regel wird dies über spezielle betriebssystemeigene Bibliotheken realisiert.
- **Akzeptanz**: Die betrachtete Sprache muß im wirtschaftlichen und akademischen Bereich akzeptiert sein und für die Erstellung komplexer Systeme benutzt werden. Es macht kaum Sinn, eine Programmiersprache zur Verifikation heranzuziehen, die nur im akademischen Bereich verwendet wird, da die sicherheitsrelevanten Aufgaben in der Regel in Industrieprojekten ihre Anwendung finden.

Threadbibliotheken

In der Regel werden alle Threadfunktionalitäten über das Betriebssystem zur Verfügung gestellt. Gewachsene Programmiersprachen wie C, die bei der Entwicklung nicht auf die Unterstützung von Threads angelegt waren, benutzen spezielle betriebssystemabhängige Bibliotheken zur Nachbildung mehrerer Kontrollfäden. Dies birgt den Nachteil, daß nicht festgelegt ist, welche Arten von Threads verwendet werden. So bieten beispielsweise Posix- Threads, Green Threads, Solaris Threads oder Windows Threads nicht die gleichen Programmierschnittstellen (vgl. [KSS95]). Durch die verschiedenen Implementierungen der jeweiligen Thread- API läßt sich somit keine verläßliche Aussage über deren verhalten machen.

Es ist somit sinnvoll, sich entweder auf eine Bibliothek zu spezialisieren, oder eine Schnittmenge aller Threadimplementierungen zu benutzen, die zu einer Programmierschnittstelle zusammengefaßt wurde.

Java als Quellsprache

Die einzige Programmierschnittstelle die dieses gewährleistet und innerhalb von Industrie und Forschung akzeptiert ist, ist die Java Thread- API. Sie stellt auf allen Betriebssystemen ein einheitliches Verhalten für Nebenläufigkeit durch Threads zur Verfügung.

Threads sind eine Basistechnologie im Java- Programmiermodell, was allein dadurch deutlich wird, daß die Basisklasse aller in Java erzeugter Klassen bereits Methoden zur Synchronisation mit anderen Objekten bietet, die innerhalb von Threads ausgeführt werden. Hieraus ergibt sich, daß Java Threads nicht nur als "Aufsatz" auf bestehende Konstrukte angewendet werden, sondern Bestandteil der Programmiersprache sind.

1.2.2.2 Quellcode

Da die Entscheidung für Java als Quellsprache bereits gefallen ist, stellt sich nun die Frage, in welcher Repräsentation Java Programme durch das hier zu entwickelnde Werkzeug verarbeitet werden sollen. Als Repräsentationen kommen der Java Quelltext oder die Java Bytecodedarstellung in Frage.

Für die Benutzung der Programmdarstellung in Java Bytecode (Java Class File Format) ergeben sich folgende Vorteile gegenüber der Verwendung von Java Quelltext.

- **Standardisiert**: Das *Java Class File Format* ist ein von Sun Microsystems festgelegter Standard (vgl. [LiYe97]). Es existieren keine Derivate dieses Formates und nur Sun darf Änderungen an diesem Standard durchführen. Durch dieses Monopol bleibt ein einheitliches Format gewahrt, auf das sich verläßlich aufbauen läßt. Im Vergleich dazu entstehen laufend neue Versionen von Java Übersetzern mit neuen Sprachkonstrukten und Eigenschaften, auf deren geänderte Syntax und Semantik man mit einer Änderung am Softwareanteil dieser Arbeit reagieren müßte.
- **Sprachenunabhängigkeit**: Das *Java Class File Format* läßt sich nicht nur für Applikationen benutzen, die direkt in Java geschrieben wurden. So existiert beispielsweise ein Java Class File Assembler ("Jasmin", vgl. [MeyDo97]), mit dem es direkt möglich ist, die Befehle einer Java Laufzeitumgebung anzusprechen. Des weiteren ist zu erwarten, daß Übersetzer für andere Sprachen entwickelt werden, die die Java Virtual Machine als Zielarchitektur haben. Als Beispiel dafür können bereits funktional erweiterte Java Compiler wie *BALI* (vgl. [BALI99])oder *GJ* (vgl. [BOSW98]) gelten.
- **Verifikation von Programmen, die lediglich als Bytecode vorliegen**: In der Regel werden Java Programme unter Zuhilfenahme von Bibliotheken erzeugt (z.B. Java Soft Swing, Symantec Klassenbibliotheken, etc.), von denen meist nur die Bytecodedarstellung und aus kopierschutzrechtlichen Gründen nicht der Quelltext vorliegt. Würde man nun Java Quelltext zur Verifikation heranziehen, könnten diese Systeme nicht überprüft werden.

Eine genauere Betrachtung des Java Bytecodes befindet sich im Abschnitt “Das Java Class File Format” auf Seite 50. Aus dieser Betrachtung leitet sich ab, daß die Darstellung auf einer sehr abstrakten Ebene stattfindet. Aus diesem Grund hat die Verwendung des Class File Formats auch einige Nachteile:

- **flache Darstellung**: Die Darstellung eines Programms im Bytecode besitzt eine äußerst flache (assemblerartige) Struktur. Im Gegensatz zu anderen Darstellungsformen (z.B. *Juice*, vgl. [FrKi96]) wird kein Syntaxbaum abgelegt, aus dem die Programmstruktur direkt hervorgeht. Man bedient sich in der Bytecodedarstellung vielmehr mehrerer Abbildungen in Form von Dateien, die jeweils ein Modul darstellen. Die Querbeziehungen werden gesondert abgelegt, so daß zur Generierung des gesamten Programms mehrere Teile zusammengebunden werden müssen.
- **Abbildung auf im Quellcode verwendete Bezeichner**: Bezeichner für Variablen spielen auf Bytecodeebene eine untergeordnete Rolle. Sie sind jedoch für eine genaue Analyse und für die Übersichtlichkeit des zu erzeugenden Zielcodes unabdingbar, da sich nur durch die Angabe von Variablenbezeichnern einfache Beziehungen zwischen Quell- und Zielsprache herstellen lassen. Die Bezeichner lokaler Variablen werden in speziellen Strukturen einer Klassendatei eingetragen und lassen sich in der Regel nur durch Java Bytecode Compiler erzeugen, indem man diese explizit dazu anweist. Diese Bezeichner müssen jedoch mit mehr oder minder großem Aufwand auf die vorkommenden Objekte im Bytecode abgebildet werden.

1.2.3 Die Zielsprache CSP und das Auswertungswerkzeug FDR

Als Zielsprache wurde CSP gewählt. Die Spezifikationssprache CSP (*Communicating Sequential Processes*), wurde von C.A.R. Hoare entwickelt (vgl. [Hoare85]) und bietet formale Methoden, mit denen sich Eigenschaften von Systemen mit parallel laufenden Prozessen verifizieren lassen.

Der große Vorteil von CSP ist, daß es Werkzeuge gibt, die CSP- Prozesse automatisch gegen eine Spezifikation prüfen können. Eines dieser Werkzeuge ist FDR (Failure Divergence Refinement, vgl. [Formal97]). Dieses Werkzeug baut auf den Grundlagen für Nebenläufigkeitskonstrukte von CSP auf.

1.3 Struktur dieser Arbeit

Diese Arbeit teilt sich neben dieser Einleitung in 3 Kapitel auf. Das folgende führt in die Problematik der Nebenläufigkeit und die Nachbildung von Nebenläufigkeitsproblemen mittels CSP und FDR ein und betrachtet anschließend, welche Eigenschaften von Threads in Java Probleme in diesem Zusammenhang aufwerfen.

Der zweite Teil ist der Implementierung des Übersetzungssystems gewidmet. Es beschreibt die Struktur des Systems und geht auf die Umsetzung der einzelnen Phasen des Übersetzungsprozesses (Parsierung, Analyse, Transformation und Codegenerierung) detailliert ein.

Eine kritische Betrachtung des Geleisteten sowie Vergleiche zu anderen Entwicklungen auf diesem Gebiet befinden sich abschließend im dritten Kapitel dieser Arbeit.

Kapitel 2

Grundlagen

Dieser Abschnitt behandelt die Grundlagen des Arbeitsvorhabens. Er führt in die Probleme der praktischen Anwendung von Nebenläufigkeit ein und beschreibt, wie sie formal spezifizierbar und damit überprüfbar sind.

Zunächst werden die Paradigmen der Nebenläufigkeit, die daraus resultierenden Synchronisationsprobleme und die zugehörigen Lösungsansätze vorgestellt. Der darauf folgende Abschnitt beschreibt die Möglichkeiten, die die Programmiersprache Java im Bezug auf Multithreading bietet, und welche Synchronisationsmöglichkeiten und -probleme es in Java- Programmen geben kann. Anschließend wird die Spezifikationssprache CSP für kommunizierende sequentielle Prozesse und das Modelchecking- Werkzeug FDR behandelt, mit dem CSP- Spezifikationen ausgewertet werden können.

2.1 Nebenläufigkeit

Nebenläufigkeit eröffnet Möglichkeiten, die in sequentiellen Programmen undenkbar sind. Es ist möglich, unabhängig voneinander laufende Aktivitäten zu definieren, die nicht nur Funktionen ausführen und dann auf deren Beendigung warten, sondern die Wartezeiten zur Bearbeitung anderer Aktivitäten nutzen.

Als Beispiel hierfür mag ein Vorgang in einem Programm dienen, das Daten von einem Bandgerät liest. Zunächst gibt das Programm den Befehl zum Vorspulen des Bandes an eine definierte Stelle und wartet, bis diese erreicht wurde. Daran anschließend wird das Bandgerät aufgefordert, eine gewisse Anzahl von Blöcken zu lesen. Bis dies passiert ist, muß das Programm wiederum warten.

Würde in diesem Fall Nebenläufigkeit zum Einsatz kommen, wobei mehrere Aktivitäten parallel auf dem Rechner ablaufen würden, könnten die Wartezeiten für andere Berechnungen genutzt werden.

Dieser Abschnitt beschreibt zunächst die verschiedenen Möglichkeiten, in denen Nebenläufigkeit zum Einsatz kommen kann und beschreibt dann die daraus entstehenden Probleme und Lösungsansätze.

2.1.1 Multitasking, Multiprocessing und Multithreading

Multitasking, Multiprocessing und Multithreading sind wichtige Eigenschaften von modernen Betriebssystemen, ohne die die Entwicklung anspruchsvoller Software für Mikrocomputer nach heutigen Maßstäben nicht mehr denkbar ist.

2.1.1.1 Multitasking

Multitasking bezeichnet die Fähigkeit eines Systems, mehrere Aufgaben parallel auszuführen. Es gibt innerhalb eines Betriebssystems in der Regel zwei Methoden, mit denen sich dieses Ziel erreichen läßt. Die am meisten genutzte und elementarste Vorgehensweise ist das Multiprocessing, bei dem sich mehrere in Ausführung befindliche Programme

einen oder mehrere Prozessoren teilen. Das zweite Einsatzgebiet ist die parallele Ausführung mehrerer Aufgaben innerhalb einzelner Prozesse. Diese Vorgehensweise wird Multithreading genannt.

2.1.1.2 Multiprocessing

Multiprocessing dient dazu, mehrere Programme gleichzeitig auf einem oder mehreren Prozessoren unter Verwendung des Prozeßmodells auszuführen (vgl. [Tan95], S.36ff).

Im Prozeßmodell werden alle auf einem Computer ausgeführten Programme in eine Anzahl von (sequentiellen) Prozessen bzw. Tasks aufgeteilt. Als Prozeß bezeichnet man ein sich in Ausführung befindliches Programm, inklusive der aktuellen Werte des Programmzählers, der Register und der Variablen.

Damit unterschiedliche Prozesse im Mehrprogrammbetrieb quasi parallel auf einer CPU ausgeführt werden können, bringt das Betriebssystem die verschiedenen Prozesse aufeinanderfolgend auf dem Prozessor zur Ausführung. Wenn der Prozessor zwischen mehreren Prozessen gewechselt wird, ist die Ausführungsgeschwindigkeit der Berechnungen eines einzelnen Prozesses möglicherweise nicht gleichmäßig. Dies hat zur Folge, daß die Ausführung eines deterministischen Programms als Prozeß im Bezug auf sein Zeitverhalten als nicht deterministisch anzusehen ist.

Ein Prozeß hat folgende Eigenschaften:

- Der Prozeß beginnt seine Ausführung an einem festgelegten Punkt im Code.
- Die Ausführung von Befehlen innerhalb des Prozesses erfolgt in einer festgelegten Sequenz. Die Ausführung ist somit deterministisch.
- Während der Ausführung hat ein Prozeß Zugriff auf seinen lokalen Speicher und gegebenenfalls auf einen Speicherbereich, den er sich mit anderen Prozessen teilt (Shared Memory).

Die folgende Abbildung zeigt drei Prozesse, die gleichzeitig auf einem Betriebssystem ausgeführt werden, das Multiprocessing unterstützt.

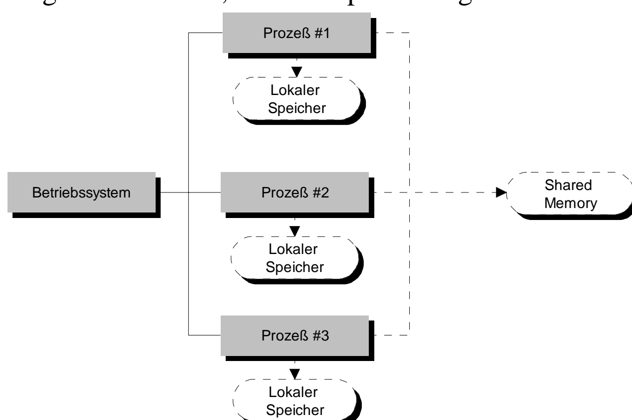

Abbildung 2-1: Prozesse in einer Multprozeßumgebung

2.1.1.3 Multithreading

Das Multithreading Paradigma ist dem Multiprocessing sehr ähnlich, mit dem Unterschied, daß Multithreading innerhalb eines Prozesses abläuft.

Ein *Thread of Control* ist ein Kontrollfaden, der zu genau einem Prozeß gehört und seine eigenen lokalen Variablen und einen eigenen Operandenstack besitzt. Threads haben jedoch zugleich Zugriff auf den globalen Speicher des Prozesses.

Ein Prozeß, der mehrere Threads enthält hat folgende Eigenschaften:

- Jeder Thread beginnt seine Ausführung an einem festgelegten Punkt im Code.
- Die Ausführung von Befehlen innerhalb eines *Threads* erfolgt in einer festgelegten Sequenz. Die Ausführung ist somit deterministisch.

- Jeder Thread führt seinen Code unabhängig von den anderen Threads innerhalb des Prozesses aus. Es gibt jedoch die Möglichkeit, daß Threads miteinander kooperieren.
- Threads haben einen gewissen Grad an Parallelität, der von verschiedenen Faktoren, wie z.B. der relativen Wichtigkeit verschiedener Threads innerhalb eines Prozesses und der Unterstützung verschiedener Threading- Funktionen durch das Betriebssystem, abhängt.
- Threads haben Zugriff auf verschiedene Datenstrukturen innerhalb eines Prozesses. Jeder Thread läuft separat, so daß alle Threads nur auf ihre eigenen lokale Variablen zugreifen können. Wenn zwei Threads eine bestimmte Operation benutzen, bekommen beide ihre eigene Kopie der Variablen, die innerhalb der Operation verwendet wird.

Das Multithreading wird durch folgende Abbildung illustriert, die drei Prozesse zeigt, in dem einer ein multithreaded- Anwendung ist.

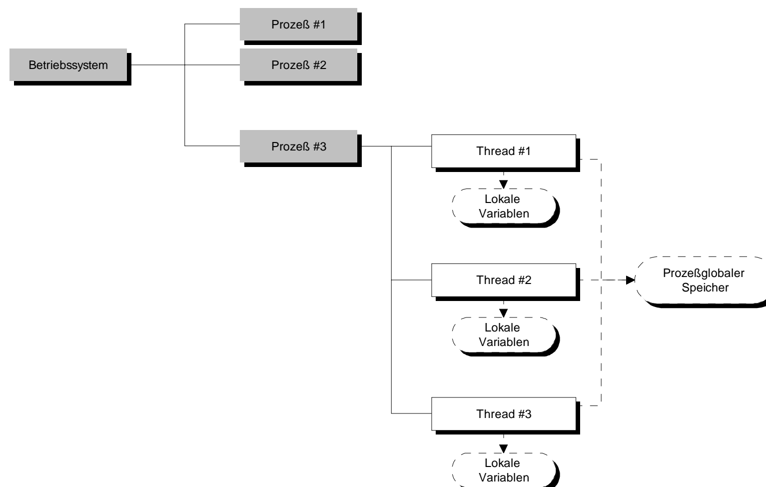

Abbildung 2-2: Prozesse und Threads in einer Multithreading Umgebung

2.1.2 Einsatzmöglichkeiten für Nebenläufigkeit

In der Literatur wird des öfteren darauf hingewiesen, daß der Einsatz des Nebenläufigkeitsprinzips kein Allheilmittel für alle Programmierprobleme ist. Im Folgenden sollen die Vor- und Nachteile abgewogen werden.
Doug Lea beschreibt in [Lea97] die Anwendungsmöglichkeiten für Multithreading, die sich bis auf einige Ausnahmen auch auf Multiprocessing anwenden lassen. Deshalb beschreibt diese Darstellung Nebenläufigkeit in seiner allgemeinen Form.
Für den Einsatz der Nebenläufigkeit sprechen folgende Argumente:

- **Implementierung reaktiver Systeme:** Einige Programme müssen mehr als eine Aktion gleichzeitig ausführen, wobei jede Aktion eine Reaktion auf eine Eingabe darstellt. Es ist zwar möglich, solche Systeme durch manuelles Ineinanderverzahnen der Aktivitäten in einem einzigen Prozeß zu programmieren, doch ist dieses Verfahren kompliziert und fehleranfällig. Programme für reaktive Systeme lassen sich mit Threads einfacher entwerfen und implementieren.
- **Verfügbarkeit**: Nebenläufigkeit ermöglicht es, Dienste jederzeit verfügbar zu halten. Die üblichen nebenläufigen Entwurfsmuster besitzen ein Objekt als Gateway Schnittstelle zu einem Dienst und erzeugen zu jeder Anfrage einen neuen Thread, um die zugehörigen Aktionen asynchron durchzuführen. Der Gateway ist in der Lage, schnell eine neue Anfrage anzunehmen. Hierdurch werden Engpässe vermieden und das Kommunikationsnetz von unerledigten Nachrichten befreit. Hierdurch wird ebenfalls der Zugriff gerechter gestaltet, da neue, schnell zu beantwortende Anfragen nicht mehr auf die Erledigung alter, zeitaufwendiger Anfragen warten müssen.

- **Steuerbarkeit**: Innerhalb von Threads können Aktivitäten unterbrochen, wieder aufgenommen und durch andere Objekte gestoppt werden. Diese Flexibilität und Einfachheit kann die sequentielle Programmierung nicht bieten, bei der die Unterbrechung oder Beendigung einer Aktivität zugunsten einer anderen oft schwierig zu implementieren ist.
- **Aktive Objekte**: Softwareobjekte modellieren nicht selten reale Objekte, die zumeist ein unabhängiges, autonomes Verhalten zeigen. Zumindest in manchen Fällen läßt sich dies am einfachsten durch Erzeugung eines neuen Tasks für jedes neu erstellte Objekt dieser Art programmieren.
- **Parallelität**: Auf Rechnern, die mit mehreren Prozessoren ausgestattet sind, kann durch nebenläufige Programmierung die vorhandene Rechenleistung besser ausgenutzt und die Leistung dadurch gesteigert werden. Sogar in einer Einprozessoranlage kann das Verflechten von Aktivitäten in Threads Verzögerungen wegen zeitaufwendiger Berechnungen vermeiden, die vor Beginn neuer Aktivitäten nicht abgeschlossen sein müssen.

Obwohl die Vorteile des Einsatzes von Nebenläufigkeit auf der Hand liegen, gibt es durchaus Bereiche, in denen der Sinn für den Einsatz zumindest hinterfragt werden muß. Dies liegt darin begründet, daß die Vorteile gegen die Nachteile des Nebenläufigkeitseinsatzes, wie z.B. hoher Ressourceneinsatz, Effizienzeinbußen und Komplexität der Programme abgewogen werden müssen. Diese Abwägung läßt sich in die Gebiete *operative Eigenschaften* und *Verwaltungseigenschaften* unterteilen.

Operative Eigenschaften

- **Sicherheit (*safety*)**: Sind Prozesse nicht völlig unabhängig voneinander, so kann es dazu kommen, daß Objekte aus ihnen Nachrichten an andere Objekte senden, die auch in anderen Tasks eine Rolle spielen. Alle diese Objekte müssen Synchronisationsmechanismen zum strukturellen Ausschluß (durch Vermeidung von gemeinsam benutzten Variablen) einsetzen, um konsistente Zustände sicherzustellen. Der Versuch, mehrere Tasks mit Objekten einzusetzen, die für den ausschließlichen Einsatz in sequentiellen Umgebungen konstruiert wurden, kann willkürlich erscheinende und schwer zu behebende Inkonsistenzen hervorrufen. Synchronisationsmechanismen können andererseits Programme komplizieren.
- **Lebendigkeit (*liveness*)**: Manchmal sind Aktivitäten innerhalb von nebenläufigen Programmen nicht lebendig. Das heißt, daß eine oder mehrere Aktivitäten aus verschiedenen Gründen einfach aufhören können, z.B. weil der Prozessor immer von anderen Aktivitäten beansprucht wird oder weil zwei unterschiedliche Aktivitäten verklemmt sind (siehe 'Verklemmung' auf Seite 14), was bedeuten kann, daß jede bis in alle Ewigkeit auf die Fortsetzung der anderen wartet.
- **Nichtdeterminiertheit**: Aktivitäten in mehreren Tasks können willkürlich miteinander verflochten sein. Die wiederholte Ausführung desselben Systems braucht nicht den identischen Verlauf zu haben. Rechenaufwendige Aktivitäten können vor solchen Aktivitäten enden, die praktisch keinen Rechenaufwand verursachen. Dies erschwert die Vorhersagbarkeit, Transparenz und Fehlerbehebung bei Systemen mit mehreren Tasks.

Verwaltungseigenschaften

- **Tasks oder Methodenaufrufe**: Tasks eignen sich nicht sonderlich für den Anfragen/Antworten Programmierstil. Muß ein Objekt aus logischen Gründen die Nachricht eines anderen Objekts abwarten, bevor es fortfährt, sollte für die Implementierung der gesamten Anfrage- Ausführung- Antwort- Sequenz ein und die selbe Task verwendet werden. Die Erzeugung eines neues Kontrollflusses bietet keine Vorteile, wenn es kein Möglichkeit für Nebenläufigkeit gibt. Umgekehrt kann eine als Task ausgeführte Aktivität nicht in der standardisierten sequentiellen Form aufgerufen werden, bei

der ein Client Argumente sendet, auf deren Verarbeitung wartet und als Antwort Ergebnisse erhält.

- **Aufwand bei der Taskerzeugung**: Eine neue Task zu erzeugen und zu starten ist speicherintensiver und dauert länger als eine Methode aufzurufen. Umfaßt eine Aktivität nur einige einfache Anweisungen, geht es wesentlich schneller, sie einfach durch einen Methodenaufruf einzuleiten, als eine eigene Task zu verwenden.
- **Aufwand bei der Kontextumschaltung**: Existieren mehr aktive Tasks als Prozessoren auf einem Rechner, schaltet das Betriebssystem von Zeit zu Zeit zwischen der einen Aktivität auf eine andere um. Dies erfordert eine Planung der Ausführungsreihenfolge, wobei die nächste auszuführende Task ausgewählt wird.
- **Synchronisationsaufwand**: Tasks, in denen Synchronisationsmaßnahmen ergriffen wurden, sind in der Regel langsamer als Tasks in denen das nicht der Fall ist. Funktionen, die Aktionen verschieben und je nach dem aktuellen Zustand der Objekte wiederaufnehmen müssen, brachen manchmal noch mehr Zeit. Verfügt das System über weniger Prozessoren als Tasks, kann es wegen des Task- und Synchronisationsaufwands dazu kommen, daß nebenläufige Systeme langsamer als sequentielle Programme laufen.
- **Abwägung zwischen Threads oder Prozessen**: Aktivitäten, die im wesentlichen unabhängig und einigermaßen umfangreich sind, lassen sich gegebenenfalls einfacher in Standalone- Programme einkapseln. Der Zugriff auf eigenständige Programme kann mittels Ausführungsmechanismen auf Betriebssystemebene oder durch entfernte Aufrufe besser erfolgen als auf mit mehreren Threads versehenen Komponenten eines einzigen Prozesses.

2.1.3 Konkurrierender Zugriff

Sowohl Multiprocessing, als auch Mutlithreading haben hinsichtlich der Kommunikation einzelner Tasks untereinander die gleichen Eigenschaften im Bezug auf den Zugriff auf gemeinsam genutzte Ressourcen.

In der Regel haben mehrere Tasks die Möglichkeit konkurrierend auf gemeinsam genutzte Ressourcen zugreifen. Als Ressourcen können z.B. gemeinsam genutzter Hauptspeicher, gemeinsam genutzte Dateien, etc. gelten.

Zeitkritische Abläufe, in der Literatur häufig als *race conditions* bezeichnet, beschreiben den Zustand, der eintritt, wenn mehrere Prozesse zu jeder Zeit auf eine gemeinsame Ressource zugreifen können.

Ein oft zitiertes Beispiel hierfür, ist das sogenannte Bankiersproblem. Es beschreibt zwei Tasks, die parallel auf ein fiktives Bankkonto zugreifen und eine Abbuchung durchführen. Jeder Task speichert den aktuellen Wert des Kontos in einer lokalen Variable, vermindert ihn um eine gewisse Summe und schreibt ihn zurück. In einem ungünstigen Fall, kann nun die gesamte Buchung des zweiten Tasks zwischen dem Auslese- und Schreibvorgang des ersten Tasks liegen, was zwangsläufig zu einem nicht korrekten Wert auf dem Konto führt.

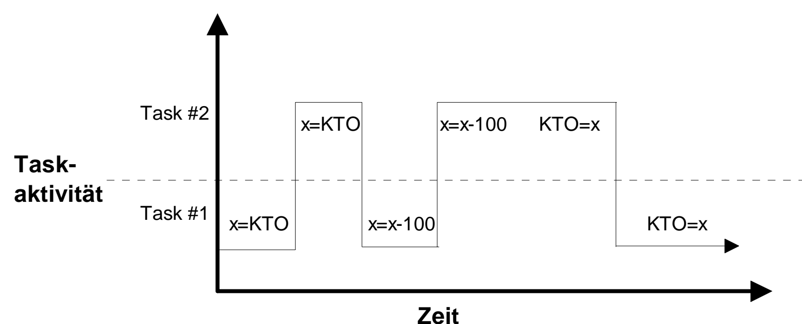

Abbildung 2-3: Das Bankiersproblem

Dieses Beispiel zeigt, wie wichtig es ist zu verhindern, daß mehrere Tasks zu einem Zeitpunkt auf eine Ressource lesend bzw. schreibend zugreifen können.

Das Problem der Vermeidung von zeitkritischen Abläufen kann auch in einer abstrakten Weise formuliert werden. Immerhin führen Tasks in der Regel die meiste Zeit lokale Berechnungen durch und greifen nur selten auf gemeinsam genutzt Ressourcen zu, wobei dann zeitkritische Abläufe stattfinden können.

Der Teil eines Programms, dessen Ausführung an einen zeitkritischen Ablauf partizipieren kann, nennt man kritischen Bereich.

Sucht man nun eine Lösung zur Vermeidung zeitkritischer Abläufe, muß diese den folgenden Bedingungen genügen (vgl. [Tan95], S.44ff).

- Maximal ein Prozeß darf sich zu jedem Zeitpunkt in seinem kritischen Abschnitt befinden.
- Ausführungsgeschwindigkeit und Anzahl der Prozessoren spielen keine Rolle.
- Kein Prozeß darf andere Prozesse blockieren, wenn er sich nicht in einem kritischen Abschnitt befindet.
- Kein Prozeß muß unendlich lange warten, bis er in seinen kritischen Abschnitt eintreten darf.

2.1.4 Synchronisation

In diesem Abschnitt werden einige korrekte Ansätze zur Durchsetzung des wechselseitigen Ausschlusses und damit der Synchronisation von Tasks untereinander vorgestellt(vgl. [Tan95]). Die aufgeführten Ansätze *Semaphor* und *Monitor* werden hierbei ausführlich beleuchtet, da sie im zu untersuchenden Java Bytecode Modell eine wichtige Rolle spielen.

Sperren von Unterbrechungen (Interrupts)

Eine einfache Vorgehensweise, um den gegenseitigen Ausschluß zu gewährleisten, ist das Sperren von Unterbrechungen während sich eine Task in ihrem kritischen Abschnitt befindet, so daß er während der kritischen Phase durch keinen anderen Task unterbrochen werden kann.

Striktes Alternieren mit Sperrvariablen

Ein weiterer Ansatz ist das strikte Alternieren mit Sperrvariablen, wobei die einzelnen Tasks eine Variable benutzen, um sich zu synchronisieren. Jede Task verfügt über eine ID und wartet innerhalb einer Schleife solange, bis der Wert der Variable ihre ID angenommen hat. Ist dies der Fall, kann sie ihren kritischen Bereich abarbeiten und anschließend die Sperrvariable auf die ID des nächsten Tasks setzen.

Test Set Lock Instruction

Der Umstand, daß beim strikten Alternieren die Prozesse lediglich der Reihe nach abgearbeitet werden, läßt sich mit der Test Set Lock Instruktion (TSL) beheben, die sich dadurch auszeichnet, daß sie ununterbrechbar eine Variable ausliest, deren Wert speichert und sie durch einen Wert der ungleich von Null ist ersetzt. Vom gespeicherten Wert wird abhängig gemacht, ob der kritische Abschnitt betreten wird. Konnte der kritische Bereich abgearbeitet werden, muß die Task die durch die TSL Instruktion benutzte Sperrvariable wieder neutralisieren.

SLEEP und WAKEUP (Schlafen und Aufwecken)

Die bisherigen Lösungen zur Durchsetzung des wechselseitigen Ausschlusses stützen sich allesamt auf die Verwendung von Warteschleifen und dem damit verbundenen Nachteil des Verbrauchs von Prozessorzeit, was zur einer Verlangsamung des gesamten Systems führt. Eine Lösung dieses Problems stellen die einfachen nicht blockierenden Primitive SLEEP und WAKEUP dar, die vom Betriebssystem zur Verfügung gestellt werden. Ruft eine Task SLEEP auf, wird er suspendiert und in die *Sleep- Queue* des Betriebssystems eingereiht, bis sie durch eine andere Task mittels WAKEUP wieder aufgeweckt wird. Der

Befehl WAKEUP bekommt in der Regel die aufzuweckende Task als Parameter übergeben, womit mehr als zwei Prozesse diese Primitive zum wechselseitigen Ausschluß benutzen können.

Semaphore

Ein Ansatz, der SLEEP und WAKEUP aufgreift, ist das Semaphor, daß auf E. W. Dijksta zurückgeht (vgl. [Dijk65]). Sein Ansatz benutzt eine Integer- Variable, um die Wecksignale für einen späteren Gebrauch zu zählen. Der spezielle Variablentyp Semaphor kann mit dem Wert Null belegt sein, der anzeigt, daß kein Wecksignal mehr berücksichtigt werden muß, oder anderenfalls einen positiven Wert haben, der anzeigt, wieviele Wecksignale noch zu berücksichtigen sind. Des weiteren existieren die zwei Operationen DOWN und UP. DOWN überprüft, ob der semaphoreneigene Integerwert größer als Null ist. Ist dies der Fall, wird der Wert dekrementiert und die aufrufende Task kann in den kritischen Abschnitt eintreten. Für den Fall, daß der Wert Null sein sollte, muß sich die aufrufende Task mittels SLEEP schlafen legen.

Die UP Operation inkrementiert den Wert des adressierten Semaphors. Falls sich Tasks bezüglich eines Semaphors schlafen gelegt haben, weil sie eine frühere DOWN Operation nicht beenden konnten, wird einer von ihnen zufällig ausgewählt und mittels WAKEUP wieder zur Ausführung gebracht.

Das Überprüfen und Verändern des Zähler sowie das Schlafenlegen bzw. Wecken eines Tasks wird in einer einzigen, unteilbaren, atomaren Aktion durchgeführt. Es ist somit sichergestellt, daß keine andere Task auf das Semaphor zugreifen kann, wenn eine Semaphoroperation erst einmal angestoßen wurde.

Monitore

Ein auf einer höheren Ebene stehendes Synchronisationsprimitiv ist der Monitor, der von C.A.R Hoare (vgl. [Hoare74]) und P. Brinch Hansen (vgl. [Brinch75]) beschrieben wurde. Ein Monitor besteht aus einer Menge von Prozeduren, Variablen und Datenstrukturen, die in einer besonderen Art von Modul bzw. Paket zusammengefaßt sind. Prozesse können Prozeduren auf einem Monitor aufrufen, wann immer sie es möchten, aber sie können nicht über Prozeduren, die außerhalb des Monitors deklariert sind, auf interne Datenstrukturen des Monitors zugreifen.

Nur ein Prozeß kann zu jedem Zeitpunkt in einem Monitor aktiv sein. Monitore sind üblicherweise Konstrukte einer Programmiersprache, die durch einen entsprechenden Compiler anders behandelt werden können als normale Unterprogrammaufrufe. Wenn ein Prozeß eine Monitorprozedur aufruft, wird zunächst überprüft, ob ein anderer Prozeß bereits im Monitor aktiv ist. Ist dies der Fall wird der aufrufende Prozeß suspendiert, bis der andere Prozeß den Monitor wieder verlassen hat. Andernfalls kann der Prozeß in den Monitor eintreten.

Es ist die Aufgabe des Compilers, den wechselseitigen Ausschluß der Monitoreinträge durchzusetzen, indem er eine binäre Semaphore einsetzt. Dies birgt trivialerweise den Vorteil, daß der Programmierer sich nicht um diese Aufgabe kümmern muß und so weniger Fehler passieren. Werden also alle kritischen Bereiche innerhalb eines Monitors implementiert, können zwei Prozesse ihre kritischen Abschnitte nie gleichzeitig ausführen.

2.1.5 Lebendigkeitsausfälle

Eines der am häufigsten vorkommenden Probleme bei der Entwicklung von Multitaskingsystemen stellen Lebendigeitsausfälle dar. Ein Lebendigkeitsausfall ist ein Zustand, in dem ein Prozeß oder Thread, aus welchem Grund auch immer, nicht mehr ausgeführt wird. Dieser Abschnitt zeigt die verschiedenen Arten des Lebendigkeitsausfall und geht auf den Fall der Verklemmung besonders intensiv ein.

2.1.5.1 Verhungern, Dauerschlaf und vorzeitige Beendigung

Verhungern

Die erste Art des Lebendigkeitsausfalls ist das Verhungern (Starvation), daß dann vorliegt, wenn eine Task, obwohl sie sich in einem lauffähigen Zustand befindet, nicht mehr ausgeführt wird, weil andere Tasks dauerhaft den Prozessor belegen. Dies kann z.B. dann der Fall sein, wenn eine große Anzahl von Berechnungen ausgeführt werden müssen, die in ihrer Priorität vor der verhungernden Task liegen. Um die Gefahr des Verhungerns gänzlich auszuschließen müßte man die Behandlung einzelner Tasks nach Prioritäten vermeiden und eine *first come, first serve* Strategie bezüglich der Prozessornutzung zugrunde legen.

Dauerschlaf

Im Falle des Dauerschlafs (Dormancy) kommt eine nicht laufende Task nie in einen Zustand der Ausführung. Dies geschieht z.B. dann, wenn die Task in die Sleep Queue des Betriebssystems eingereiht wurde, und es keine andere Task gibt, die die Weiterausführung initiiert. Dies ist z.B. dann der Fall, wenn eine Task, die als Druckserver fungiert auf eine Benachrichtigung des Druckertreibers wartet, der aber zwischenzeitlich beendet wurde oder sich in einer Starvation- Situation befindet.

Vorzeitige Beendigung

Eine vorzeitige Beendigung (*Premature Termination*) liegt dann vor, wenn eine Task durch eine Anweisung von außen (z.B. einem Signal) oder anderweitig zu früh beendet wird.

2.1.5.2 Verklemmung

Verklemmungen gehören zu den am häufigsten vorkommenden und am schwierigsten zu analysierenden Lebendigkeitsverlusten. Eine Menge von Tasks befindet sich in einem Verklemmungszustand (Deadlock), falls jede Task der Menge auf ein Ereignis wartet, das nur ein anderer Prozeß der Menge auslösen kann.

Für eine Deadlocksituation müssen folgende Bedingungen erfüllt sein (vgl. [CoElSh71]):

- **Bedingung des wechselseitigen Ausschlusses**: Jedes Betriebsmittel wird entweder von genau einem Task belegt oder ist verfügbar.
- **Belegungs- und Wartebedingung**: Eine Task, die bereits Betriebsmittel belegt, kann weitere Betriebsmittel anfordern.
- **Ununterbrechbarkeitsbedingung**: Die Betriebsmittel, die von einer Task belegt werden, können nicht entzogen werden, sondern müssen explizit vom belegenden Task freigegeben werden.
- **Zyklische Wartebedingung**: Es muß eine zyklische Kette aus zwei oder mehr Tasks existieren, so daß jede Task ein Betriebsmittel anfordert, das von dem nächsten Task der Kette belegt ist.

Das folgende Beispiel illustriert einen Ablauf, der zu einer Verklemmung führt. Es besteht aus zwei Tasks, die sich über zwei nicht blockierende Kommunikationsprimitiven (z.B. binäre Semaphore) s_1 und s_2 synchronisieren. Task 1 versucht zuerst ein DOWN auf s_1 und dann auf s_2 auszuführen und gibt sie danach jeweils durch ein UP wieder frei. Task 2 geht den selben Weg, akquiriert jedoch zunächst s_2 und dann s_1, um anschließend s_1 und dann s_2 freizugeben.

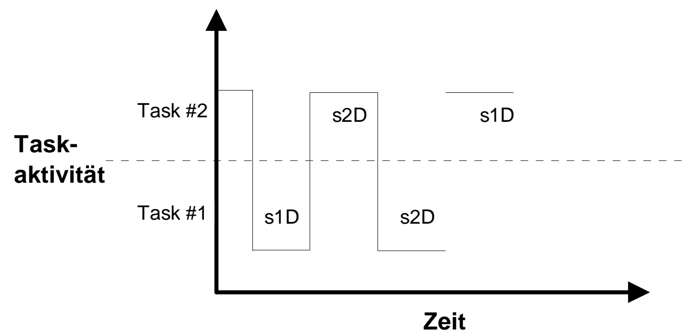

Abbildung 2-4: Beispiel eines Deadlocks

In Abbildung 2-4 ist der zeitliche Ablauf so gewählt, daß zunächst Thread 1 sein DOWN auf s_1 durchführen kann, danach jedoch der zweite Task s_2 aquirieren kann. Thread 1 versucht dies anschließend ebenfalls, muß jedoch blockieren. Anschließend will Thread 2 ein DOWN auf das von Thread 1 gehaltene s_1 ausführen und muß ebenfalls warten. Beide Threads sind nun nicht mehr in der Lage sich gegenseitig aufzuwecken. Es liegt ein Deadlock vor.

2.1.5.3 Livelocks

Ein Spezialfall in der Gruppe der Lebendigkeitsausfälle stellt der Livelock dar, da hierbei kein Lebendigkeitsausfall im eigentlichen Sinne vorliegt, bei dem eine Task, aus welchem Grund auch immer, nicht mehr ausgeführt wird. Vielmehr befindet sich hierbei eine Task in einem Zustand, in dem sie ausgeführt wird, jedoch eine unendliche Sequenz von nicht nach außen sichtbaren Aktionen durchführt. Dies ist z.B. dann der Fall, wenn eine Task in einen Zustand gerät, indem sie z.B. eine Schleife ausführt, nach deren Beendigung das eigentlich gewünschte Synchronisationsverhalten mit anderen Prozessen modelliert ist, die Task jedoch nie aus der Schleife springen kann, da z.B. eine Abbruchbedingung nie erfüllt werden kann. Das zu beobachtende Verhalten ist dem des *Verhungerns* (siehe 'Verhungern, Dauerschlaf und vorzeitige Beendigung' auf Seite 14) sehr ähnlich.

2.2 Nebenläufigkeit in Java

Die Ausführungen in den vorigen Abschnitten beschäftigten sich mit den allgemeinen Anwendungen und Problemen von Multitasking. Im folgenden wird speziell die Unterstützung der Nebenläufigkeit durch die Programmiersprache Java betrachtet. Innerhalb Java ist die gegenseitige Beeinflussung von Prozessen nicht vorgesehen, wohingegen einige mächtige Konstrukte zur Entwicklung von Applikationen mit mehreren Kontrollfäden auf Thread- Basis existieren.

Java gehört zu den relativ wenigen objektorientierten Programmiersprachen, die ohne den Einsatz von unterstützenden Werkzeugen und Bibliotheken Threads und verwandte Nebenläufigkeitskonstrukte anbieten. In Java gelingt deshalb die Programmierung von nebenläufigen Anwendungen in der Regel leichter und intuitiver als in den meisten anderen Programmiersprachen.

Im folgenden sollen zunächst die Möglichkeiten vorgestellt werden, die die Java- eigene *Thread- API* und spezielle Konstrukte in der Programmiersprache selbst bieten, um nebenläufige Programme zu entwickeln. Ein weiterer Abschnitt beschäftigt sich mit den Möglichkeiten der Synchronisation in Java und zeigt hierfür Beispiele (vgl. [Flan96] und [OaWo99]).

2.2.1 Die Java Thread- API

Dieser Abschnitt stellt die verschiedenen Möglichkeiten vor, die die Programmiersprache Java anbietet, um Anwendungen zu entwickeln, die mehrere Kontrollfäden enthalten.

2.2.1.1 Java Thread Konzepte

Java enthält einige grundlegende Konstrukte und Klassen, die speziell für die Unterstützung nebenläufiger Programmierung entworfen wurden. Auf Klassen- bzw. Schnittstellenebene sind dies die Klassen `java.lang.Thread` und `java.lang.Object`, sowie die Schnittstelle `java.lang.Runnable`. Als spezielle Java Schlüsselwörter zur Unterstützung von Synchronisationsaufgaben existieren `synchronized` und `volatile`. Java Threads werden auf der Basis der Java Virtual Machine in betriebssystemabhängige Threads übersetzt, so daß die Methoden innerhalb der API, die Threads direkt manipulieren als `native`, also systemabhängig, deklariert sind.

User und Deamon Threads

Es existieren innerhalb Javas zwei Arten von Threads: User und Daemon (Dämon) Threads. Der Unterschied zwischen diesen beiden Typen liegt darin, daß alle Daemon Threads automatisch beendet werden, wenn der letzte User Thread beendet wurde. Kontrollfäden, die als Dämonen definiert wurden, können somit als User- Thread- abhängige Services angesehen werden.

Scheduling

Das System, daß entscheidet, wann, wie oft und wie lange eine Task (in diesem Fall ein Java Thread) durch den Prozessor verarbeitet wird, heißt Scheduler (vgl. [Tan95]). Der Java Scheduler benutzt das sogenannte Prioritätenschedulingverfahren, bei dem jede ganzzahlige Priorität mit einer bestimmten Warteschlange verbunden ist. Jeder Thread besitzt eine Priorität, wobei ein Thread mit hoher einem Thread mit geringerer Priorität bei der Ausführung vorgezogen wird. Die einer angegebenen Task zugewiesene Priorität muß dabei so gewählt, daß die relative Dringlichkeit im Vergleich zu anderen Tasks in einem bekannten oder unbekannten Kontext berücksichtigt wird. So sollten z.B. Threads die in Interaktion mit dem Benutzer eines Programms stehen im Vergleich zu Threads, die beispielsweise langwierige Berechnungen ausführen, eine höhere Priorität haben.

2.2.1.2 API Klassen und Schnittstellen

Dieser Abschnitt zeigt eine Übersicht über die unterschiedlichen API Klassen und Schnittstellen, mit denen sich Threadfunktionen ansprechen lassen und stellt deren Beziehungen untereinander dar.

Das folgende Diagramm zeigt die zur Threadsteuerung vorgesehenen Klassen inklusive ihrer Abhängigkeiten.

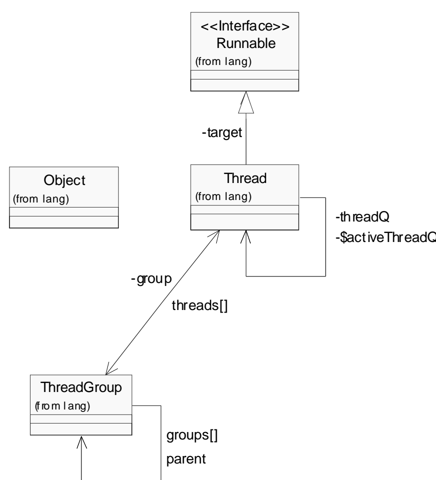

Abbildung 2-5: Abhängigkeiten der threadrelevanten Klassen

Die Klasse java.lang.Object

Die Klasse `java.lang.Object` stellt die Wurzel der Klassenhierarchie in Java dar. Jede Klasse ist eine Unterklasse von `java.lang.Object`. Interessant für die Betrachtung der Nebenläufigkeit in Java sind die Methoden `notify`, `notifyAll` sowie diverse Signaturen der Methode `wait`.

- `notify`: Die Methode `notify` weckt einen einzelnen Thread, der auf das Freiwerden eines Monitors für das Objekt wartet, auf dem die Mehode aufgerufen wurde.
- `notifyAll`: Äquivalent zu `notify` weckt `notifyAll` alle Threads, die auf den Monitor für das Objekt warten.
- `wait`: Die wait Methoden stellen die Gegenstücke zu den `notify` Methoden dar. `wait(long timeout)` wartet maximal die angegebene Timeoutzeit lang auf eine Benachrichtigung mittels eine der notify Methoden. Ist die Benachrichtigung erfolgt oder die Timeoutzeit abgelaufen setzt der entsprechende Thread seine Abarbeitung fort. `wait(long timeout, int nanos)` und `wait()` legen ein äquivalentes Verhalten an den Tag, mit dem Unteschied, daß die Wartezeit noch feiner definiert werden kann, bzw. unendlich groß ist.

`java.lang.Object` enthält also im Bezug auf Threads hauptsächlich Methoden zur Synchronisierung.

Das Interface java.lang.Runnable

Das Interface `java.lang.Runnable` sollte von allen Klassen, deren Instanzen innerhalb von Threads ausgeführt werden sollen, implementiert werden. Das Interface dient dazu, ein gemeinsames Protokoll für Objekte, die Code innerhalb eines Threads ausführen wollen, darzustellen. Es enthält eine abstrakte Methode `run`, die überladen werden muß. Jede Klasse, die `java.lang.Runnable` implementiert muß deshalb eine Methode `public void run()` enthalten, die den innerhalb des Threads auszuführenden Code enthält. Es ist dadurch möglich Klassen zu erzeugen, die keine Unterklassen von java.lang.Thread sind, deren Objekte jedoch als eigenständige Kontrollfäden lauffähig sind.

Die Klasse java.lang.Thread

Die weitreichendsten Funktionalitäten stellt die Klasse `java.lang.Thread` zur Verfügung. Sie verfügt über Möglichkeiten zum Threadaufbau, zur Threadsteuerung und zur Veränderung von Prioritäten und Abarbeitungsfolgen. Sie implementiert das Interface Runnable und besitzt somit eine Methode `run`, die den auszuführenden Code enthält. Diese Methode wird ausgeführt, wenn der Thread gestartet wird. Sie enthält initial keinen eigenen ausführbaren Code, so daß sie in konkreten Anwendungen überladen werden muß.
Die Klasse `java.lang.Thread` enthält alle Methoden, die in den Abschnitten 2.2.1.3 (Seite 18), 2.2.1.4 (Seite 18) und 2.2.1.5 (Seite 19) erläutert werden.

Die Klasse java.lang.ThreadGroup

Threads können innerhalb Javas gruppiert werden. Hierzu dient die Klasse `java.lang.ThreadGroup`, die den Zugriff auf mehrere Threads automatisiert. Dies hat den Vorteil, daß mehrere Threads mit einem Methodenaufruf manipuliert werden können, wodurch die Gefahr minimiert wird, daß ein oder mehrere Threads vergessen werden. Die Threadgruppe dient jedoch hauptsächlich dem Erzwingen von Sicherheitsstrategien, indem sie den Zugriff auf Threadoperationen dynamisch beschränkt. So ist es z.B. nicht erlaubt, einen nicht zur aktuellen Gruppe gehörenden Thread zu stoppen.
Des weiteren ist es möglich, Hierarchien von Threadgruppen zu bilden. So werden beispielweise alle Unterthreadgruppen eines Threads suspendiert, wenn die Elterngruppe suspendiert wird. Per Voreinstellung ist jeder Thread Mitglied der der Gruppe `Thread`. Konstruiert ein Objekt eine neue Threadgruppe, wird es unterhalb der aktuellen Gruppe eingeordnet.
Folgende Methoden, die auf Objekte des Typs `java.lang.Thread` aufgerufen wer-

den können, sind mit gleicher Semantik auch auf Threadgruppen ausführbar: `stop()`, `resume()`, `interrupt()`, `setDaemon()` und `destroy()`. Eine Beschreibung der einzelnen Methoden findet sich im Abschnitt 2.2.1.3 (Seite 18) und 2.2.1.4 (Seite 18). Eine Methode mit der sich angeben läßt, wie hoch die Priorität der Threads innerhalb einer Gruppe sein sollen, ist dabei nicht enthalten. Stattdessen läßt sich mittels `setMaxPriority()` angeben, welche Priorität die einzelnen Threads maximal haben können.

Die Ausnahme java.lang.ThreadDeath

Die `ThreadDeath` Ausnahme wird geworfen, sobald ein Thread, bzw.die Methode `run()` innerhalb eines Threads terminiert. Dies kann zum einen dadurch passieren, daß `run` tatsächlich erfolgreich abgearbeitet wurde, zum anderen kann es jedoch auch sein, daß der Thread von "außen" (z.B. durch `destroy`) terminiert wurde oder es innerhalb des Threads zu einer anderweitigen Ausnahme kam, die die weitere Ausführung ausschließt. Die `ThreadDeath` Exception dient dabei in der Regel zur internen Verwaltung innerhalb von Java Programmen und sollte nicht explizit vom Programmierer abgefangen werden.

2.2.1.3 Threadaufbau und -abbau

Die folgende Aufzählung zeigt die einzelnen Instanzmethoden von `java.lang.Thread` zum Auf- und Abbau von Threads.

- `Thread` (Konstruktor): Diese Methode dient zur Erzeugung eines Threads. Als Parameter können der Name des zu erzeugenden Threads, und ein von `java.lang.Runnable` abgeleitetes Zielobjekt angegeben werden, dessen `run` Methode zur Ausführung dann kommt. Wurde kein Zielobjekt angegeben, wird die `run` Methode des Objekts dieser Klasse verwendet.
- `setDaemon`: Diese Methode setzt die Dämoneigenschaften des Threads. Initital ist die Eigenschaft abgeschaltet, so daß es sich bei frisch initialisierten Threads immer um User Threads handelt.
- `setName`: Gibt der Threadinstanz einen Namen. Dies ist beim Erzeugen von Ereignisspuren und beim Beheben von Fehlern nützlich, spielt aber ansonsten keine Rolle.
- `getName`: Gibt den Namen der Threadinstanz zurück.
- `start`: Dient zur Erzeugung eines neuen Threads des Typs dieser Klasse und startet ihn. Die `run-` Methode wird dabei automatisch ausgeführt. Keine der Synchronisationssperren des aufgerufenden Threads bleiben automatisch erhalten. Solange keine spezielle Steuerungsmethode (z.B. `stop`) auf dem Thread aufgerufen wurde, endet er mit der Beendigung der `run` Methode.
- `stop`: Stoppt die aktuelle Threadinstanz. Verwaltungsinformationen werden automatisch zurückgesetzt, so daß z.B. alle Locks, die die Threadinstanz belegt freigegeben werden.
- `destroy`: Zerstört einen aktuelle Thread, wobei keinerlei Rücksicht auf den momentanen Zustand des Threads genommen wird. Es werden keine Verwaltungsinformationen bereinigt, so daß z.B. alle Locks, die sich auf den Thread beziehen, unverändert bleiben .

2.2.1.4 Threadsteuerung

Im folgenden werden die Methoden betrachtet, die die Klasse `java.lang.Thread` zur Steuerung von Threads zur verfügung stellt.

- `isAlive`: Diese Methode liefert zurück, ob sich der angebene Thread noch in Ausführung befindet, oder beispielsweise mittels `stop` beendet wurde.
- `sleep`: Diese Methode unterbricht den Thread für einen festgelegten Zeitraum und läßt ihn dann automatisch weiterlaufen. Wenn noch andere Threads aktiv sind, wird

der Thread möglicherweise nach Ablauf der festgesetzten Zeit nicht sofort wieder aufgenommen.

- `suspend`: Die Methode hält einen Thread vorübergehend an, so daß er normal weiterläuft, wenn ein Thread die Methode `resume` auf diesem Thread aufruft.
- `resume`: Diese Methode stellt das Gegenstück zu `suspend` dar. Es läßt einen mittels `suspend` suspendierten Thread weiterlaufen.
- `join`: Ein Aufruf dieser Methode bewirkt, daß auf die Beendigung des Threads gewartet wird.
- `interrupt`: Der Aufruf dieser Methode auf eine Threadinstanz bricht eine in Ausführung befindliche Operation, die mittels `sleep`, `wait` oder `join` angestoßen wurde mit einer `InterruptedException` (siehe 'Ausnahmen im Bezug auf Threads' auf Seite 19) ab.
- `activeCount`: Diese Klassenmethode gibt an, wieviele Threads im ganzen Programm noch aktiv sind. Dies betrifft alle Threads, auf die ein Aufruf von `isAlive` ein positives Ergebnis liefern würde.

2.2.1.5 Beeinflussung von Threadprioritäten

Wenn die Hardwareausstattung des Zielsystems nicht so gestaltet ist, daß jeder Thread eine eigene CPU zugewiesen bekommen kann, teilen sich alle aktiven Java Threads die vorhandenen Prozessoren. Das heißt, daß jeder lauffähige Thread abwechselnd mit einem anderen immer nur eine Zeit lang ausgeführt wird. Ein Thread ist genau dann lauffähig, wenn er sich innerhalb zweier Aufrufe von `start` und `stop` befindet und nicht von einer `wait` Methode betroffen ist.
Lauffähige Threads, die nicht ausgeführt werden, verwaltet das Java Laufzeitsystem unter Berücksichtigung von Prioritäten in Warteschlangen. Dabei enthält jeder Thread per Voreinstellung dieselbe Priorität wie der ihn erzeugende Thread.
Existieren gleichzeitig mehrere lauffähige Threads, sucht das Java Laufzeitsystem einen Thread aus, der nach den folgenden Regeln die höchste Priorität hat (vgl. [Lea97]):

- Sind mehrere Threads mit der höchsten Priorität vorhanden, wählt die Virtual Machine willkürlich einen davon aus, um ihn als nächstes zur Ausführung zu bringen (das entspricht nicht unbedingt der Forderung der Fairneß, wie er z.B. in [Tan95] beschrieben ist).
- Ein laufender Thread mit niedriger Priorität wird ausgesetzt, also vom Laufzeitsystem unterbrochen, wenn ein Thread mit höherer Priorität ausgeführt werden muß. Dagegen werden Threads mit gleicher Priorität nicht unbedingt füreinander unterbrochen.

Die folgende Aufzählung zeigt die einzelnen Instanzmethoden von `java.lang.Thread` zur Beeinflussung von Threadprioritäten.

- `setPriority`: Diese Methode setzt die Priorität der Threadinstanz. Die Prioritätsparameter können zwischen `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY` liegen, wobei die Minimum- und Maximumkonstanten als Klassenattribute von `java.lang.Thread` gehalten werden und innerhalb eines Java Programms nicht geändert werden dürfen.
- `yield`: Die Methode `yield` dient zur Steuerungsübertragung, so daß ein oder mehrere andere Threads ausgeführt werden können.

2.2.1.6 Ausnahmen im Bezug auf Threads

Während der Generierung und der Laufzeit eines Threads können eine Reihe von Ausnahmen auftreten, die unterschiedliche Gründe haben und sich in den meisten Fällen sauber voneinander abgrenzen lassen. Im folgenden sind die einzelnen Thread- bezogenen Ausnahmen und die Methoden aufgeführt, die diese werfen können, aufgeführt.

java.lang.InterruptedException
Die in der Praxis am häufigsten auftretende Ausnahme ist die Unterbrechungsausnahme. Sie zeigt an, daß die Methode, die diese Ausnahme wirft, früher als erwartet beendet wurde. Folgende Methoden können diese Ausnahme explizit erzeugen:

- `join`: der Thread, auf dessen Terminierung gewartet wird, ist nicht beendet worden.
- `sleep`: die Zeit, die der Thread suspendiert werden sollte, wurde nicht erreicht.
- `wait`: `wait` hat (innerhalb einer unter Umständen angegebenen Timeoutzeit) kein entsprechendes notify erhalten.

java.io.InterruptedIOException
Verschiedene Methoden von Klassen, die Ein- und Ausgaben ausführen, können die `InterruptedIOException` werfen, wenn sie mittels `interrupt` unterbrochen wurden und sich nicht innerhalb einer Warteschlange befindet. Die Abgrenzung der Methoden, die diese Ausnahme werfen ist sehr schwierig, da manche Eingage-/ Ausgabeoperationen innerhalb der Java- API unterbrechbar sind und manche nicht. Innerhalb der verschiedenen API Versionen wird dies zusätzlich unterschiedlich gehandhabt. Laut [OaWo99] plant Sun, diesen Misstand erst in einer der nächsten Versionen, deren Versionsnummer über 2 liegen wird, zu beheben, so daß sich zum jetzigen Zeitpunkt hierüber noch keine verläßlichen Angaben machen lassen.

java.lang.IllegalThreadStateException
Eine Ausnahme, die den unzulässigen Zustand eines Threads bei einer Anfrage anzeigt, ist die `IllegalThreadStateException`. Folgende Methoden können diese Ausnahme hervorrufen:

- `Thread`**- Konstruktoren:** Für den Thread wurde eine bereits zerstörte Threadgruppe angegeben.
- `start`: `start` wurde ein zweites Mal auf die Threadinstanz aufgerufen.
- `setDaemon`: Die Methode wurde nach dem Start einer Threadinstanz aufgerufen.
- `destroy`: Der Thread wurde bereits zerstört.

java.lang.IllegalArgumentException
Es ist möglich daß eine Methode der Threadklassen mit inkorrekten Parametern aufzurufen wird. Die `IllegalArgumentException` wird genau in diesem Fall geworfen, die bisher von genau einer Methode erzeugt werden kann.

- `setPriority`: Die angegebene Priorität hat die zulässigen Minimal- bzw. Maximalwerte für Threadprioritäen unter bzw. überschritten.

java.lang.IllegalMonitorStateException
Die `IllegalMonitorStateException` wird vom Threadsystem geworfen, wenn versucht wird, eine Operation auf einen Monitor ausgeführt werden soll, dieser jedoch die Anfrage nicht bearbeiten kann, weil er sich in einem falschen Zustand dafür befindet.

- `wait`, `notify` und `notifyAll`: Die Methode wurde auf einem Objekt aufgerufen, für das kein Monitor existiert.

java.lang.SecurityException
Innerhalb des Java Threadmechanismus und innerhalb von Hierarchien von Threadgruppen, ist es möglich, Zugriffsberechtigungen zu setzen, die eingehalten werden müssen. Ist eine Zugriff auf einen Thread oder eine Threadgruppe nicht möglich, wird die `SecurityException` geworfen.

- `Thread.checkAccess` und `Threadgroup.checkAccess`: Es wurde versucht, den Zugriff auf eine Threadgruppe zu testen, auf die dem Thread oder der Threadgruppe kein Zugriff erlaubt ist.
- `setPriority`: Die angegebene Priorität hat die zulässigen Minimal- bzw. Maximalwerte der Threadgruppe unter bzw. überschritten.

2.2.2 Synchronisation in Java

Greifen mehrere Threads auf dasselbe Objekt zu, so können sie sich gegenseitig beeinflussen. Folgende Konflikte beim Zugriff auf gemeinsam genutzte Objekte auf unterster Implementierungsebene können auftreten (vgl. [Lea97]):

- **Lese/Schreib- Konflikte**: Führt ein Thread eine nicht atomare Schreibaktion auf einen Speicherbereich aus und liest ein anderer Thread in einer Unterbrechung der Schreibaktion die Daten aus, sind falsche oder sinnlose Werte die Folge.
- **Schreib/ Schreib- Konflikte**: Führen mehrere Threads Schreibaktionen auf denselben Speicherbereich aus und unterbrechen sich dabei, sind falsche oder sinnlose Werte die Folge.

Um solche Probleme zu vermeiden, stellt Java eine Reihe von Synchronisationsmöglichkeiten zur Verfügung, die nachfolgend besprochen werden.

2.2.2.1 *Verwendung des* `synchronized` *Schlüsselworts*

Das wichtigste Hilfsmittel zur Vermeidung von Synchronisationskonflikten ist das Schlüsselwort `synchronized`, das auf jede Java Methode und jeden Codeblock innerhalb jeder Methode als Qualifizierer verwendet werden darf. Es wird hauptsächlich dazu verwendet, daß immer nur ein Thread auf ein Objekt zugreifen kann, das die Methode oder der Codeblock verändert oder liest. Für jeden `synchronized` Block wird hierbei eine Monitor aquiriert (siehe ’Synchronisation’ auf Seite 12). Dies vermeidet ein willkürliches Verzahnen der Aktionen aus Methodenrümpfen.

Atomare Operationen in Java

Java sorgt dafür, daß die meisten einfachen Operationen ununterbrechbar (atomar) sind und somit in Umgebungen mit mehreren Threads ohne ausdrückliche Synchronisation ohne Zugriffkonflikte funktionieren. Hierzu gehören Zugriffs- und Zuweisungsoperationen auf allen eingebauten skalaren Typen außer `long` und `double`, die eine Sonderstellung haben (siehe auch ’Code- Segmente’ auf Seite 57). Dies hat zur Folge, daß nicht synchronisierte Operationen auf Größen dieser Typen bei nebenläufigen Zugriff zu falschen Werten führen können.

2.2.2.2 *Warten und Benachrichtigen*

Für den Aufruf der Methoden `Object.wait`, `Object.notify` und `Object.notifyAll` muß die Vorraussetzung gelten, daß auf das Zielobjekt eine Synchronisationssperre gilt. Normalerweise wird dies über die ausschließliche Verwendung dieser Methoden in Methoden oder Blöcken erreicht, die auf ihre Zielobjekte synchronisiert sind. Diese Bedingung kann in der Regel nicht zur Übersetzungszeit geprüft werden, so daß Abweichungen hiervon zur Laufzeit eine Ausnahme vom Typ `IllegalMonitorStateException` erzeugt.

Semantik von java.Object.wait

Ein Aufruf von `wait` hat folgende Aktionen zur Folge (vgl. [Lea97]):

- Der aktuelle Thread wird angehalten.
- Das Java Laufzeitsystem plaziert den Thread in eine interne und von außen nicht manipulierbare Warteschlange, die mit dem Zielobjekt verbunden ist.
- Die Synchronisationssperre für das Zielobjekt wird aufgehoben, während alle anderen durch den Thread gehaltenen Sperren bestehen bleiben.

Tritt während eines `wait` ein `interrupt` auf, kommt derselbe `notify`- Mechanismus zum tragen, außer, daß die Steuerung auf das mit dem Aufruf des `wait` verbundene `catch`- Konstrukt wieder übergeben wird.

Semantik von java.Object.notify und java.Object.notifyAll

Ein Aufruf von `notify` hat folgende Aktionen zur Folge:
Falls vorhanden, wird ein willkürlich ausgewählter Thread *T* vom Java Laufzeitsystem aus der mit dem Zielobjekt verbundenen internen Warteschlange entfernt.

- *T* muß die Synchronisationssperre für das Zielobjekt wiedererlangen. Dieses läßt *T* immer wenigstens so lange blockieren, bis der Thread, der `notify` aufruft, die Sperre aufhebt. Erlangt ein anderer Thread vorher diese Sperre, bleibt *T* weiterhin blokkiert.
- *T* wird nach der Stelle wiederaufgenommen, an dem vorher `wait` aufgerufen wurde.

Die Methode `notifyAll` hat äquivalente Auswirkungen wie `notify`, mit dem Unterschied, daß die einzelnen Schritte für alle in der Warteschlange des Zielobjekts befindlichen Threads ausgeführt werden.

2.2.3 Beispiele

Dieser Abschnitt liefert einige Beispiele für die Verwendung der Java Thread-API.

Threadaufbau/ -abbau

Die Erzeugung einer Threadklasse kann, wie in Abschnitt 2.2.1.2 (Seite 16) und Abschnitt 2.2.1.3 (Seite 18) erläutert, über die Implementierung des Interfaces `java/lang/Runnable` oder über das Erben der Klasse `java/lang/Thread` durchgeführt werden. Folgendes Beispiel zeigt einen Klassenrumpf, der eine Threadklasse zur Berechnung von Primzahlen zur Verfügung stellt und das Interface `Runnable` implementiert.

```
class PrimeRun implements Runnable {
  long minPrime;
  PrimeRun(long minPrime) {
    this.minPrime = minPrime;
  }

  public void run() {
  // compute primes larger than minPrime ...
  }
}
```

Abbildung 2-6: Threadklasse, die das Runnable Interface implementiert

Zum Starten des Threads muß zunächst eine Instanz der Klasse erstellt werden, die an den Konstruktor von `java/lang/Thread` übergeben wird:

```
  PrimeRun p = new PrimeRun(143);
  new Thread(p).start();
```

Folgendes Beispiel zeigt im Gegensatz dazu die gleiche Klasse, die `java/lang/Thread` erweitert.

```
class PrimeThread extends Thread {
  long minPrime;
  PrimeThread(long minPrime) {
    this.minPrime = minPrime;
  }

  public void run() {
  // compute primes larger than minPrime ...
  }
}
```

Abbildung 2-7: Threadklasse, die java/lang/Thread erweitert

Das Starten dieser Klasse kann direkt über den Aufruf der Start- Methode realisiert werden:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

Wenn ein Thread erst einmal gestartet wurde, kann er wieder beendet werden. Dies zeigt folgendes Beispiel von vier Primzahlen- berechnenden Threads, die zunächst gestartet und anschließend der Reihe nach wieder gestoppt werden.

```
PrimeThread p1 = new PrimeThread(100);
PrimeThread p2 = new PrimeThread(200);
PrimeThread p3 = new PrimeThread(300);
PrimeThread p4 = new PrimeThread(400);
p1.start(); p2.start(); p3.start();p4.start();
p1.stop(); p2.stop();p3.stop();p4.stop();
```

Threadsteuerung und Threadpriorisierung

In Abschnitt 2.2.1.4 (Seite 18) werden die Methoden `suspend` und `resume` beschrieben. Folgendes Beispiel zeigt einen einfachen Blinker mit einem Thread, der permanent eine Bildschirmausgabe erzeugt:

```
class Blinker extends Thread {
  public void run() {
    for (;;) {
      System.out.println("BLINK");
      this.yield();
    }
  }

  public static void main(String args[]) {
    Blinker blinker=new Blinker();
    blinker.start();
    for (;;) {
      blinker.suspend();
      //.. do something
      blinker.resume();
    }
  }
}
```

Abbildung 2-8: Beispiel für den Einsatz von suspend, resume und yield

Der Hauptthread (`main`) suspendiert und aktiviert diesem Thread fortlaufend, so daß der Eindruck des Blinkens entsteht.
Ebenso wird die in Abschnitt 2.2.1.5 (Seite 19) vorgestellte Methode `yield` verwendet, die in unserem Fall den Scheduler anweist nach genau einer Ausgabe einen anderen Thread auf den Prozessor zu legen. Hierdurch wird sichergestellt, daß jeweils auch wirklich nur eine Ausgabe erzeugt wird.

Synchronisation

Das folgende Programm stellt eine Lösung zum in Abschnitt 2.1.3 (Seite 11) vorgestellten Bankiersproblem dar. Zur Synchronisierung des konkurrierenden Zugriffs auf einen kritischen Abschnitt, wird dieser in einen synchronisierten Block (siehe 'Verwendung des synchronized Schlüsselworts' auf Seite 21) eingebettet.

```
class BankersProb {
  private Account account;

  class Account {
    private int amount=0;

    public synchronized void changeAmount(int val) {
      int x=amount;
      x=x+val;
      if(x>0)
        amount=x;
      System.out.println(amount);
    }
  }

  class IncThread extends Thread {
    public void run() {
      for (;;) account.changeAmount(100);
    }
  }

  class DecThread extends Thread {
    public void run() {
      for (;;) account.changeAmount(-100);
    }
  }

  BankersProb() {
    this.account=new Account();
    IncThread t1=new IncThread();
    DecThread t2=new DecThread();
    t1.start(); t2.start();
  }
}
```

Abbildung 2-9: Beispiel für den Einsatz des synchronized Schlüsselworts

Ein oftmals angeführtes Beispiel für Warten und Benachrichtigen (siehe 'Warten und Benachrichtigen' auf Seite 21) ist das Leser/ Schreiber (Producer/ Consumer Problem, vgl. [Court+71]), bei dem zwei Threads über einen gemeinsamen Puffer verfügen. Dieser Puffer wird von einem Prozeß gefüllt und vom anderen ausgelesen. Hierbei liegt das Problem vor, daß der füllende Prozeß nur dann in den Puffer schreiben darf, wenn dort auch Platz ist und der lesende nur Lesen darf, wenn sich ein Element im Puffer befindet. In diesem Beispiel existiert ein Schreiber- (Producer-) und ein Leser- (Consumer-) Thread, die auf ein gemeinsames Pufferobjekt zugreifen. Dieser Pufferobjekt verwendet `wait` und `notify`, um den Leser und den Schreiber entsprechend der aufgestellten Anforderungen zu synchronisieren.

```
class Producer extends Thread {
  private Buffer buffer;

  public Producer(Buffer b) {
    buffer = b;
  }

  public void run() {
    for (;;) buffer.put();
  }
}

class Consumer extends Thread {
  private Buffer buffer;

  public Consumer(Buffer b) {
    buffer = b;
  }

  public void run() {
    for (;;) buffer.get();
  }
}

class Buffer {
  private boolean available = false;

  public synchronized void get() {
    while (available == false) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    available = false;
    notify();
  }

  public synchronized void put() {
    while (available == true) {
      try {
        wait();
      } catch (InterruptedException e) {}
    }
    available = true;
    notify();
  }
}

public class ProducerConsumerTest {
  public static void main(String args[]) {
    Buffer c = new Buffer();
    Producer p1 = new Producer(c);
    Consumer c1 = new Consumer(c);
    p1.start();   c1.start();
  }
}
```

Abbildung 2-10: Beispiel für den Einsatz von wait und notify zur Synchronisation

2.3 CSP und FDR

Dieser Abschnitt beschäftigt sich mit der Zielsprache des Systems und dem damit verbundenen Auswertungswerkzeug. Die Spezifikationssprache CSP (Communicating Sequential Processes), wurde von C.A.R. Hoare entwickelt (vgl. [Hoare85]) und bietet formale Methoden, mit denen sich Eigenschaften von Systemen mit parallel laufenden Prozessen verifizieren lassen.

FDR (Failure Divergence Refinement, vgl. [Formal97]) ist ein auf den Grundlagen für Nebenläufigkeitskonstrukte von CSP aufbauendes Werkzeug, mit dem sich Modelle gegen Spezifikationen automatisiert prüfen lassen.

Communicating Sequential Processes (CSP)

CSP ist eine Spezifikationssprache, in der Prozesse ihre Zustände verändern können, indem sie Ereignisse (Events) auslösen. Prozesse können mit Operatoren zusammengestellt werden, die eine Synchronisation auf Ereignisse erzwingen. Jede Komponente dieser Zusammenstellung muß dabei bereit sein, an einem gegebenen Ereignis zu partizipieren, bevor das ganze System in einen anderen Zustand übergehen kann.

Die Theorie CSPs basiert auf mathematischen Modellen, die zunächst nichts mit der Sprache zu tun haben. Diese Modelle definieren das in Form von Events, nach außen sichtbare Verhalten von Prozessen und versuchen nicht, ein komplettes Bild des Prozessverlaufs zu untersuchen.

CSP ist explizit nicht als Programmiersprache anzusehen, sondern als Modell zur Beschreibung von Algorithmen mit synchronen Nachrichtenaustausch. Für eine Programmiersprache besitzt CSP nicht genug Möglichkeiten zur Abstraktion klassischer Programmierkonstrukte.

2.3.1 Sprachelemente von CSP

Dieser Abschnitt führt in die Sprachelement von CSP ein und gibt einige Beispiele, die in abgewandelter Form auch in [Baillie98] zu finden sind.

2.3.1.1 Prozesse, Alphabete und Ereignisse in CSP

Ein Prozeß ist ein Objekt, das ein Verhalten besitzt, welches als eine Aneinanderreihung von Ereignissen, die dieser Prozeß initiiert, definiert ist. Die Menge der Ereignisse, auf die der Prozeß zurückgreifen kann, wird Alphabet (α) genannt.

Das Alphabet eines einfachen Getränkeverkaufsautomaten `VA` kann, definiert über den Einwurf einer Mark und der Ausgabe einer Cola, z.B. folgendermaßen aussehen:

α`VA= {1DM, Cola}`

Sein Verhalten läßt sich dann mit folgender Gleichung beschreiben:

```
VA= 1DM -> Cola -> VA
```

Hierbei wird der Verkaufsautomat als rekursiver Prozeß `VA` beschrieben, der fortlaufend die Ereignisse `1DM` und nachfolgend `Cola` auslösen kann, wobei der Pfeil (`->`) hierbei die sequentielle Komposition von Ereignissen darstellt.

Vordefinierte Prozesse

Im folgenden werden einige in CSP vordefinierte Prozesse beschrieben, die ein vordefiniertes Verhalten aufweisen.

- $\mathbf{STOP}_\alpha$: Beschreibt einen Prozeß mit dem Alphabet α, der niemals ein Ereignis aus α ausführt.
- $\mathbf{CHAOS}_\alpha$: Beschreibt einen Prozeß mit dem Alphabet α, der beliebig Ereignisse aus α ausführt.
- **SKIP**: Beschreibt den erfolgreich abgearbeiteten Prozeß.

2.3.1.2 Spuren (Traces) in CSP

Die Menge der Spuren eines Prozesses ist die Menge aller möglichen Sequenzen der Ereignisse, die ein Prozeß auslösen kann. Für einen Prozeß *P* wird die Spurenmenge als `trace(P)` bezeichnet.

Für den Prozeß `VA` ergibt sich somit folgende Spurenmenge:
`traces(VA)={<>,<1DM>,<1DM,Cola>,<1DM,Cola,1DM>, ... }`, wobei `<>` die leere Spur darstellt, die für jeden Prozeß möglich ist.

2.3.1.3 Alternativen (Choices) in CSP

Wenn ein Prozeß nicht nur linear arbeiten, sondern über Verzweigungen verfügen soll, die entweder nichtdeterministisch oder von außen gesteuert sind, bedarf dies der Einführung von Alternativen (choices).

Intern und extern gesteuerte Alternativen

Die generelle Notation der Alternative ist senkrechte Balken (|). Die allgemeine Alternative läßt sich in extern gesteuerte Alternative ($\Box$, external choice) und intern gesteuerte Alternative (Π, internal choice) unterteilen.

Zur Verdeutlichung der Unterschiede zwischen beiden Varianten betrachten wir den (defekten) Verkaufsautomaten `VAmzh`, wobei der Automat entweder nach Einwurf des Geldes ein Getränk zurückgibt, oder stoppt.

```
VAmzh=    1DM -> Cola -> VAmzh
          □
          1DM -> STOP
```

Nachdem das Ereignis `1DM` ausgeführt wurde, soll sich das System entscheiden, welcher Zweig der Alternative ausgeführt wird.
Im obigen Versuch ist das nicht unbedingt der Fall, da die Alternative durch die Verwendung des external- choice Operators extern gesteuert ist und es somit zu einer Beeinflussung durch die Umgebung (das Environment) des Prozesses kommen kann. Die Umgebung kann dabei eine Menge parallel laufender Prozesse mit (zum Teil) gleichen Alphabet sein (siehe auch 'Nebenläufigkeit in CSP' auf Seite 27). Damit ist das Verhalten von `VAmzh` deterministisch.
Da hingegen benutzt der folgende Automat die intern gesteuerte Alternative und gibt somit das von uns gewünschte Verhalten wieder, da er tatsächlich intern (also nichtdeterministisch) entscheidet, welchen Zweig der Alternative er ausführt.

```
VAmzh=    1DM -> Cola -> VAmzh
          Π
          1DM -> STOP
```

if-then-else Alternative

Die dritte Möglichkeit der Alternative ist das if- then- else Konstrukt, daß auf eine angegebene Bedingung reagieren kann und je nach Ergebnis dieser Bedingung den einen oder den anderen Zweig ausführt:

```
f(x) = if x=0 then STOP
       else f(x-1)
```

In oben definierter Funktion `f` wird der STOP Prozeß ausgeführt, wenn der an `f` übergebene Wert 0 ist, andernfalls geht die Funktion mit dem dekrementierten Übergabewert in Rekursion.

2.3.1.4 Nebenläufigkeit in CSP

Wenn zwei Prozesse (z.B. P und Q) parallel zueinander ausgeführt werden (CSP Notation: `P || Q`), benötigen sie Synchronisation auf Ereignissen, die in den Alphabeten beider Prozesse enthalten sind. Für einen uns schon bekannten Getränkeverkaufsautomaten

und einen Getränkeverkaufsautomatennutzer würde das Verhalten folgendermaßen aussehen:

VA= 1DM -> Cola -> VA

VAN= Cola -> VAN
□
1DM -> Cola -> VAN

In diesem Fall kann der Automatennutzer (VAN) das Ereignis Cola auslösen, ohne vorher bezahlt zu haben (das Ereignis 1DM ausgelöst zu haben). Der Verkaufsautomat läßt jedoch Cola als initiales Ereignis nicht zu, so daß die erste Alternative des Verkaufsautomatennutzers in einem kombinierten System von VA und VAN nicht erlaubt ist. Die beiden Prozesse müssen sich also untereinander auf geteilten Ereignissen synchronisieren. Kein Prozeß kann somit mit seiner Ausführung fortfahren, bevor er nicht bereit ist, am Gesamtsystem zu kooperieren.
Das Verhalten des Gesamtsystems kann also wie folgt ausgedrückt werden:

```
(VA || VAN) = 1DM -> Cola -> (VA || VAN)
```

Bisher haben wir nur Systeme untersucht, deren Prozesse ein gleiches Alphabet besitzen. Wenn jedoch zwei Prozesse P und Q nebenläufig sind, wobei deren Alphabete nicht identisch sind ($\alpha P \neq \alpha Q$), müssen sie sich auf die Schnittmenge beider Alphabete synchronisieren ($\alpha P \cap \alpha Q$). Alle anderen Ereignisse sind unabhängig und somit nicht an der Synchronisation beteiligt.

2.3.1.5 Kommunikation in CSP

Die bisher erläuterten Sprachkonstrukte haben sich mit parallel ausgeführten Prozessen und deren Synchronisation auf einzelnen Ereignissen beschäftigt, wobei keine Daten zwischen den einzelnen Prozessen ausgetauscht wurden. Die Übertragung von Daten von einen zu einem anderen Prozeß wird in CSP als *Kommunikation* (Communication) bezeichnet, die eine spezielle Klasse von Ereignissen darstellt.
Hierbei ist ein Kommunikationereignis nicht durch ein einzelnes Ereignis beschrieben, sondern durch ein Tupel, das aus einem Kanal (channel) und einer Nachricht, die über diesen Kanal zu versenden oder zu empfangen ist beschrieben wird.
Um eine Nachricht *x* über einen Kanal c zu versenden muß das Kanal/ Empfänger Tupel (*c*,*x*) mit dem Ausgabeoperator Ausrufezeichen (!) versehen werden, so daß sich der Ausdruck c!x ergibt. Für das Empfangen von Nachrichten ergibt sich äquivalent ein Ausdruck mit dem Eingabeoperator Fragezeichen (?): c?x.
So kann z.B. ein Puffer wie folgt definiert werden:

αBUFFER= {in.x,out.x}, wobei $x \in$ Values
BUFFER= in?x -> out!x -> BUFFER

Das Kanal/ Nachrichten- Tupel beschreibt ein einzelnes Ereignis, wobei der Wert von x in out!x an das Vorkommen von x in in?x gebunden ist. Der Puffer kann mit allen anderen parallel laufenden Prozessen kommunizieren, die in!y oder out?y in ihrem Alphabet besitzen.
Ein Prozeß, der Daten in den Puffer schreibt, kann nun folgendermaßen aussehen:

αPRODUCER= {produce.p,in.p} , wobei $p \in$ Values
PRODUCER= produce?p -> in!p -> PRODUCER

Ein zugehöriger Empfänger kann folgendermaßen definiert werden:

αRECEIVER= {out.p} , wobei $p \in$ Values
RECEIVER= out?p -> RECEIVER

2.3.1.6 Beispiel: Die speisenden Philosophen in CSP

Ein oftmals in der Literatur angeführtes Beispiel für Nebenläufigkeit ist das Problem der speisenden Philosophen (vgl. [Dijk65b]). Hoare beschreibt dieses Problem in [Hoare85] anhand von 5 Philosophen $PHIL_0$, $PHIL_1$,..., $PHIL_5$ in CSP. Diese 5 Philosophen befinden sich in einer Philosophenschule, die über einen Speisesaal mit 5 Stühlen und einem

Tisch verfügt. Zu jedem Stuhl gehört eine Gabel. Um zu Essen braucht ein Philosoph jeweils eine Gabel zu seiner Rechten und zu seiner Linken. Die meiste Zeit seines Darseins verbringt der Philosoph mit Denken. Falls er jedoch hungrig wird, geht er in den Speisesaal, setzt sich auf seinen Stuhl, nimmt die Gabeln zu seiner Linken und seiner Rechten und beginnt zu essen. Nach Einnahme der Mahlzeit legt er die beiden von ihm benutzten Gabeln wieder zurück, steht auf, verläßt den Speisesaal und beginnt von neuem zu denken.
Die Menge der auszuführenden relevanten Ereignisse eines Philosophen ist demnach definiert als:

```
αPHILi= {   i.sitsdown, i.getsup,
            i.picksupfork.i, i.picksupfork.(i ⊕ 1),
            i.putsdownfork.i, i.putsdownfork.(i ⊕ 1) }
```

Hierbei ist $x \oplus y$ als modulo 5 Addition definiert, wodurch durch $i \oplus 1$ die Gabel identifiziert wird, die zur Rechten des Philosophen i gehört.
Das Alphabet für die Gabeln ist definiert als

```
αFORKi= {   i.picksupfork.i, (i ⊗ 1).picksupfork.i,
            i.putsdownfork.i, (i ⊗ 1).putsdownfork.i }
```

Hierbei ist $x \otimes y$ wiederum als modulo 5 Subtraktion definiert, wodurch $i \otimes 1$ den Philosophen zur Linken der Gabel i identifiziert.
Hierdurch benötigt jedes Ereignis (ausgenommen sitdsdown und getsup) die Teilnahme eines Philosophen und einer Gabel. Diese Abhängigkeit wird durch folgendes Diagramm verdeutlicht.

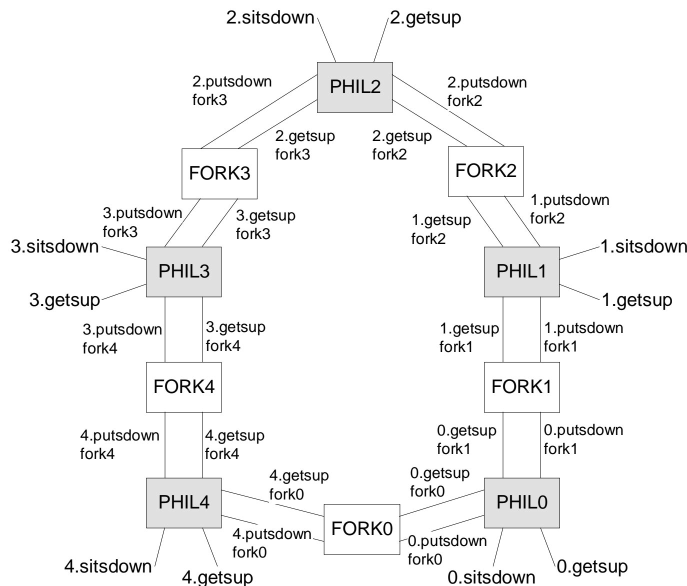

Abbildung 2-11: Verbindungsdiagramm der speisenden Philosophen

Das Verhalten eines Philosophens kann demnach als unendliche Folge von sechs Ereignissen definiert werden.

```
PHILi=  i.sitsdown ->
        i.picksupfork.i -> i.picksupfork.(i ⊕ 1) ->
        i.putsdownfork.i, i.putsdownfork.(i ⊕ 1) ->
        i.getsup -> PHILi
```

Hierbei setzt sich ein Philosoph hin, nimmt die linke und rechte Gabel auf, legt sie wieder hin und steht auf.

Für eine Gabel ergibt sich folgendes Verhalten:

```
FORKi=   i.picksupfork.i -> i.putsdownfork.i -> FORKi
         | i ⊗ 1 .picksupfork.i -> i ⊗ 1 .putsdownfork.i
              -> FORKi
```

Eine Gabel kann immer nur von dem Philosophen aufgenommen und anschließend abgelegt werden, dem sie gehört oder von dem Philosophen der links der Gabel sitzt.

Die gesamte Philosophenschule kann nun durch die nebenläufige Kombination des Verhaltens der einzelnen Philosophen und Gabeln definiert werden.

```
PHILOS=  PHIL0 || PHIL1 || PHIL2 || PHIL3 || PHIL4
FORKS=   FORK0 || FORK1 || FORK2 || FORK3 || FORK4
COLLEGE=PHILOS || FORKS
```

2.3.2 CSP Refinement

Als Verfeinerung eines Prozesses in CSP bezeichnet man einen anderen Prozeß, wenn dieser dasselbe nach außen sichtbare Verhalten an den Tag legt. Dieser Abschnitt beschreibt die einzelnen Definition von Verfeinerungen von Prozessen in CSP, wie sie in [Rosecoe97] ausgeführt sind.

2.3.2.1 Traces Refinement

Ein Prozeß *Q* ist ein *traces refinement* eines anderen Prozesses *P*, wenn alle möglichen Ereignissequenzen, die *Q* ausführen kann, auch von P ausführbar sind. Diese Beziehung wird als *Q* verfeinert *P* ($P[_T Q$) bezeichnet. Sei nun angenommen, daß *P* eine Spezifikation ist, die nur sichere Zustände eines Systems beschreibt, so bedeutet dies, daß *Q* eine sichere Implementation von *P* ist, wenn $P[_t Q$ gilt. Dies ist dadurch sichergestellt, daß keine falschen Ereignisse ausgeführt werden können.

$$P[_T Q \equiv traces(Q) \subseteq traces(P)$$

Traces Refinement wird hauptsächlich für die Überprüfung von Sicherheitseigenschaften benutzt, wobei sich jedoch keine Aussage über das tatsächliche Verhalten eines Prozesses machen läßt, da z.B. der Prozeß STOP, der niemals einen Event ausführt, immer eine gültige Verfeinerung jedes Prozesses ist, und somit jeder Sicherheitsspezifikation gerecht wird.

Als Beispiel für eine einfache Verfeinerung können folgende Prozesse P und Q dienen.

```
Q = enter -> exit -> Q
P = enter -> Phelp(0)
Phelp (i)=
  exit ->
  if(i<100) then enter ->Phelp(i+1)
  else P
```

Hierbei sehen P und Q äußerst unterschiedlich aus, wobei nicht direkt zu erkennen ist, daß beide tatsächlich die gleichen Spuren erzeugen.

2.3.2.2 Failures Refinement

Eine genauere Unterscheidung zwischen Prozessen kann dadurch bestimmt werden, daß in einer Spezifikation festgelegt wird, welche Ereignisse ein Prozeß ausführen, und welche er verweigern kann. Ein Mißerfolg (*Failure*) ist dabei ein Tupel (*s*,*X*), wobei s eine endliche Spur eines Prozesses *P* ist ($s \in traces(P)$) und *X* die Menge der Events, die verweigert werden können. *X* wird dabei als Rückweisungsmenge (*refusal set*) bezeichnet. Das Mißerfolgsrefinement ist dabei so definiert, daß alle Mißerfolge eines Prozesses *Q* auch in einer (sicheren) Spezifkation P enthalten sind.

$$P[_F Q \equiv failures(Q) \subseteq failures(P)$$

Wenn ein Prozeß gänzlich jede Ausführung von Ereignissen ablehnt, wird dieser Zustand als Verklemmung (Deadlock, siehe 'Verklemmung' auf Seite 14) bezeichnet. Der einfachste Prozeß, der eine Verklemmung herbeiführt ist STOP. Eine Verklemmung tritt in der Regel dann auf, wenn parallele Prozesse sich nicht auf einen Event synchronisieren können.

Failure Refinement wird hauptsächlich im Rahmen der Mißerfolges Divergenz Verifikation von Prozessen genutzt, von denen bereits bekannt ist, daß sie divergenzfrei sind.

2.3.2.3 Failures- Divergence Refinement

Das Mißerfolgsmodell erlaubt nicht, Livelocks (siehe 'Livelocks' auf Seite 15) innerhalb eines Systems zu erkennen. Hierbei beschreibt ein Livelock die Eigenschaft, daß ein Prozeß eine unendlich Sequenz interner Aktionen ausführt und so niemals ein nach außen sichtbares Ereignis ausführt. Um dies festzustellen wird ein Modell benötigt, daß eine höhere semantische Abstraktion beinhaltet.

Das Failures- Divergence Modell erfüllt diese Forderung, indem zusätzlich zum Failures Modell Abweichungen betrachtet werden. Die Abweichungen eines Prozesses ist die Menge der Spuren, nach denen ein Prozeß in einen Divergenzzustand (Livelockzustand) gelangen kann. Es müßte also eine Analyseform existieren, die feststellt ob ein System das Potential besitzt, in einen Zustand zu gelangen, aus dem er niemals mehr einen sichtbaren Event ausführen kann. Hierbei können in der Spezifikation ebenfalls Divergenzeigenschaften angegeben werden, um Situationen zu beschreiben, die für die Analyse außer acht gelassen werden können (don't care Eigenschaften). Die Relation des Failures- Divergence Refinement ($[_{FD}$) ist dabei wie folgt definiert.

$$P [_{FD} Q \equiv failures(Q) \subseteq failures(P) \wedge divergences(Q) \subseteq divergences(P)$$

Nach einer Divergenz verhält sich ein Prozeß chaotisch, was bedeutet, daß er jedes Ereignis ausführen oder verweigern kann. Dies bedeutet, daß das Verhalten zweier Prozesse als identisch anzusehen ist, wenn sie divergiert sind.

Failures- Divergence Refinement wird benutzt, um die Sicherheit, Lebendigkeit sowie die Kombination von beiden Eigenschaft für ein System nachzuweisen.

2.3.3 Der Model Checker FDR

FDR (Failures- Divergence- Refinement, vgl. [Formal97]) ist ein model- checking- Werkzeug für Zustandsmaschinen. Es basiert auf der im vorigen Abschnitt angerissenen Theorie konkurrierender Prozesse CSP ([Hoare85]).

FDR kann die im vorigen Abschnitt erläuterten Verfeinerungen überprüfen und ermöglicht ebenfalls, Zustandsmaschinen auf Determinismus zu untersuchen. Es wird aus diesen Gründen hauptsächlich zur Verifikation der Sicherheitseigenschaften von Systemen benutzt (vgl. [Roscoe95], [RWW94]).

2.3.3.1 Funktionsumfang von FDR

In den ersten Versionen von FDR wurden lediglich explizite model- checking Techniken unterstützt. Dies bedeutet, daß die Analyse als rekursive Induktion durchgeführt wurde, die den gesamten Zustandsraum zweier Systeme überprüft und damit jedes Paar von möglicherweise korrespondierenden Zuständen untersucht.

Die aktuell vorliegende Weiterentwicklung von FDR in der Version 2.28 verfügt im Vergleich dazu über eine höhere Flexibilität und erlaubt dabei unter anderem die Unterstützung von Operationen, die nicht Bestandteil von CSP sind und eine verbesserte Handhabung von Mehrwegsynchronisation. Hierbei werden Regeln definiert, die festlegen, welche Komponenten in welchen Zuständen Ereignisse auslösen dürfen, anstatt einen Vergleich auf deren konkrete Zustände durchzuführen. Des weiteren kann das System fortlaufend äquivalente Systeme erzeugen, die über kleinere Subsysteme mit kleineren Zustandsräumen verfügen und so die Überprüfung eines Systems beschleunigen.

Im Bezug auf die Repräsentation innerhalb von CSP verfügt FDR über alle drei Modelle, die zur Überprüfung des Verhaltens von Systemen herangezogen werden können.

2.3.3.2 Sprachkonstrukte von FDR

Die Sprachkonstrukte von FDR sind in [Formal95] und [Formal97] beschrieben und basieren hauptsächlich auf den in CSP verwendeten Konstrukten. Dieser Abschnitt beschreibt lediglich die wichtigsten und von dem hier zu entwickelnden Übersetzungssystem benutzten Sprachelemente, da eine komplette Einführung in die Syntax und Semantik der Sprachkonstrukte an dieser Stelle zu ausführlich wäre. Eine vollständige Syntaxreferenz befindet sich in [Formal97], die FDR Grammatik in Anhang D auf Seite 111.

Boolesche Ausdrücke

Als boolesche Konstanten stehen in FDR die Wahrheitswerte `true` und `false` zur Verfügung. Diese können über folgende logischen Operationen miteinander verknüpft werden:

Operation	Beschreibung
`not x`	Negierung eines booleschen Ausdrucks
`x and y`	logischer UND
`x or y`	logisches ODER

Abbildung 2-12: Boolsche Operationen in FDR

Numerische Ausdrücke

FDR unterstützt Ganzzahlarithmetik im Bereich von -2147483647 bis 2147483647. Die folgende Tabelle gibt einen Überblick über die verschiedenen Operationen auf numerischen Werten.

Operation	Beschreibung
`x+y`	Addition zweier Werte
`x-y`	Subtraktion zweier Werte
`x*y`	Multiplikation zweier Werte
`x/y`	ganzzahlige Division zweier Werte
`x%y`	Rest der Division zweier Werte
`-x`	Negierung eines Werts

Abbildung 2-13: Arithmetische Operationen in FDR

Des weiteren sind die folgenden Vergleichsoperationen definiert, die allesamt boolsche Werte als Ergebnis liefern.

Operation	Beschreibung	unterstützte Datentypen
`x==y`	Gleichheitsrelation	numerische Werte, boolesche Werte
`x!=y`	Ungleichheitsrelation	numerische Werte, boolesche Werte
`x<y`	x ist kleiner als y	numerische Werte
`x<=y`	x ist kleiner oder gleich y	numerische Werte
`x>y`	x ist größer als y	numerische Werte
`x>=y`	x ist größer oder gleich y	numerische Werte

Abbildung 2-14: Vergleichsoperationen in FDR

Typen

Typen sind in FDR grundsätzlich mit ihrer Elementmenge verbunden. Typangaben können zur Definition von Kanälen und rekursiv zur Festlegung anderer Typen verwendet werden.

Grundsätzlich kann zwischen benannten und unbenannten Typen unterschieden werden. Unbenannte Typen werden in der Regel als Menge angegeben, so daß die Typedefinition {0..3} äquivalent zu {0,1,2,3} ist.
Benannte Typen gelten lediglich als eine abkürzende Bezeichnung von unbenannten Typen. Jedes vorkommen eines Typnamens in einer FDR Spezifikation kann als durch die zugehörige Typdefinition ersetzt angesehen werden. So wird beispielsweise nach der Definition des Typen `Values= {0..3}` jedes vorkommen von `Values` durch den Ausdruck `{0..3}` ersetzt.

Benannte und unbenannte Typen können mehrdimensional definiert werden, so daß `{0..9}.{0..9}` ein zweidimensionales Feld mit 100 Elementen einführt.

Für Typen gilt in FDR strukturelle Typäquivalenz (vgl. [Watt96]), so daß die Nutzung identischer Elementmengen gleiche Typen einführt. Die nachfolgenden Definition bezeichnen deshalb den gleichen Typ:

`{0,1,2,3}` entspricht `Values= {0..3}`

Datentypen

Datentypen sind benannte Typen, die nicht parametrisiert werden können und nicht nur aus bisher eingeführten Typen, sondern auch aus Literalen bestehen können, die alternative Bestandteile der Datentypdefinition darstellen. Aus diesem Grund sind sie den Variant- Records der Programmiersprache Pascal (vgl. [Wirth95]) sehr ähnlich. Sie werden mit dem Schlüsselwort `datatype` eingeführt. Eine Datentypdefinition kann dazu dienen, eine Menge atomarer Konstanten zu definieren, die gegebenenfalls indiziert werden können. Hierzu werden in [Formal97] die folgenden Beispiele angeführt:

```
datatype SimpleColour = Red | Green | Blue
Gun = {0..15}
datatype ComplexColour =       RGB.Gun.Gun.Gun | Grey.Gun |
                               Black | White
```

Für Datentypen gilt ebenfalls strukturelle Typäquivalenz.

Einschränkungen von Typdefinitionen

FDR verfügt im Vergleich zu CSP über eine für unsere Zwecke gravierende Einschränkung: Alle Typdefinitionen, die zur Kommunikation eingesetzt werden, müssen als simpler Produkttyp ausdrückbar sein (*rectangular datatype*, vgl. [Formal97]). So ist es beispielsweise nicht möglich den folgenden Typ zu definieren, und ihn als Protokoll eines Kommunikationskanals zu verwenden:

```
datatype BrokenComplexColour =
                RGB.{r.g.b | r<-Gun, g<-Gun, b<-Gun, r+g+b <128 }
```

Prozeßdefintionen

Dieser Abschnitt stellt die elementaren Elemente zur Zusammenstellung von Prozessen vor. Die folgende Tabelle zeigt die einzelnen Strukturierungsausdrücke, aus denen sich ein Prozeß modellieren läßt:

Ausdruck	Beschreibung
`a->x`	Nachdem das Ereignis *a* stattgefunden hat, wird x ausgeführt. Der Prozeß muß am Ereignis *a* partizipieren, so daß die Ausführung von a nicht verweigert werden kann.
`P;x`	Nachdem der Prozeß *P* erfolgreich beendet wurde, wird *x* ausgeführt.
`P=Q`	Definition eines Prozesses *P* der äquivalent zu Prozeß *Q* ist. Durch dieses vorgehen lassen sich z.B. auch rekursive Prozesse definieren.
`P|~|Q`	Definition einer intern gesteuerten Alternative, bei der nichtdeterministisch entweder *P* oder *Q* ausgewählt wird.
`P[]Q`	Definition einer extern gesteuerten Alternative, bei der je nach Status der Umgebung entweder *P* oder *Q* ausgewählt wird.
`if b then x1 else x2`	Bedingte Verzweigung, wobei *b* ein boolescher Ausdruck sowie *x1* und *x2* vom gleichen Typ sein müssen.

Abbildung 2-15: Strukturierungsausdrücke in FDR

Wie bereits in Abschnitt 2.3.1.1 (Seite 26) erläutert, existieren in CSP vordefinierte Prozesse mit festen Eigenschaften. Diese sind ebenfalls in FDR verfügbar, wie die folgende Tabelle zeigt.

Ausdruck	Beschreibung
`STOP`	repräsentiert das verklemmte Prozeßverhalten, der keine Ausführung von Ereignissen zuläßt
`SKIP`	repräsentiert den erfolgreich abgeschlossenen Prozeß
`CHAOS(A)`	repräsentiert den Prozeß, der das chaotische Verhalten auf die Elemente des Alphabets A modelliert

Abbildung 2-16: Vordefinierte Prozesse in FDR

Bezüglich der Kommunikation zwischen einzelnen sequentiellen Prozessen untereinander ergibt sich analog zur Kommunikation in CSP (siehe auch 'Kommunikation in CSP' auf Seite 28) folgende Liste der unterstützten Operationen:

Ausdruck	Beschreibung
`chan!exp`	Auswertung des Ausdrucks exp in einen Ergebniswert v und Ausführung des Ereignisses chan.v.
`chan.i!exp`	Auswertung des Ausdrucks exp in einen Ergebniswert v und Ausführung des Ereignisses chan.i.v, wenn i ein Index des Kanalfeldes chan ist.
`chan?atom`	Auslesen des Wertes in chan und Zuweisung des Ergebnisses in die atomare CSP Variable atom.
`chan.i!atom`	Auslesen des Wertes im Kanalfeld chan an der Stelle des Index´ i und Zuweisung des Ergebnisses in die atomare CSP Variable atom.

Abbildung 2-17: Kommunikation in FDR

Zur Definition von Nebenläufigkeitseigenschaften eines Systems stehen die nachfolgend aufgeführten Definitionen zur Verfügung.

Ausdruck	Beschreibung
`P|||Q`	Definition einer unsynchronisierten, parallelen Ausführung von P und Q (interleaving).
`P[|X|]Q`	Definition der synchronisierten Komposition zweier Prozesse. P und Q müssen sich dabei über die Ereignismenge X synchronisieren. Ereignisse, die nicht in X liegen gelten als unabhängig.
`P\A`	Die Menge A legt eine Teilmenge des Alphabets des Prozesses P fest, die bei der Ausführung von P nach außen nicht sichtbar sind (hiding).

Abbildung 2-18: Nebenläufigkeitsdefinition in FDR

Lokale Definitionen

Lokale Definition können innerhalb von Prozessen mit Hilfe der `let ... within` Klausel realisiert werden. Allerdings ist es in der vorliegenden Version von FDR nicht möglich, Kanal- und Datentypdefinitionen lokal zu definieren.

```
P=
  let
    loop(i,to)=     if (i<=to) then loop(i+1,to)
                    else SKIP
  within loop(0,3); loop(4,10)
```

Abbildung 2-19: Lokale Definitionen in FDR

Hierbei werden die lokalen Definition in Form von Prozeßdefinitionen hinter dem Schlüsselwort `let`, und der Rumpf des Prozesses, der die lokalen Definition beinhaltet, hinter dem Schlüsselwort `within` angegeben.

Variablenbindung

Bezeichner werden als Werte behandelt, sofern sie nicht in einem Kontext erscheinen, in dem sie an Variablen gebunden sind. Variablenbindungen werden in folgenden Kontexten eingeführt (vgl. [Formal97]):

- **Explizite Deklarationen auf höchster Ebene:** Nach der Definition `Z= {1,2,3}` wird der Bezeichner Z in allen anderen Definitionen als Variable behandelt, die mit der Menge {1.2.3} verknüpft ist.
- **Prozeßparameter, Parameter lokaler Definitionen:** Auf der rechten Seite einer Prozeßdeklaration oder lokalen Definition `P(x,y,z)= ...` sind die Bezeichner `x`, `y` und `z` als Parameter an P gebunden.
- **Eingabeereignisse:** Im Prozeß `P=c?x -> P` ist x an den Prozeß P gebunden. Der damit verbundene Wert ist der Wert im Kanal `c`.

2.3.3.3 Eine Bespielverifikation mit FDR

In diesem Abschnitt soll die CSP Abstraktion für das Problem der speisenden Philosophen (siehe ’Beispiel: Die speisenden Philosophen in CSP’ auf Seite 28) mit dem FDR Werkzeug auf seine Eigenschaften bezüglich Verklemmung und Livelock analysiert werden.

Entwicklung einer FDR Abstraktion

Die CSP Spezifikation kann bis auf einige kleinere Änderungen direkt übernommen werden. Da es keine direkt indizierbaren Events in FDR gibt, werden alle in der CSP Spezifikation benutzten Ereignisse in Kanäle übersetzt, die durch eine angegebene (mehrdimensionale) Menge von Indizes angesprochen werden können. Außerdem wurde die in der CSP Abstraktion angegebenen abstrakten modulo- Operationen in die Anwendung konkreter FDR Operationen umgesetzt.
Eine Erweiterung ist bei der Spezifikation der gesamten Philosophenschule notwendig. Die Philosophen (`PHILOS`) und Gabeln (`FORKS`) synchronisieren sich bekanntlich über

die beiden Ereignisse `picksupfork` und `putsdownfork`, was in der Spezifikation explizit über die Menge der Synchronisationsereignisse (`[|{|picksupfork,putsdownfork|}|]`) anzugeben ist.
Des weiteren können in FDR- Abstraktionen explizit zu überprüfende Anforderungen an ein System über die `assert` Klausel angegeben werden. Dies wurde in diesem Fall zur Festlegung der auszuführenden Deadlock- und Livelockprüfung angewendet.

```
-- Problem der dinierenden Philosophen in FDR
-- MHE, 20/06/99

-- Anzahl der Philosophen
philNum=5

-- IDs der Philosphen und Gabeln
philIDs={0..(philNum-1)}
forkIDs={0..(philNum-1)}

-- Kanäle, die sich auf die Philosophen beziehen
channel sitsdown: philIDs
channel getsup: philIDs

-- Kanäle, die sich auf Philosophen und Gabeln beziehen
channel picksupfork: philIDs.forkIDs
channel putsdownfork: philIDs.forkIDs

-- Prozess der einen Philosophen modelliert
PHIL(i)= sitsdown.i ->
    picksupfork.i.i ->
    picksupfork.i.(i+1)%philNum ->
    putsdownfork.i.i ->
    putsdownfork.i.(i+1)%philNum ->
    getsup.i ->
    PHIL(i)

-- Prozess der eine Gabel modelliert
FORK(i)= picksupfork.i.i ->
    putsdownfork.i.i ->
    FORK(i)
    |~|
    picksupfork.(i-1)%philNum.i ->
    putsdownfork.(i-1)%philNum.i ->
    FORK(i)
-- Modellierung aller Philosophen
PHILOS=     PHIL(0) ||| PHIL(1) ||| PHIL(2) |||
            PHIL(3) ||| PHIL(4)
-- Modellierung aller Gabeln
FORKS=      FORK(0) ||| FORK(1) ||| FORK(2) |||
            FORK(3) ||| FORK(4)
-- Modellierung der gesamten Philosophenschule
COLLEGE= (     PHILOS
               [|{|picksupfork,putsdownfork|}|]
               FORKS)

-- Definition der zu verifizierenden Eigenschaften
assert COLLEGE :[deadlock free [F]]
assert COLLEGE :[livelock free]
```

Abbildung 2-20: Problem der speisenden Philosophen in FDR

Deadlock- und Livelockanalyse

Nach dem Laden und Ausführen der Überprüfung auf Verklemmungsfreiheit liefert die Oberfläche des FDR Werkzeugs die folgende Ausgabe:

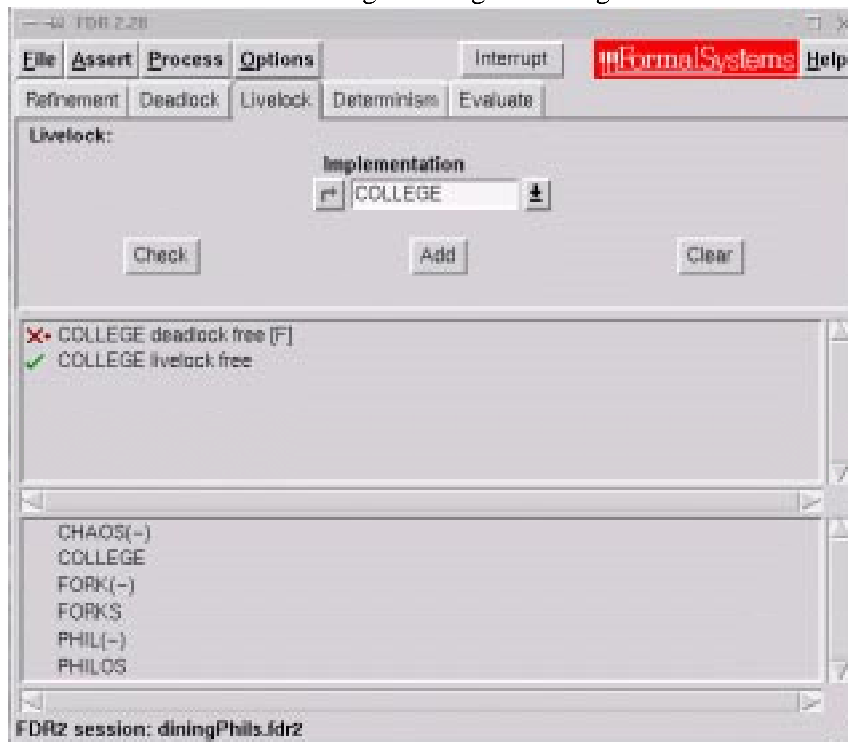

Abbildung 2-21: FDR Session: Problem der speisenden Philosophen

Die Philosophenschule ist offensichtlich nicht deadlockfrei, was sich einfach dadurch nachvollziehen läßt, indem man den Fall betrachtet, daß alle Philosophen sich gleichzeitig setzen und ihre linke Gabel aufnehmen. Danach ist keine rechte Gabel mehr frei, die sich ein Philosoph nehmen könnte und das System blockiert.
Des weiteren haben wir mit Hilfe von FDR nachgewiesen, daß das System nicht über die Livelockeigenschaft verfügt.

2.4 Verifizierbares Verhalten von Threads

In diesem Abschnitt wird dargestellt, welches Verhalten von Threads sinnvoll überwacht werden kann. Im wesentlichen werden hier die im Abschnitt “Die Java Thread- API” auf Seite 16 dargestellten Technologien zur Threadsteuerung auf ihre Verwendbarkeit innerhalb einer Verhaltensanalyse von Programmen mit mehreren Kontrollfäden untersucht und eine Modellierung des gewünschten Verhaltens in CSP vorgestellt. Die einzelnen hier entwickelten Spezifikationen sind als komplettes Paket in Anhang C auf Seite 105 aufgeführt.

2.4.1 Threadaufbau und -abbau

Das Starten und Stoppen von Threads wird über die Methoden `java.lang.Thread.start`, `java.lang.Thread.stop`, `java.lang.Thread.destroy` und `java.lang.Thread.join` (siehe auch ’Threadaufbau und -abbau’ auf Seite 18) bewerkstelligt. Hierbei stellt sich die Frage, welche Beziehungen die einzelnen Methoden zueinander haben und in welcher Reihenfolge sie sinnvoll ausgeführt werden können. So macht es zum Beispiel wenig Sinn, das Verhalten eines Threads beobachten zu wollen, bevor er gestartet wurde. Bevor also überhaupt eine threadsteuernde oder -synchronisierende Methode auf einem Thread ausgeführt wird, muß er zunächst gestartet sein. Aus diesem Gründen ergibt sich folgendes Verhalten:

```
Thread= threadStart -> (     threadStop
                        []   threadDestroy
                        []   threadJoin )
```

Hierbei haben wir bisher die Semantik von `join` nicht betrachtet. Da `join` auf die Beendigung eines Threads wartet, kann diese Methode nur sinnvoll nach der Beendigung eines Kontrollfadens eingesetzt werden.

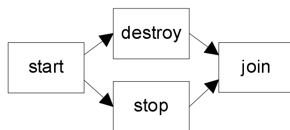

Abbildung 2-22: Angestrebtes Start- und Stoppverhalten von Threads

```
Thread= threadStart ->  (  threadStop
                           []
                           threadDestroy
                           []
                           (threadStop-> threadJoin)
                           []
                           (threadDestroy-> threadJoin) )
```

Hierbei ist zu beachten, daß der Fall des Neustarts eines Threads nicht beachtet wird. Ein Kontrollfaden kann mittels `stop` oder `destroy` beendet und dann mittels `start` erneut gestartet werden. Dieses Verhalten ist zwar in Java ausdrückbar, läßt aber auf einen schlechtes Systemdesign schließen. Die einschlägige Literatur (vgl. [Lea97] und [OaWo99]) rät ausdrücklich vom Thread- Neustart ab und empfiehlt stattdessen die Methoden `suspend` und `resume` (siehe ’Threadsteuerung’ auf Seite 18) zu verwenden, um den entsprechen Thread lediglich von der Ausführung zu suspendieren, nicht aber zu beenden.

2.4.2 Threadsuspendierung

Die Suspendierung und Weiterausführung eines Threads macht nur dann Sinn, wenn der zu suspendierende oder weiterzuführende Thread vorher gestartet wurde. Auf einem Thread darf jedoch `resume` ausgeführt werden, bevor er mittels `suspend` suspendiert wurde. Hieraus ergibt sich:

```
threadStart -> ( threadSuspend -> threadResume
                 []
                 threadResume)
```

Suspend und resume dürfen beliebig oft nach dem Start und vor dem Beenden eines Threads ausgeführt werden. Dies verdeutlicht folgendes Diagramm:

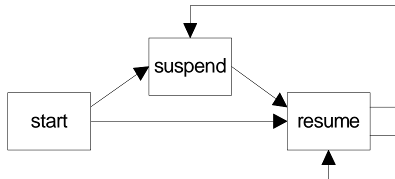

Abbildung 2-23: Angestrebtes Suspendierungsverhalten von Threads

Für dieses Verhalten ergibt sich folgende Spezifikation:

```
pause=          threadSuspend -> threadResume -> pause
                []
                threadResume -> pause
                []
                SKIP
SYS = threadStart -> pause
```

2.4.3 Monitorverhalten

Synchronisierte Blöcke werden, wie bereits in “Verwendung des synchronized Schlüsselworts” auf Seite 21 beschrieben, in einen Monitor engebettet. Das Eintreten in den Monitor wird durch den Java Bytecode Befehl `monitorenter` und das Austreten durch `monitorexit` initiiert. Da sich zu einem Zeitpunkt nur jeweils ein Thread in einem Monitor befinden darf, muß das Eintreten und Austreten in einer festen Reihenfolge erfolgen.

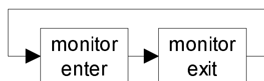

Abbildung 2-24: Angestrebtes Monitorverhalten

Die Ereignisse `monitorenter` und `monitorexit` können beliebig oft nacheinander ausgeführt werden, so daß sich hierfür folgende Prozeßspezifikation ergibt:

```
Mon= monitorenter -> monitorexit -> Mon
```

2.4.4 Synchronisationsverhalten

Die Synchronisation der einzelnen Threads untereinander wird, wie in Abschnitt “Warten und Benachrichtigen” auf Seite 21 beschrieben, mit den Methoden `wait`, `notify` und `notifyAll` bewerkstelligt. Die Methode `wait` wartet entweder auf ein `notify` oder ein `notifyAll`, das alle wartenden Threads in einen lauffähigen Zustand bringt. Die Ereignisse `notify` bzw. `notifyAll` dürfen jedoch auch ohne ein voriges `wait` ausgeführt werden. Dies führt zu folgendem angestrebten Synchronisationsverhalten:

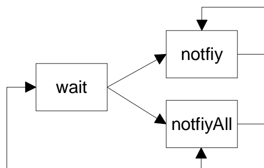

Abbildung 2-25: Angestrebtes Synchronisationsverhalten von Threads

Das Verhalten von `wait` und `notify` kann fortlaufend ausgeführt werden, so daß sich folgender rekursiver Prozeß ergibt:

```
notifyBeh=       notify -> (notifyBeh [] waitNotify)
                 []
                 notifyAll -> (notifyBeh [] waitNotify)
waitNotify=      wait -> notifyBeh
                 []
                 notifyBeh
```

2.4.5 Ausnahmen (Exceptions)

Das Werfen einer Ausnahme während der Programmausführung innerhalb eines Java Bytecode Interpreters stellt ein Verhalten dar, das nicht direkt in CSP/FDR nachgebildet werden kann. Aus diesem Grund wird das Auftreten einer Ausnahme durch den `STOP` Prozeß modelliert, der kein Ereignis mehr ausführt (siehe ’Prozesse, Alphabete und Ereignisse in CSP’ auf Seite 26):

```
Exception= STOP
```

2.5 Auswertung paralleler Java Programme

Für die Auswertung von parallelen Java Programmen ist zunächst einmal interessant, welche Operationen parallel ablaufen können. Innerhalb eines Java Programms sind dies die einzelnen erzeugten Threads und der Hauptkontrollfaden (*Main- Thread*). Außerdem muß abgebildet werden, ob es Bedingungen gibt, die zur Erzeugung neuer Kontrollfäden führen.

Im Bezug auf die Synchronisation der einzelnen Threads untereinander muß ebenfalls ermittelt werden, welche Synchronisationsoperationen ausgeführt werden, und welche Bedingungen es hierfür gibt.

2.5.1 Ermittlung von Kontrollfäden

Um die im Abschnitt “Verifizierbares Verhalten von Threads” auf Seite 38 dargestellten gewünschten Verhaltensmuster analysieren zu können, muß man zunächst einmal wissen, welche Kontrollfäden innerhalb eines Programms überhaupt existieren.

Starten von Threads im Hauptkontrollfaden

Der Hauptkontrollfaden wird innerhalb eines Java Programms immer durch die Methode `main()` beschrieben, die uns für alle Betrachtungen als Einstiegspunkt dient. Weitere Threads können zunächst nur in der Main- Methode gestartet werden. Es ist also sinnvoll, zu untersuchen, ob direkt in `main` oder innerhalb der Methoden, die `main` aufruft Threads gestartet werden. Ein zuverlässiger Indikator hierfür ist die Methode `java/lang/Thread/start`, die immer dann aufgerufen wird, wenn ein zuvor erzeugter

Kontrollfaden seine Arbeit aufnehmen soll.
Als Ergebnis dieser Betrachtungen erhalten wir den Hauptkontrollfaden und alle von ihm gestarteten Threads.

Starten von Threads in Threads

Kontrollfäden können an jeder Stelle eines Programms, also auch innerhalb laufender Threads, gestartet werden. Dies führt dazu, daß eine Betrachtung der im Hauptkontrollfaden angestoßenen Threads nötig ist, um gänzlich alle Kontrollfäden zu ermitteln. Der Vorgang hierzu ist analog zu dem im Hauptkontrollfaden. Sollte hierdurch die Existenz neuer Threads zu Tage gefördert worden sein, wiederholt sich dieser Vorgang rekursiv.

Folgendes Diagramm veranschaulicht die Ermittlung von Threadinstanzen:

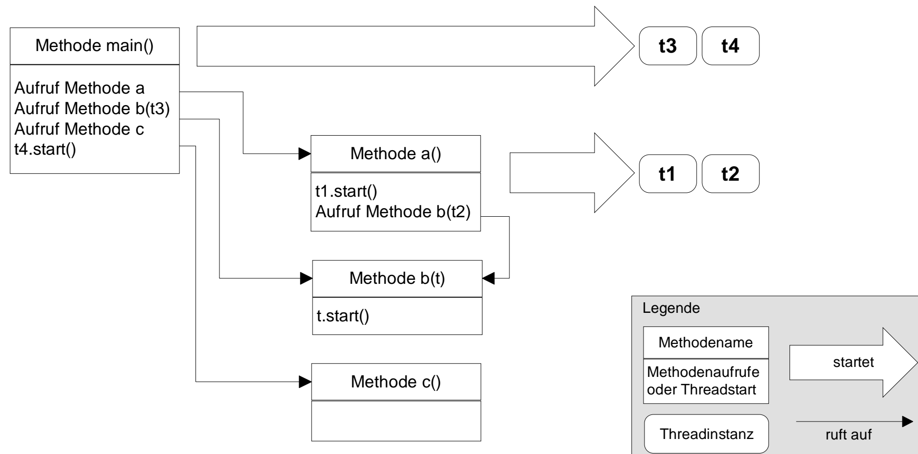

Abbildung 2-26: Ermittlung von Threadinstanzen

Die dargestellte Methode `main` ruft die Methoden `a`, `b` sowie `c` auf und startet den Thread `t4` direkt. Die aufgerufene Methode `b` startet den Thread, den sie als Parameter übergeben bekommt. Hieraus resultiert, daß `main` die Threads `t3` und `t4` startet. Die Methode `a` startet ihrerseits den Thread `t1` direkt, sowie den Thread `t2` über den Aufruf von `b`. Hieraus resultiert, daß in dem vorliegenden Beispiel die Kontrollfäden `t1`, `t2`, `t3` und `t4` angestoßen werden.

2.5.2 Ermittlung benötigter Synchronisationsvariablen

Nachdem bekannt ist, welche Kontrollfäden innerhalb eines Programms vorhanden sind, ist es nun von Interesse zu Wissen, auf welche Objekte sich die einzelnen Threads synchronisieren. Analog zu Abschnitt “Verifizierbares Verhalten von Threads” auf Seite 38 sind dies die Objekte, auf die die Monitorfunktionen und `Wait`- bzw. `Notify`- Funktionen aufgerufen werden.
Betrachtet man nun alle Variablen des zu untersuchenden Programms auf die die Bytecodeoperationen `MONITENTER` und `MONITOREXIT` sowie die Methoden `wait`, `notify` und `notifyAll` aufgerufen werden, entsteht eine Zusammenstellung aller Synchronisationsvariablen.
Auch hierbei kann es (wie bei der Ermittlung der Kontrollfäden) dazu kommen, daß die betrachteten Operationen und Methoden auf Parameter weiterer Methoden aufgerufen werden. In diesem Fall muß überprüft werden, ob es sich bei der Variable, die der aufrufenden Methode übergeben wird, ebenfalls um einen Parameter handelt. Ist dies nicht der Fall, wird die Variable in die Liste der Synchronisationsvariablen eingetragen, andernfalls wird rekursiv weitergesucht.
Als Resultat liegt eine Liste aller Synchronisationsvariablen des Systems vor.

2.5.3 Ermittlung benötigter Strukturvariablen

Der Ablauf eines Programms ist in der Regel nicht linear. Es existieren in den meisten Fällen eine Reihe von bedingten Sprüngen, deren Bedingungen für den korrekten Ablauf eines Programms unabdingbar sind. Folgendes Java- Code- Fragment soll hierfür als Beispiel dienen:

```
public void run()  {
   for (;;) {
      if(count>5) obj.wait();
      else count++;
   }
}
```

Abbildung 2-27: Beispiel für Strukturvariablen

In diesem Beispiel existiert eine Variable `obj`, auf die die Methode `wait` aufgerufen wird. Dieser Aufruf ist jedoch an die Bedingung geknüpft, daß der Zähler `count` den Wert 5 überschreitet. Für eine Betrachtung des Programmablaufs im Hinblick auf das Synchronisationsverhalten sind somit auch alle Variablen interessant, die in einer Bedingung für Synchronisationsmethoden benutzt werden.

Diese Betrachtung läßt sich weiter auf die Steuerung von Kontrollfäden ausweiten, da auch die Aufrufe von Steuerungsfunktionen (`start`, `stop`, `suspend`, `resume`, etc.) an Bedingungen geknüpft sein können.

Als Strukturvariablen gelten also die Variablen, die den Aufruf von Methoden auf Synchronisationsvariablen in irgend einer Art und Weise beeinflussen.

2.5.4 Ermittlung benötigter Algorithmen

Nicht alle Algorithmen, die innerhalb eines Programms definiert wurden, beeinflussen die Threadsteuerung oder Synchronisation und sind für die Analyse interessant. Jeder sequentielle Algorithmus, dessen Ergebnis das Threadverhalten nicht beeinflußt, kann demnach für die Analyse unbetrachtet bleiben (vgl. [BKPS97]).

Als Thread- beeinflussende Algorithmen können in unserem Kontext alle Teile von Methoden angesehen werden, die keine Synchronisations- oder Strukturvariablen enthalten. Alle Methoden oder Methodenteile, die keine Synchronisations- oder Strukturvariablen enthalten, können somit unbetrachtet bleiben.

Betrachten wir eine abgeänderte Version unseres Beispiels aus Abschnitt 2.5.3:

```
public int f(int i) {
   return i*i+1;
}
public void run()  {
   for (;;) {
      if(count>5) {
         obj.wait();
      } else {
         i=f(count);
         a++;
         count=f(a);
      }
   }
}
```

Abbildung 2-28: Beispiel für die Ermittlung benötigter Algorithmen

Die zusätzlich eingeführte Variable `i` hat keinen Einfluß auf das Synchronisationsverhalten, da sie in keiner Beziehung zu einer Synchronisationsvariable (in diesem Fall `obj`) steht. Sie hängt zwar von der bereits als Strukturvariable erkannten Variable `count` ab, wird jedoch für die Berechnung von Werten einer Strukturvariable oder innerhalb einer Bedingung für Synchronisationsoperationen nicht benutzt.

Anders verhält sich dies für die Variable `a`. Sie wird dazu benutzt die Strukturvariable

count zu berechnen und ist deshalb auch als Strukturvariable anzusehen. Zusätzlich wird die Methode f zur Berechnung von count aufgerufen, womit sie einen benötigten Algorithmus darstellt und für die Betrachtung der Synchronisationseigenschaften interessant ist.

Dies führt dazu, daß i für die Analyse nicht betrachtet werden muß, count sowie a Strukturvariablen sind und obj eine Synchronisationsvariable ist. Die Methode f befindet sich innerhalb eines strukturell benötigten Algorithmus' und muß somit für die Analyse betrachtet werden.

2.6 Abstraktionsmodell

In diesem Abschnitt wird ein Abstraktionsmodell entwickelt, das darauf zugeschnitten ist, Java- Programme auf CSP/ FDR- Spezifikationen abzubilden. Java ist eine Programmiersprache mit mächtigen Sprachkonstrukten, während der CSP- Dialekt von FDR lediglich der Spezifikation von Systemen dient. Dies hat zur Folge, daß viele Elemente der Programmiersprache Java in aufwendige CSP/ FDR- Konstrukte übersetzt werden müssen.

Die hier verwendete Beschreibung orientiert sich absichtlich an den Quellcode Konstrukten von Java und CSP/FDR, da hierdurch das Abstraktionsmodell und die daraus resultierende Transformation intuitiver Verständlich ist, als beispielsweise auf Ebene der abstrakten Syntax.

2.6.1 Nachbildung des Java- Objektmodells

Eines der aufwendigsten Aufgaben ist es, das Java Objektmodell in CSP/FDR Spezifikationen zu übertragen. Im Java Laufzeitsystem enthält jedes Objekt eine eindeutige ID (Objektidentität, vgl. [GoJoSt96]).

Jedes Objekt besitzt standardmäßig die Objekt- ID null, die als Indikator dafür dient, daß das Objekt nicht initialisiert ist. Nach dem Aufruf von new wird dem Objekt eine eindeutige ID zugeordnet.

Nachbildung von Java Referenzvariablen

In CSP/FDR ist es nur sehr schwer möglich, mit vertretbarem Aufwand Objekte zu simulieren, da nur ganzzahlige und boolesche Datentypen existieren. Aus diesem Grund wird die Objektidentität über numerische Referenzen in Variablenkonstrukte verlagert.

Jede Variable, die von mehreren Objekten instanziert wird, wird durch deren Objekt- IDs indiziert. Im folgenden Beispiel existieren die Klassen a und b, wobei a die Attribute a1 und a2 besitzt.

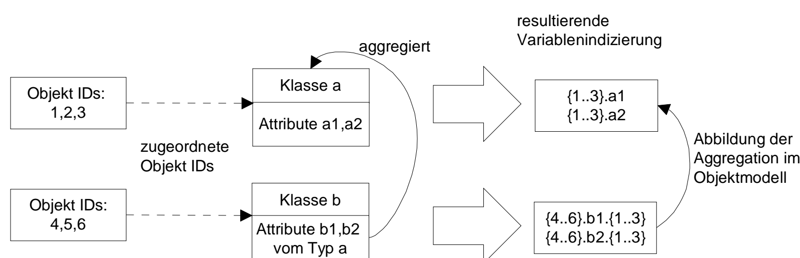

Abbildung 2-29: Beispiel für die Abbildung des Java Objektmodells

Die Klasse b besitzt die Attribute b1 und b2 vom Typ a. Der Klasse a wurden die Objekt- IDs 1 bis 3 zugeordnet, der Klasse b die Objekt- IDs 4 bis 6. Die Attribute der Klasse a lassen sich somit mit allen Objekten der Klasse a indizieren ({1..3}.a1, {1..3}.a2).

Die Klasse b hingegen kann über ihre Attribute b1 und b2 auf die Objekte vom Typ a zugreifen. Dies wird dadurch realisiert, daß alle Attribute, mit dem Objektwerten der Klasse a belegt werden können (in unserem Beispiel {1..3}). Hieraus ergibt sich, daß die Variablenkonstrukte {4..6}.b1 und {4..6}.b2 auf alle Objekte der Klasse a zurückgreifen können. Dies entspricht der Semantik von Referenztyp-Variablen (vgl. [GoJoSt96]).

Nachbildung eines Objekt- ID- Generators

Objekt- IDs werden zur Laufzeit dynamisch vergeben, so daß man nicht darauf verzichten kann, einen Objekt ID- Generator für CSP/FDR Spezifikationen zu entwickeln, der jedem zu initialiserenden Objekt eine eindeutige ID zuordnet.
Wenn man nach dem oben beschriebenen Prinzip vorgeht, trifft man dabei schnell auf das Problem, daß die Teilmengen der Objekt- IDs an Typen gebunden sind. Würden die Objekt- IDs einfach der Reihe nach vergeben, wird die Zuordnung der instanziierbaren Objekte an Variablen sehr komplex.

Aus diesem Grund ist es sinnvoll, für jeden Typ innerhalb einer Spezifikation einen eigenen Objekt- ID- Generator zu erzeugen. Unangenehmerweise gibt es in Java einen direkten Vergleich auf Objektidentität. Würden Objektreferenzen verschiedener Typen nun von 0 bis n durchnumeriert werden, könnte das Problem auftreten, daß zwei Objekte unterschiedlichen Typs als identisch erkannt werden würden.

Aus dieser Tatsache resultiert die folgende Mischform eines Objekt ID Generators in CSP/FDR:

```
channel readNewObjectID: CLASSES.{0..MAX_OBJ}
NewObjectIDGenerator(class,val,max) =
   val<=max & readNewObjectID.class!val ->
   NewObjectIDGenerator(class,val+1,max)
```

Die Funktion `NewObjectIDGenerator` geht dabei so vor, daß sie von einem Minimalwert, der bei der Initialisierung angeben wird, bis zu einem definierten maximalen Wert, neue Objekt- IDs in den Kanal `readNewObjectID` schreibt. Dies hat den Vorteil, daß alle Objekte eindeutige Objekt- IDs in einem vordefinierten Bereich erhalten. Der Nachteil dabei ist die Tatsache, daß diese Bereiche statisch festgelegt werden müssen.

Folgende Abbildung zeigt die Klassen `a` und `b` inklusive einiger Instanzen samt zugewiesenen Objekt- IDs:

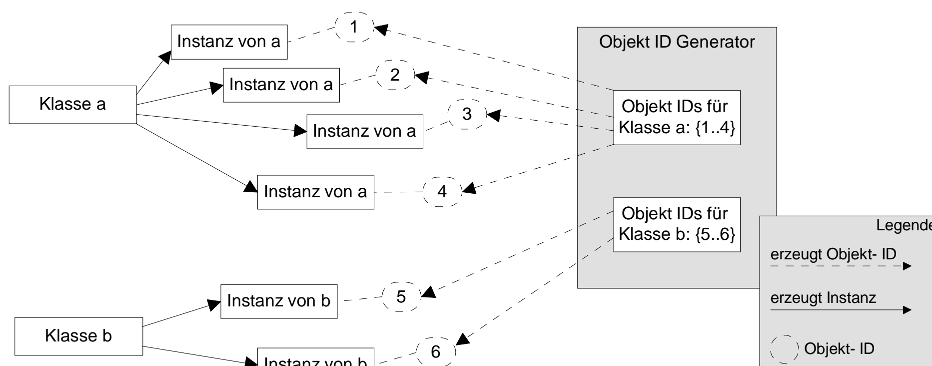

Abbildung 2-30: Verteilung von Objekt- IDs

2.6.2 Abbildung von Variablen auf Kanalprotokolle

Einer der sensibelsten Aspekte bei der Abstraktion von Java Programmen in CSP/FDR ist die Abbildung von Variablen auf Kanaldefinitionen und Kanalprotokolle, da Kanäle die Grundlage für die Kommunikation zwischen Prozessen sind.

Hierbei müssen folgende Eigenschaften der Programmiersprache Java berücksichtigt werden, um eine korrekte Abbildung von Variablenbereichen auf Kanalprotokolle zu gewährleisten:

- es existieren primitive Typen
- es existieren zusammengesetzte Typen (Objekttypen)
- es existieren (multidimensionale) Feldtypen

Desweiteren haben wir bereits herausgefunden, daß es für Kommunikationskanäle nicht möglich ist, Protokolle zu erstellen, deren Typen nicht als einfache Produkttypen definiert sind (siehe ’Sprachkonstrukte von FDR’ auf Seite 32).

Strukturvariablen

Da sich wie bereits erwähnt keine Kommunikationsprotokolle entwickeln lassen, die universell getypt sind, führen wir eine Unterscheidung der einzelnen Typen ein, die in folgender Aufzählung dargestellt ist:

- numerisches Kanalprotokoll
- boolesches Kanalprotokoll
- objektwertiges Kanalprotokoll

Die primitiven und Objektvariablen können mit diesen drei Kanälen gut abgebildet werden. Lediglich Feldtypen stellen hierfür ein Problem dar, da jede Felddimension einen weitere Indexmenge einführt. Aus diesem Grund unterstützen wir nur Felder mit einer Dimension, woraus die nachstehenden Feldkanalprotokolle folgen:

- eindimensionales numerisches Feld- Kanalprotokoll
- eindimensionales boolesches Feld- Kanalprotokoll
- eindimensionales objektwertiges Feld- Kanalprotokoll

Für alle diese Protokolle werden eigene Kanäle mit den Ereignissen `read` und `write` zum lesenden bzw. schreibenden Zugriff definiert.

Thread- und Synchronisationsvariablen

Thread und Synchronisationsvariablen werden nicht zur Kommunikation benutzt, weshalb es ausreichend ist, diese Arten von Variablen als einfache Kanalprotokolle abzubilden. So können alle Thread- und Synchronisationsvariablen als Instanz der Ereigniskanäle abgebildet werden.

Jeder Thread- oder Synchronisationsvariablen wird hierzu ein eindeutiger numerischer Wert zugewiesen, mit dem sich für Threadvariablen die Ereignisse `start`, `stop`, `destroy`, `join` sowie `resume` und `suspend` ansprechen lassen. Für Synchronisationsvariablen sind die entsprechenden Kanäle `wait`, `notify`, `notifyAll`, sowie `monitorenter` und `monitorexit` zu reservieren.

2.6.3 Abbildung von Java Methoden auf CSP Funktionen

Durch die Einführung von Variablenkonstrukten und der Nachbildung des Java Objektmodells stellt sich die Frage, wie sich diese im Zusammenspiel mit Methodenaufrufen und deren Abbildung in CSP/FDR integrieren lassen. Hierzu betrachten wir zunächst die Definition von Funktionen und anschließend den Mechanismus des Aufrufs und der Rückgabe von Werten.

Methoden in Java und Funktionen in CSP/FDR haben syntaktisch gesehen eine sehr ähnliche Form. Sie können eine Reihe von Parametern aufnehmen und einen Wert zurückgeben. Die Semantiken der beiden Konstrukte sind hingegen sehr unterschiedlich:

- Parameter primitiven Typs werden in Java Methoden *by- value* übergeben, alle anderen *by-reference*. Im Vergleich dazu, kann in CSP/FDR- Abstraktionen das Vorkommen eines Parameters im Rumpf einer Funktion als Ersetzung durch den übergebenen Parameter angesehen werden.
- Rückgabewerte können in Java Methoden primitiven oder zusammegesetzten Typs sein. In CSP/FDR können nur Werte primitiven Typs zurückgegeben werden.

2.6.3.1 Transformation von Methodendefinitionen

Die folgenden Transformationsregeln basieren direkt auf der unterschiedlichen Semantik von Java Methoden und CSP/FDR Funktionen.

- Alle primitiven Parameter in Java Methoden werden in CSP/FDR Funktionen *by- value* übergeben.
- Für alle primitiven Parameter wird in der CSP/FDR- Abstraktion ein Variablenkonstrukt erzeugt, wenn dem Parameter in der Java Methode ein Wert zugewiesen werden kann. Dies ist notwendig, da in Java- Programmen Zuweisungen an primitive Parameter vorgenommen werden können, was in CSP/FDR nicht zulässig ist.

- Alle nicht- primitiven Parameter in Java Methoden werden in CSP/FDR Funktionen *by-value* durch die ID der entsprechenden Variable übergeben, die als Parameter verwendet wird. Des weiteren wird die ID des Objekts, welches die Variable enthält als jeweils zweiter Parameter *by- value* übergeben.
- Ist die Java Methode nicht statisch, so ist sie an ein Objekt gebunden. Dies führt in CSP/FDR- Funktionen zur Einführung zweier Parameter (`this` und `objectID_this`), die die ID der entsprechenden Variable und die ID des Objekts, welches die Variable enthält als by- value übergebene Parameter erwarten.
- Ist die Java Methode mit einen Rückgabewert definiert worden, wird ein zusätzlicher Rückgabeparameter in der CSP/FDR Funktion eingeführt. Dieser Rückgabeparameter enthält eine by- value übergebene Variablen- ID, in der der Rückgabewert übertragen wird.

Als Beispiel hierfür dient die folgende nicht statische Java- Methodendefinition, die einen primitiven Parameter erwartet und einen primitiven Wert zurückliefert:

```
public int calc(int val)
```

Transformiert in eine CSP/FDR Funktion, werden zwei Parameter für die `this`- Variable, sowie jeweils einer für den primitiven Methodenparameter und den Rückgabewert eingeführt.
Der primitive Parameter wird innerhalb des Funktionsrumpfes in das Variablenkonstrukt `val` kopiert. Vor dem Rücksprung aus der Funktion, wird ein etwaiger Rückgabewert dem Variablenkonstrukt zugewiesen, das die im Parameter `return` übergebene Variablen- ID besitzt.

```
calc(this,objectID_this,param0,return) =
  val!param0 ;
  ... x berechnen ...
  return!x ->
  SKIP
```

2.6.3.2 Transformation von Methodenaufrufen

Die Transformation von Methodenaufrufen in CSP/FDR- Funktionsaufrufe ist eng an die Übersetzung der Methodendefinition geknüpft, da jeder Aufruf die in der Definition der aufzurufenden Methode angegeben Parameter berücksichtigen muß.

- Als Parameter übergebene primitive Konstanten in Methoden, werden unverändert übergeben. Die Werte der als Parameter übergebene Variablen primitiven Typs, werden ausgelesen und als Wert übergeben.
- Für nicht- primitive Variablen in Methoden werden die IDs der entsprechenden Variablenkonstrukte übergeben. Außerdem werden die Objekt- IDs der Variablenkonstrukte ausgelesen und als jeweils zusätzliche Parameter übergeben.
- Wird der Rückgabewert eines Methodenaufrufs einer Variablen zugeordnet, wird das entsprechende CSP/FDR Variablenkonstrukt als letzter Parameter übergeben. Existiert noch keine Variable, dem der Rückgabewert zugeordnet werden soll (z.B. in Konstrukten der Form `x=2+calc(1)`), wird ein neues Variablenkonstrukt angelegt, das als Rückgabeparameter übergeben wird.

Eine Java Methode, die die im vorigen Beispiel eingeführte Methode benutzt, sieht folgendermaßen aus:

```
public int calc1() {
   int x=calc(1);
   return x;
}
```

Nach Abschluß der Transformation liegt eine Funktion vor, die das `this`- Objekt der Java Methode in Form von zwei Parametern übergibt, die primitive Konstante `1` unverändert an die aufzurufende Funktion weiterreicht und als Rückgabeparameter das Variablenkonstrukt `calc1_x` benutzt.

```
calc1(this,objectID_this,return) =
  calc(this,objectID_this,1,calc1_x) ;
  calc1_x?x ->
  return!x -> SKIP
```

Kapitel 3

Implementierung

Dieses Kapitel beschreibt die Implementierung des Übersetzungssystems. Es stellt zunächst die einzelnen Komponenten vor und beschreibt anschließend deren Implementierung im Detail.

3.1 Struktur des Systems

Folgendes Blockdiagramm gibt einen Überblick über die einzelnen Komponenten des Systems. Sie lassen sich grob in die Bereiche der Bytecode- und Syntaxbaumanalyse sowie Codeerzeugung gliedern. Nach Abschluß der Bytecodeanalyse liegen eine Reihe von Syntaxbäumen vor, die durch die Syntaxbaumanalyse transformiert werden. Aus diesen transformierten Bäumen wird anschließend eine CSP/FDR- Abstraktion erzeugt.

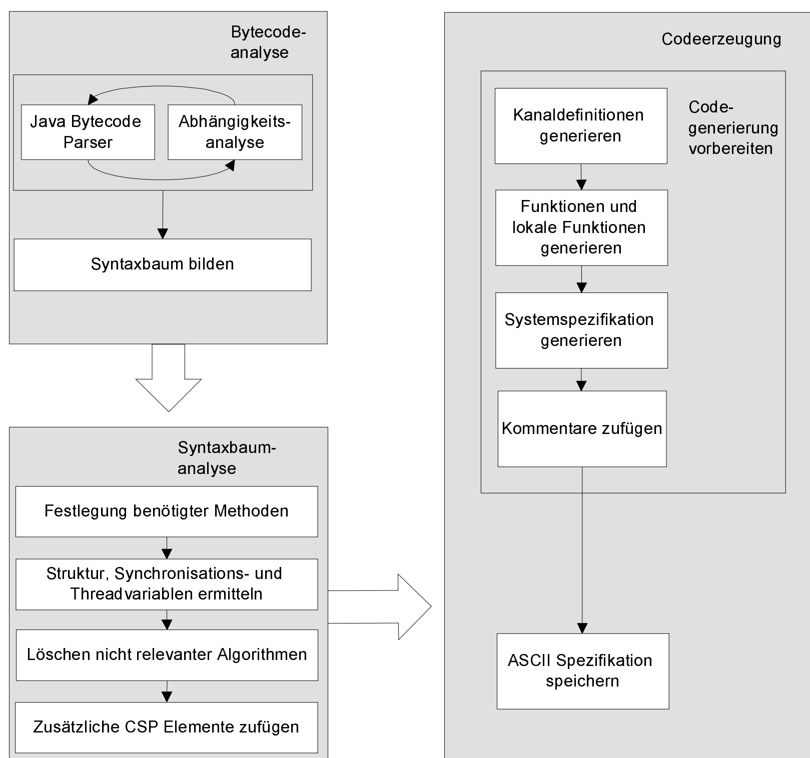

Abbildung 3-1: Struktur des Gesamtsystems

Die folgenden Abschnitte befassen sich detailliert mit der Implementierung der einzelnen Komponenten des Systems.

Zunächst folgt die Beschreibung des Java Class File Parsers mit einer Untersuchung des Class File Formats, einer Beschreibung der einhergehenden Ablage der eingelesenen Daten und der Abhängigkeitsanalyse der Klassendateien untereinander.

Der zweite Abschnitt beschäftigt sich mit der Analyse des gelesenen Bytecodes und beschreibt die hierzu verwendeten Strukturen und die daraus resultierende abstrakte Syntax. Anschließend werden die einzelnen Optimierungs- und Analyseoperationen auf der abstrakten Syntax beschrieben.

Der dritte Bereich dieses Kapitels erläutert die Übersetzung der vorliegenden Methodeninformationen und Syntaxbäume in eine CSP/ FDR- Spezifikation. Ein besonderer Schwerpunkt ist dabei den Transformationsregeln gewidmet, die festlegen, wie die einzelnen Java Sprachbestandteile in CSP/ FDR- Konstrukte überführt werden.

3.2 Java Class File Parser

An dieser Stelle soll zunächst eine genauere Betrachtung des Java Class File Formats vorgenommen werden, da der implementierte Übersetzer dieses Format als Quelle benutzt. Anschließend findet eine Untersuchung der Laufzeitstrukturen und der Codeanteile von Methoden statt. Im praktischen Teil dieser Arbeit ist ein solches Laufzeitsystem und ein Analysator für den nutzbaren Code von Methoden enthalten, so daß eine eingehende Betrachtung dieser Aspekte unabdingbar ist.
Im Anschluß daran wird die Arbeitsweise der Parserimplementierung und die durch den Parser gebildete Struktur zur Repräsentation der eingelesenen Dateien beschrieben. Daran anschließend wird die Vorgehensweise zur Auflösung von Abhängigkeiten einzelner Klassendateien untereinander diskutiert.

3.2.1 Das Java Class File Format

Das Java Class File Format stellt die Repräsentation einer Java- Klasse oder eines Java-Interfaces nach dem Übersetzen dar. Class- Dateien sind vergleichbar mit Objekt- Dateien, die beispielsweise von C- Compilern erzeugt werden. Sie enthalten die Strukturen und Symbole, die für die Benutzung des jeweiligen Typen notwendig ist.
Das Dateiformat, daß jeder Java Bytecode Compiler erzeugt und die jede Implementierung einer Java Virtual Machine interpretieren können muß, ist in [LiYe97] verbindlich festgelegt.

3.2.1.1 Dateiformat

Jede Klassendatei beschreibt genau einen Java Typen, wobei dies genau eine Java- Klasse oder ein Java- Interface sein kann. In jeder Klassendatei sind folgende Bestandteile enthalten:

- ein Konstanten- Pool, der alle von der Klasse benutzten Literale und Symbole enthält,
- eine symbolische Referenz zur Superklasse der jeweiligen Klasse,
- symbolische Referenzen auf die implementierten Interfaces der jeweiligen Klasse bzw. Interfaces,
- eine Liste der Felder die durch die Klasse definiert wird,
- eine Liste der Methoden die durch die Klasse definiert wird,
- Instruktionen für die einzelnen Methoden, die die jeweilige Klasse enthält,
- optionale Informationen, die durch Debugger genutzt werden können.

Datenrepräsentation innerhalb von Klassendateien

Jede Klassendatei besteht aus einem Bytestrom. Alle 16-Bit, 32-Bit und 64-Bit Werte werden gebildet, in dem sie aus zwei, vier oder acht aufeinanderfolgenden separaten 8-Bit Werten zusammengesetzt werden. Daten, die mehrere Bytes umfassen, werden immer im Big-Endian Format abgelegt, wobei zuerst die höherwertigen und dann die niederwertigen Bytes abgelegt werden.

Ganzzahlen und Fließkommawerte

Das Java Virtual Machine Class File Format unterscheidet zwei ganzzahlige und zwei Fließkomma Datentypen.
Die ganzzahligen Darstellungen sind `Integer` und `Long` benannt, wobei diese 32-Bit bzw. 64-Bit vorzeichenbehaftete Ganzzahlen darstellen.

`Float` und `Double` sind 32-Bit bzw. 64-Bit Fließkommawerte, die nach dem IEEE 754 Format (vgl. [IEEE754]) abgelegt werden.

Zeichenketten

Java benutzt zur Repräsentation von Zeichen den Unicode- Zeichensatz (vgl. [Unicode92]), der nahezu alle Zeichensätze der auf der Welt vertretenen Schriften abdeckt. Jedes Zeichen wird dabei durch einen 16- Bit- Wert repräsentiert.

Unicode- Zeichenketten werden innerhalb von Klassendateien im UTF8 Format (vgl. [Open93]) dargestellt. Dieses Format bietet gegenüber einer einfachen, linearen Zeichenrepräsentation den Vorteil, daß Unicode- Zeichen, die mit 8-Bit korrekt dargestellt werden könnten, auch nur ein Byte Speicherplatz pro Zeichen innerhalb der Zeichenkette benötigen. Das Format bietet jedoch zusätzlich die Möglichkeit, 16-Bit große Unicode Zeichen abzulegen. Der Grund für die Wahl dieses Formates liegt darin, die Ablage von String- Konstanten innerhalb von Class Files mit möglichst geringen Speicherverbrauch zu verbinden.

Voll qualifizierte Bezeichner

Innerhalb von Klassendateien werden alle Typen, Methoden und Felder mit voll qualifizierten Bezeichnern angegeben.

Ein Klassen- bzw. Interfacename ist dann voll qualifiziert, wenn er aus dem Java Paketnamen in dem er definiert wurde und dem eigentlichen Klassen bzw. Interfacenamen besteht (z.B. `java/lang/String` oder `java/lang/Runnable`).

Methoden- und Feldbezeichnern wird der jeweilige Klassen- bzw. Interfacename vorangestellt. Eine Methode `foo` und ein Feld `x` in der Klasse `demo` des Paketes `test` hätte dann innerhalb der Klassendatei die Repräsentation `test/demo/foo` bzw. `test/demo/x`.
Hierbei ist zu beachten, daß das innerhalb der Programmiersprache Java benutze Trennzeichen Punkt (.) bei der internen Darstellung in ein Slash- Zeichen (/) überführt wird.

Die primitiven Typen verfügen über eine Sonderstellung, da sie nicht als Klassen definiert sind und bereits durch den einfachen Namensbezeichner als voll qualifiziert gelten (z.B. `short` oder `long`).

3.2.1.2 Struktur des Java Class File Modells

Wir betrachten in diesem Abschnitt zunächst die wichtigsten Bestandteile einer Klassendatei, bevor eine detaillierte Beschreibung, und die Bildung der Beziehungen der einzelnen Elemente untereinander, erläutert wird.

Magic number

Die *magic number* ist ein Schlüssel, der angibt, daß die vorliegende Datei im Java Class File Format abgelegt wurde. Die magic number für Java Class Files besteht aus vier Bytes, die die Dezimalwerte 202,254,186,190 enthalten. Sinnigerweise ergibt das in hexadezimaler Notation CA, FE, BA, BE.

Versionsnummer

Innerhalb des Class Files ist zusätzlich eine Versionsnummer vom jeweils benutzten Compiler abgelegt, der das jeweilige Class File erzeugt hat. Diese kann für eventuelle Kompatiblitätsüberprüfungen benutzt werden.

Konstanten- Pool

Der Konstanten- Pool ist ein Feld, daß aus Strukturen variabler Länge besteht. Die einzelnen Strukturen innerhalb des Konstanten- Pools stellen verschiedene String Konstanten, Klassen- und Feldnamen sowie numerische Konstanten dar. Auf diese einzelnen Elemente wird innerhalb der Class- File Struktur referenziert.

Konstantentyp	Beschreibung	Referenzen
`Integer`	Repräsentation einer Konstanten vom Java Datentyp `int`	keine
`Float`	Repräsentation einer Konstanten vom Java Datentyp `float`	keine
`Long`	Repräsentation einer Konstanten vom Java Datentyp `long`	keine
`Double`	Repräsentation einer Konstanten vom Java Datentyp `double`	keine
`Utf8`	Repräsentation einer Zeichenkette im UTF8- Format	keine
`String`	Repräsentation einer Konstanten vom Java Datentyp `java.lang.String`	Referenz auf eine `Utf8` Konstante, die den eigentlichen String enthält
`Class`	Repräsentation einer Klasse oder eines Interfaces	Referenz auf eine `Utf8` Konstante mit voll qualifiziertem Java Klassennamen
`Fieldref`	Repräsentation einer Referenz auf ein Feld in einer beliebigen Klasse	Referenz auf eine `Class` Konstante und eine `NameAndType` Konstante
`Methodref`	Repräsentation einer Referenz auf eine Methode in einer beliebigen Klasse	Referenz auf eine `Class` Konstante und eine `NameAndType` Konstante
`InterfaceMethodref`	Repräsentation einer Referenz auf eine Methode in einem beliebigen Interface	Referenz auf eine `Class` Konstante und eine `NameAndType` Konstante
`NameAndType`	Repräsentation eines Feldes oder einer Methode, ohne Angabe einer Klassen oder Interfacezugehörigkeit	Referenz auf zwei `Utf8` Konstanten, wobei die erste den Namen und die zweiten den Typ als Felddeskriptor oder Methodendeskriptor enthält

Abbildung 3-2: Arten von Konstanten im Konstanten Pool

Modifizierer

Es existieren innerhalb der Class File Definition eine Reihe von Modifizierern, die verschiedene Eigenschaften von Klassen, Interfaces, Feldern und Methoden festlegen. Die folgende Abbildung gibt jeweils den Namen dieser Modifizierer an, liefert eine Beschreibung der Bedeutung und legt fest, welche Konstrukte eines Programms in Class File Darstellung diese Modifizierer anwenden dürfen.

Modifier	Beschreibung	möglicher Anwender
`ABSTRACT`	Es gibt keine Implementierung dieser Methode.	beliebige Klasse, beliebiges Interface, beliebige Methode
`FINAL`	Ein Überschreiben der Methode ist nicht möglich.	beliebige Klasse, beliebiges Feld, beliebige Klassen/ Instanzmethode
`INTERFACE`	Das Konstrukt ist ein Interface.	jedes Interface
`NATIVE`	Das Konstrukt ist in einer anderen Sprache als Java (möglicherweise betriebssystemabhängig) implementiert.	beliebige Klassen/ Instanzmethode
`PRIVATE`	Das Konstrukt kann nur in der Klasse benutzt werden, in der es definiert wurde.	Feld einer Klasse, beliebige Klassen-/ Instanzmethode
`PROTECTED`	Das Konstrukt kann in Unterklassen verwendet werden.	Feld einer Klasse, beliebige Klassen/ Instanzmethode
`PUBLIC`	Auf das Konstrukt kann von außerhalb des Pakets zugegriffen werden.	beliebige Klasse, beliebiges Interface, beliebige Feld, beliebige Methode
`STATIC`	Auf das Konstrukt kann nur innerhalb der Klasse zugegriffen werden.	beliebiges Feld, beliebige Klassen/ Instanzmethode
`SUPER`	Das Konstrukt besitzt Superklasse bzw. Superinterface.	beliebige Klasse, beliebiges Interface
`SYNCHRONIZED`	Das Konstrukt ist synchronisiert. Es erfolgt der automatische Einsatz von Monitoren zur Durchsetzung des gegenseitigen Ausschlusses.	beliebige Klassen/ Instanzmethode
`TRANSIENT`	Auf das Konstrukt kann kein persistenter Objektmanager schreibend oder lesend zugreifen.	Feld einer Klasse
`VOLATILE`	Das Konstrukt darf nicht innerhalb eines Caches gespeichert werden.	Feld einer Klasse

Abbildung 3-3: Modifizierer in Class Files

Zwischen den Modifizierern innerhalb von Klassendateien bestehen die folgenden Abhängigkeiten:

- Die Verwendung von `FINAL` und `ABSTRACT` schließt sich gegenseitig aus.
- Innerhalb einer Interfacedefinition bedingt die Angabe von `INTERACE` das Setzen von `ABSTRACT` für alle Methoden innerhalb dieser Klassendatei.
- In Interfacedefinitionen (Modifizierer `INTERFACE` ist gesetzt), darf `FINAL` nicht gesetzt werden.
- Nur jeweils eine Konstante aus `PUBLIC`, `PRIVATE` und `PROTECTED` darf gesetzt werden.
- Innerhalb von Interfaces müssen alle Methoden die Modifizierer `PUBLIC` und `ABSTRACT` besitzen.
- Für Methoden darf `ABSTRACT` nicht in Verbindung mit `FINAL`, `NATIVE`, `SYNCHRONIZED` oder `STATIC` gesetzt werden.

Referenzen auf Klassen, Interfaces, Methoden und Felder

Die bisherige Darstellung bezog sich lediglich auf den Inhalt einzelner isolierter Klassendateien. Es ist jedoch zusätzlich die Möglichkeit gegeben, auf Felder und Methoden anderer Klassen zuzugreifen. Hierzu bietet das Java Class File Modell Referenzen auf Methoden und Felder externer Klassen und Interfaces an.
Referenzen zu Klassen und Interfaces sowie zu Feldern und Methoden in Klassen und Interfaces werden als Symbole im Konstanten Pool abgelegt. Die hierfür vorgesehenen Konstantentypen sind `Class`, `Fieldref`, `Methodref` und `InterfaceMethodref`.
Ein Eintrag von Typ `Class` referenziert dabei auf einen UTF8 String (siehe 'Dateiformat' auf Seite 50) im selben Konstanten- Pool, der als Inhalt einen voll qualifizierten Klassennamen besitzt.
Feldreferenzen (Konstantentyp `Fieldref`), Methodenreferenzen auf Klassen (`Methodref`) und Methodenreferenzen auf Interfaces (`InterfaceMethodref`) haben hingegen eine Referenz auf einen Eintrag des Konstantentyps `Class` und auf einen Eintrag vom Typ `NameAndType`. Hierdurch wird eine Verbindung zur Klasse bzw. zum Interface geschaffen, indem sich das Feld bzw. die Methode befindet. Der Eintrag vom Typ `NameAndType` stellt dann eine Verknüpfung zu dem jeweiligen konkreten Feld oder der Methode her.

Ablage von Feldern und Methoden

Die Angabe der Felder und Methoden stellt ein Array von Strukturen variabler Länge dar, die eine vollständige Beschreibung eines Feldes bzw. einer Methode der Klasse oder des Interfaces liefert. Das Array enthält dabei nur Angaben über Felder und Methoden, die in dieser Klasse definiert worden sind. Vereinbarungen, die von Klassen oder Interfaces geerbt oder implementiert wurden, werden hier nicht explizit angegeben.

Spezielle Methoden

In Klassendateien sind alle Instanz- und Klassenmethoden der jeweiligen Klasse in ihrer ursprünglichen Quelltextnotation vorhanden. Eine Ausnahme bilden dabei die Konstruktoren, die den speziellen, in der Programmiersprache Java ungültigen Bezeichner <init> erhalten. Der Grund hierfür liegt darin, daß die Konstruktoren in einem Java-Programm nicht direkt ohne vorausgehende Reservierung eines Objekts für die Instanziierung aufgerufen werden sollen.

Ähnlich verhält es sich bei Interfacedefinitionen. Interfaces besitzen keine konkreten Konstruktoren, müssen aber dennoch von der Laufzeitumgebung initialisiert werden. Hierzu erhält jede Klassendatei eines Interfaces die Initialisierungsmethode <clinit>, die ebenfalls nicht von der Programmiersprache Java aus ansprechbar ist und für die Verwendung durch das Laufzeitsystem reserviert bleibt.

Feld- und Methodensignaturen

Eine Feldsignatur stellt den Typ einer Klassen- oder Instanzvariable dar und besteht aus einer Zeichenkette, die von der Grammatik `FieldDescriptor` gebildet wird.

FieldDescriptor:	*BaseType:*
FieldType	**B**
ComponentType:	**C**
FieldType	**D**
FieldType:	**F**
ArrayType	**I**
BaseType	**J**
ObjectType	**S**
ArrayType:	**Z**
[*ComponentType*	*ObjectType:*
	L <classname> ;

Abbildung 3-4: Grammatik für Feldsignaturen

Alle Methoden innerhalb eines Java Class Files besitzen eine Signatur, der die Typen der formalen Parameter und den Typ des Rückgabewerts festlegt. Diese Signatur wird durch die Grammatik `MethodDescriptor` beschrieben.

MethodDescriptor:
*(ParameterDescriptor *) ReturnDescriptor*
ParameterDescriptor:
FieldType
ReturnDescriptor:
FieldType
V

Abbildung 3-5: Grammatik für Methodensignaturen

Die terminalen Symbole der Grammatiken für Feld- und Methodensignaturen haben dabei die in der folgendem Abbildung dargestellte Bedeutung.

Terminalsymbol	Erwarteter Datentyp
[	Einführung einer Felddimension (Typdefinition muß folgen).
B	`byte`
C	`char`
D	`double`
F	`float`
I	`int`
J	`long`
L	Typ der nachfolgend angegebenen Klasse
S	`short`
V	`void`
Z	`boolean`

Abbildung 3-6: Nicht- Terminale für Signaturengrammatiken

Somit sind folgende Beispiele gültige Signaturen:

- `Ljava2csp/test/foo;`
- `[[[[S`
- `[Ljava2csp/test/foo2;`

Attribute

Attribute stellen eine Beschreibung der Eigenschaften der einzelnen Bestandteile einer Klasse dar. Eine Liste der möglichen Attributtypen findet sich in der folgenden Abbildung.

Attributtyp	Bedeutung	Anzahl	Referenzen
`SourceFile`	Angabe der Quellcodedatei aus der die Class Datei compiliert wurde	max. 1 pro Klasse	`UTF8` String mit Dateinamen im Konstanten Pool
`ConstantValue`	Angabe des Werts einer Konstanten	max. 1 pro Feld	Integer, Float, Long oder Double Wert im Konstanten Pool
`Code`	Instruktionen, die bei Aufruf einer Methode ausgeführt werden können (Codesegment einer Methode)	max. 1 pro Methode	`Exceptions` Attribut, `LineNumbertable` Attribut (optional), `LocalVariableTable` Attribut (optional)
`Exceptions`	Information über Ausnahmen innerhalb einer Methode geworfen werden können	max. 1 pro Methode	Class Referenz im Konstanten Pool, die Typ der zu werfenden Ausnahme darstellt
`LineNumberTable`	Abbildung von Positionen im Codesegment zu Zeilen im Quellcode	beliebig pro Methode	-
`LocalVariableTable`	Abbildung lokaler Variablen im Codesegment auf ihre Namen im Quellcode	beliebig pro Methode	-

Abbildung 3-7: Class- File- Attribute und deren Bedeutung

Die aktuelle Java Class File Spezifikation enthält als einziges mögliches Attribut für eine Klasse bzw. ein Interface das `SourceFile`- Attribut.

An Felder kann das Attribut `ConstantValue` geknüpft sein, während Methoden die Attribute `Code`, `Exception`, `LineNumberTable` und `LocalVariableTable` enthalten können. `LineNumberTable` und `LocalVariableTable` sind dabei optionale Attribute, die z.B. von Debuggern genutzt werden können, um Positionen im Bytecode auf Zeilennummern der Quelldateien bzw. lokale Variablen auf deren Namen im Quellcode umsetzen zu können. Diese Konstrukte werden in der Regel nur angelegt, wenn der entsprechende Java Compiler explizit dazu veranlaßt wird.

Begrenzungen in Klassendateien

Die Elemente einer Klassendatei sind Limitierungen hinsichtlich ihrer Größe unterworfen. Die folgende Abbildung zeigt die einzelnen Elemente und die dazugehörigen Größeneinschränkungen.

Beschreibung	Limitierung
Einträge im Kostanten Pool	65535 Worte pro Klasse
Felder	65535 Felder pro Klasse
Methoden	65535 Methoden pro Klasse
Länge des Bytecodes	65535 Bytes pro Methode
Lokale Variablen	65535 Worte pro Methode
Argumente einer Methode	255 Worte
Operandenstack	65535 Worte pro Methode
Array Dimensionen	255 Array Dimensionen pro Array

Abbildung 3-8: Limitierungen der Elemente einer Klassendatei

3.2.2 Laufzeitstrukturen

Während der Laufzeit eines Java- Programms werden eine Reihe von Strukturen benutzt, die zur internen Organisation der Programmausführung dienen.

3.2.2.1 Register

Im Gegensatz zu Hardware Prozessoren enthält die Java Laufzeitumgebung keine allgemein verwendbaren Register. Innerhalb der Virtual Machine werden jedoch Zustandsinformationen der einzelnen ausgeführten Threads, Zeiger auf die momentan ausgeführte Methode und ein Programmzähler gehalten, der angibt, an welcher Stelle innerhalb des Codes sich die Ausführung gerade befindet. Diese Informationen sind nicht vom Code aus zu nutzen, sondern lediglich vom Laufzeitsystem aus selbst zugreifbar.

3.2.2.2 Operandenstack

Die virtuelle Java Maschine arbeitet Stack- basiert. Operationen innerhalb der Maschine erhalten ihre Argumente durch das Auslesen des Operandenstacks und legen ihre Rückgabewerte ebenfalls dort wieder ab. Der Operandenstack enthält 32-Bit Einträge. Werte vom Datentyp `long` und `double` benötigen durch ihre Repräsentation im 64-Bit Format zwei Einträge auf dem Stack. Der Operandenstack ist im Unterschied zu hardwarebasierten Prozessoren nicht global. Jeder Methodenaufruf führt zur Einführung eines neuen Stacks, so daß die einzelnen Methodenausführungen gänzlich voneinander getrennt sind. Für die Verwaltung der einzelnen Operandenstacks ist allein das Laufzeitsystem zuständig.

3.2.2.3 Methodenrahmen (Frames)

Wenn eine Methode innerhalb der Java Virtual Machine aufgerufen wird, führt dies zur Erzeugung eines neuen Methodenrahmens. Die Lebensdauer des Methodenrahmens ist äquivalent zur Ausführungszeit der Methode.
Die Frame Struktur enthält einen Operandenstack, Platz zur Speicherung lokaler Variablen der zugehörigen Methode und Zustandsvariablen wie dem Programmzähler, einen Zeiger auf die Klasse der ausgeführten Methode und einen Zeiger auf den Frame der aufrufenden Methode.

Frame Stack

Die Verwaltung der einzelnen Frames erfolgt auf dem sogenannten *Java Stack* (vgl. [LiYe97]). Beim Aufruf einer Methode wird der Frame der aufrufenden Methode vom Laufzeitsystem auf den *Java Stack* gelegt. Nach der Beendigung der Ausführung der Methode kann eine Virtual Machine Implementierung durch Entnahme des obersten Elements vom Java Stack effizient wieder in den Kontext der aufrufenden Methode zurückspringen.
Der Aufbau der Java Virtual Machine impliziert, daß jeder Kontrollfaden einen eigenen Stack Frame besitzt. Dadurch wird gewährleistet, daß die Threads unabhängig voneinander auf eigenen Frames operieren.

3.2.2.4 Lokale Variablen

Jeder Methodenaufruf führt zur Allozierung von Speicherplatz für die Menge der lokalen Variablen der aufgerufenen Methode. Die lokalen Variablen enthalten zunächst eine Referenz auf das Objekt, auf dem die Methode aufgerufen wurde (`this`) und die Inhalte der formalen Parameter, die der Methode übergeben wurden. Compiler können zusätzlich anonyme lokale Variablen erstellen, die zur Sicherung von temporären Daten benutzt werden. Innerhalb der Programmiersprache Java, werden lokale Variablen durch ihren Bezeichner identifiziert, während sie innerhalb der Laufzeitumgebung eine eindeutige Numerierung pro Frame erhalten.

3.2.3 Code- Segmente

Das Code Segment stellt für den praktischen Teil dieser Arbeit die wohl wichtigste Struktur im Java Class File Modell dar. Jede Methode, die innerhalb einer Klasse definiert wurde, kann maximal ein Codesegment enthalten, das aus einer Struktur des Attributtyps `Code` hervorgeht.

Innerhalb der Code Struktur liegen alle Befehle als Bytestrom für die Java Virtual Maschine vor, die diese beim Aufruf der Methode auszuführen hat. Die verschiedenen Befehle lassen sich in die Gebiete arithmetische Operationen, Verzweigungen, Datenmanipulation sowie Methodenaufrufe aufteilen, die wir nachfolgend im einzelnen beleuchten.

3.2.3.1 Arithmetische Operationen

Alle arithmetischen Funktionen, die die Programmiersprache Java anbietet, existieren direkt auf der Ebene des Bytecodes. Sie sind dabei unterteilt nach den primitiven Typen, die das Class File Format beherrscht. Eine Liste der verfügbaren Operationen und Datentypen, auf denen sie ausgeführt werden können, liefert die nachfolgende Abbildung. Alle Operationen entnehmen ein oder zwei Werte des angegebenen Typs vom Operandenstack und legen das Ergebnis nach der Berechnung wieder zurück. Der Ergebnistyp der Operationen ist jeweils wieder der unterstützte Datentyp.

Operation	Beschreibung	unterstützte Datentypen
`add`	Addition zweier Werte	`int,long,float,double`
`sub`	Subtraktion zweier Werte	`int,long,float,double`
`mul`	Multiplikation zweier Werte	`int,long,float,double`
`div`	Division zweier Werte	`int,long,float,double`
`rem`	Rest der Division zweier Werte	`int,long,float,double`
`neg`	Negierung eines Werts	`int,long,float,double`
`shl`	links shift	`int,long`
`shr`	rechts shift	`int,long`
`ushr`	logisches rechts shift (ohne Betrachtung des Vorzeichenbits)	`int,long`
`and`	bitweises und	`int,long`
`or`	bitweises oder	`int,long`
`xor`	bitweises exklusives oder	`int,long`
`2l`	Konvertierung in den Datentyp long	`int,float,double`
`2f`	Konvertierung in den Datentyp float	`int,long,double`
`2d`	Konvertierung in den Datentyp double	`int,long,float`
`2i`	Konvertierung in den Datentyp int	`long,float,double`
`2b`	Konvertierung in den Datentyp byte	`int`
`2c`	Konvertierung in den Datentyp char	`int`
`2s`	Konvertierung in den Datentyp short	`int`

Abbildung 3-9: Arithmethische Befehle im Java Class File Format

Die Operationen `2b`, `2c` und `2s` stellen eine Besonderheit dar, da die Ergebnisstypen zwar `byte`, `char` bzw. `short` sind, die Virtual Machine diese Datentypen allerdings nicht direkt unterstützt, sondern sie auf `int` Werte abbildet.

3.2.3.2 Sprünge und Verzweigungen

Sprünge und Verzweigungen sind wichtige Eigenschaften in einem Laufzeitsystem. Sie sorgen für die Struktur innerhalb eines Programms und ermöglichen so die Umsetzung komplexer Algorithmen.

Sprünge

Sprünge (unbedingte Verzweigungen) sorgen dafür, daß der Programmzähler auf einen explizit angegebenen Wert gesetzt wird und die Abarbeitung des Codes durch das Laufzeitsystem an dieser Stelle fortgesetzt wird. Des weiteren existieren spezielle Sprünge für Unterprogrammaufrufe, die den Wert des momentanen Programmzählers sichern und an

eine andere Stelle innerhalb des Codes verzweigen. Um aus einem Unterprogrammaufruf zurückzukehren, setzen spezielle Rücksprungoperationen den gespeicherten Programmzähler zurück und sorgen für die weitere Ausführung nach der aufrufenden Position.

Operation	Beschreibung
`goto, goto_w`	Sprung an eine Position im Code mit einfachem oder erweitertem Offset
`jsr, jsr_w`	Unterprogrammaufruf mit einfachem oder erweitertem Offset
`ret, ret_w`	Rückkehr aus einem Unterprogrammaufruf mit einfachem oder erweitertem Offset

Abbildung 3-10: Sprünge

Die Operationen `goto_w`, `jsr_w` und `ret_w` benutzen im Gegensatz zu den anderen dargestellten Operationen, deren Sprungadresse maximal ein 8-Bit Wert sein kann, eine 16-Bit große Sprungadresse.

Bedingte Verzweigungen

Die bedingten Verzweigungen unterscheiden sich von den Sprüngen dadurch, daß sie nur dann ausgeführt werden, wenn eine festgelegte Bedingung erfüllt ist. Andernfalls wird die Programmausführung hinter der Verzweigungsanweisung fortgeführt. Die im folgenden vorgestellten Operationen bedienen sich zur Durchführung eines Vergleiches jeweils eines bzw. zweier Elemente vom Operandenstack.

Operation	Beschreibung
`ifnull`	Sprung, wenn Null- Referenz (`null`) vorliegt
`ifnonnull`	Sprung, wenn die Null- Referenz (`null`) nicht vorliegt
`ifeq`	Sprung, wenn Integerwert 0 vorliegt
`iflt`	Sprung, wenn Integerwert kleiner als 0
`ifle`	Sprung, wenn Integerwert kleiner oder gleich 0
`ifne`	Sprung, wenn Integerwert ungleich 0
`ifgt`	Sprung, wenn Integerwert größer als 0
`ifge`	Sprung, wenn Integerwert größer oder gleich 0
`if_icmpeq`	Sprung, wenn zwei Integerwerte gleich sind
`if_icmpne`	Sprung, wenn zwei Integerwerte ungleich sind
`if_icmplt`	Sprung, wenn der erste Integerwert kleiner ist als der zweite
`if_icmpgt`	Sprung, wenn der erste Integerwert größer ist als der zweite
`if_icmple`	Sprung, wenn der erste Integerwert kleiner oder gleich dem zweiten ist
`if_icmpge`	Sprung, wenn der erste Integerwertgrößer oder gleich dem zweiten ist

Abbildung 3-11: Bedingte Verzweigungen

Zusätzlich zu den vorgestellten bedingten Sprüngen, existieren die Operationen `lookupswitch` und `tableswitch`, die das Java- Konstrukt `switch` abbilden. Sie bieten die Möglichkeit, Sprünge für eine Reihe von Referenzwerten anzugeben. Diese Werte können mit einem zur Laufzeit erzeugten Wert verglichen werden, wobei bei einer Gleichheit der dafür angegebene Programmzählerwert angesprungen wird.

3.2.3.3 Datenmanipulation

Die Manipulationen auf Daten innerhalb der Java Virtual Machine lassen sich in Stackmanipulation sowie Zugriffe auf lokale Variablen, Objekte und Arrays unterteilen.

Stackmanipulation

Ein elementarer Bestandteil der virtuellen Java Maschine ist der Operandenstack, der für eine Vielzahl von Berechnungen benötigt wird. So benutzen beispielsweise nahezu alle Bytecodeoperationen den Stack zur Entnahme und Rückgabe von Argumenten und Ergebniswerten. Die folgende Abbildung zeigt die Funktionen, die ein Laufzeitsystem zur Ablage von Konstanten auf den Stack anbietet.

Operation	Beschreibung
bipush	einen Wert des Datentyps byte auf den Stack legen
sipush	einen Wert des Datentyps short auf den Stack legen
ldc	eine ein Wort große Konstante auf den Stack legen
ldc_w	eine ein Wort große Konstante auf den Stack legen (erweiterter Index)
ldc2_w	eine zwei Wort große Konstante auf den Stack legen (erweiterter Index)
aconst_null	(Die Null- Referenz) null auf den Stack legen
iconst_m1	Konstante -1 vom Datentyp int auf den Stack legen
iconst_<i>	Konstante <i>=0,1,2,3,4 oder 5 vom Datentyp int auf den Stack legen
lconst_<l>	Konstante <l>=0 oder 1 vom Datentyp long auf den Stack legen
fconst_<f>	Konstante <f>=0.0, 1.0 oder 2.0 vom Datentyp float auf den Stack legen
dconst_<d>	Konstante <d>=0.0 oder 1.0 vom Datentyp double auf den Stack legen

Abbildung 3-12: Operationen zum Legen von Kostanten auf den Stack

Nachdem nun erläutert ist, wie konstante Werte auf den Stack kommen, folgt eine Beschreibung der Funktionen zur Veränderung von Werten auf dem Stack.

Operation	Beschreibung
pop	das oberste Wort vom Stack entnehmen
pop2	die obersten zwei Wörter vom Stack nehmen
dup	das oberste Wort auf dem Stack duplizieren
dup2	die obersten zwei Wörter auf dem Stack duplizieren
dup_x1	das oberste Wort auf dem Stack duplizieren und unter dem zweiten Wort auf dem Stack ablegen
dup2_x1	die obersten zwei Wörter auf dem Stack duplizieren und unter dem dritten Wort auf dem Stack ablegen
dup_x2	das oberste Wort auf dem Stack duplizieren und unter dem dritten Wort auf dem Stack ablegen
dup2_x2	die obersten zwei Wörter auf dem Stack duplizieren und unter dem vierten Wort auf dem Stack ablegen
swap	die beiden oberen Wörter auf dem Stack vertauschen

Abbildung 3-13: Operation zur Stackmanipulation

Manipulation Lokaler Variablen

Lokale Variablen werden in der Laufzeitumgebung zur Ausführungszeit erstellt. Sie können lediglich gelesen und geschrieben werden. Die im folgenden aufgeführten Operationen sind mit dafür vorgesehenen Suffixen für die Datentypen `int`, `long`, `float`, `double` sowie für beliebige Objekt- und Arraytypen zulässig.

Operation	Beschreibung
`load`	Wert aus lokaler Variable lesen und auf den Stack legen
`load_<n>`	Wert aus lokaler Variable <n> lesen und auf den Stack legen
`store`	Wert vom Stack nehmen und in lokaler Variable speichern
`store_<n>`	Wert vom Stack nehmen und in lokaler Variable <n> speichern

Abbildung 3-14: Operationen auf lokalen Variablen

Objekterzeugung und -manipulation

Es ist möglich, neue Objekte zu erzeugen und Felder einzelner Objekte zu verändern. Ein neues Objekt wird mittels der Operation `new` innerhalb der Virtual Machine angelegt. Diese Operation erwartet dabei den Typ des zu schaffenden Objekts, wobei hierbei keine implizite Initialisierung stattfindet.

Des weiteren lassen sich die Felder eines Objekts, nachdem es initialisiert wurde manipulieren. Dabei muß zunächst zwischen statischen Feldern und nichtstatischen Feldern unterschieden werden. Die erste Gattung wird innerhalb der Klassendarstellung gespeichert, während die zweite innerhalb einer Instanz einer Klasse (einem Objekt) abgelegt wird. Aus diesem Grund werden die Zugriffsoperationen in statische und nichtstatische unterschieden, wie folgende Abbildung zeigt.

Operation	Beschreibung
`putfield`	Wert eines Feldes innerhalb eines Objektes setzen
`getfield`	Wert eines Feldes innerhalb eines Objektes auslesen
`putstatic`	Wert eines statischen Feldes setzen
`getstatic`	Wert eines statischen Feldes auslesen

Abbildung 3-15: Operationen auf Feldern

Arrayerzeugung und -manipulation

Arrays werden als Objekte repräsentiert und gehören damit zu definierten Klassen. Dies wird allein dadurch deutlich, daß man alle Methoden, die sich auf ein Objekt der Klasse `java/lang/Objekt` ausführen lassen, auch auf Array Objekten durchführen lassen. Auf die Einträge eines Arrays wird allerdings durch Indizes zugegriffen, statt, wie bei Objekten, durch Referenzen auf Felder. Außerdem sind Felder im Vergleich zu Objekten homogen (alle Komponenten sind vom selben Typ) und stellen Konstrukte variabler Länge dar.

Arrayklassendefinition

Arraytypen werden innerhalb der Virtual Machine durch Klassen repräsentiert, die automatisch zur Laufzeit erzeugt werden. Wird ein Array mit einem konkreten Typen definiert, erzeugt das Laufzeitsystem einen Arraytypen für diese Definition, der wiederverwendet werden kann. Die Klassenhierarchie für einige Arraytypen wird im folgenden dargestellt.

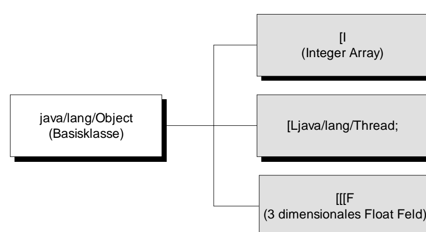

Abbildung 3-16: Beispiel für Array Klassen

Die folgende Abbildung zeigt zunächst die drei Funktionen, die Arrays erzeugen, und anschließend die Operationen, mit denen sich Werte aus Arrays lesen und in Arrays speichern lassen.

Operation	Beschreibung
`newarray`	neues Array für Zahlenwerte erstellen
`anewarray`	neues Array für Objektreferenzen erstellen
`multianewarray`	neues multidimensionales Array erstellen
`aload`	Wert aus Array auslesen
`astore`	Wert in ein Array speichern

Abbildung 3-17: Operationen auf Arrays

Die Operationen `aload` und `astore` sind für Arrays der Datentypen `int`, `long`, `float`, `double`, `byte`, `short` sowie für beliebige Objekt- und Arraytypen zulässig.

3.2.3.4 Methodenaufruf und -rückehr

In Java Programmen werden unablässig Methoden aufgerufen, so daß der Mechanismus zum Aufruf und zur Rückehr von Methoden effizient gestaltet sein muß.

Der Ablauf eines allgemeinen Methodenaufrufs ist in der folgenden Abbildung dargestellt. Er beschreibt, welche Operationen dabei im einzelnen explizit angestoßen werden müssen, und welche durch die Laufzeitumgebung automatisch durchgeführt werden.

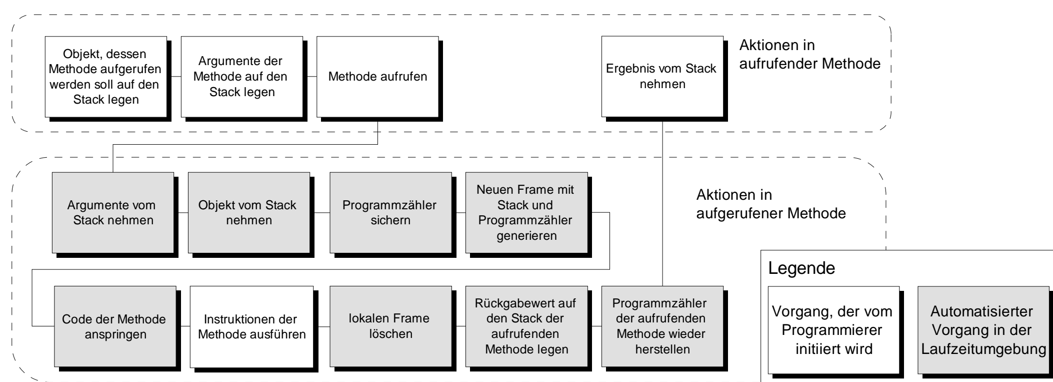

Abbildung 3-18: Ablauf eines Methodenaufrufs

Instruktionen für Methodenaufrufe

Es gibt die folgenden vier Instruktionen, mit denen sich Methoden aufrufen lassen. Sie sind dabei jeweils auf eine oder mehrere Arten von Methoden spezialisiert.

Operation	Beschreibung
`invokevirtual`	Aufruf einer Instanzmethode
`invokespecial`	Aufruf einer Methode in einer speziellen Klasse
`invokestatic`	Aufruf einer Klassenmethode
`invokeinterface`	Aufruf einer Interfacemethode

Abbildung 3-19: Instruktionen zum Aufruf von Methoden

`Invokespecial` besitzt im Vergleich zu den anderen Instruktionen eine etwas komplexere Semantik. Sie wird benutzt, um folgende Aufrufe zu realisieren:

- Aufruf von Instanzinitialisierungsmethoden (`<init>`)
- Aufruf von Methoden auf ein Objekt innerhalb von Methoden des selben Objekts (`this`- Aufrufe),
- Aufruf privater Methoden (Zugriff nur innerhalb des Objekts)
- Aufruf von Methoden der Superklasse

Auflösung von Methodenaufrufen

Ein Methodenaufruf wird durch die Angabe eines Methodennamens und dessen Signatur identifiziert. Hierbei ist durch die Bildung von Klassenhierarchien nicht immer sichergestellt, daß die Methode auch direkt in der angesprochenen Klasse definiert wurde. Beispielsweise kann eine Methode mit einer speziellen Signatur auf eine Klasseninstanz aufgerufen werden, wobei gerade diese Methode in der Superklasse oder der Superklasse der Superklasse des Typs definiert wurde. Die Auflösung dieser Verbindungen und die Auswahl der "richtigen" Methode wird automatisch von der Laufzeitumgebung durchgeführt. Sie berücksichtigt hierbei auch die korrekte Handhabung der Zugriffsrechte (`public`, `private`, `protected`) für die angesprochene Methode.

Instruktionen für Methodenrückkehr

Aus einer Methode in Ausführung wird mittels der `return` Instruktion zurückgesprungen. Es existieren für die Rückgabe von Werten der Datentypen `int`, `long`, `float` und `double`, sowie für Objekte jeweils spezielle `return` Instruktionen, denen jeweils das Datentyppräfix vorangestellt ist.

3.2.3.5 Threadunterstützung

Auf Ebene der Java Virtual Machine, gibt es drei Bereiche, in denen ein Java Programm mit Threads interagiert:

- Aufrufe von Methoden der Klasse `java.lang.Thread`
- Aufruf von Methoden, die synchronisiert sind
- Anwendung der Instruktionen `monitorenter` und `monitorexit`, um einzelne Codeabschnitte zu synchronisieren

Monitore

Innerhalb der Java Laufzeitumgebung werden Monitore benutzt, um den Zugriff auf Objekte durch mehrere Threads zu synchronisieren. Monitore lassen sich durch die beiden Instruktionen `monitorenter` und `monitoexit` ansprechen, wobei die erste Instruktion versucht, in einen kritischen Abschnitt einzutreten und die zweite anzeigt, daß der kritische Abschnitt verlassen wurde. Die Funktionsweise ist äquivalent zur Monitordefinition in Abschnitt 2.1.4 (Seite 12).

Synchronisierte Methoden

Methoden können als synchronisiert vereinbart werden, in dem sie mit dem Modifizierer `synchronized` definiert wird. Die gesamte Methode kann dann jeweils nur von einem Thread zur Zeit aufgerufen werden.
Innerhalb der Laufzeitumgebung findet dabei als erste Instruktion innerhalb der Methode eine Instruktion `monitorenter` auf das Objekt statt, in dem diese Methode aufgerufen wurde (`this`). Dies impliziert, daß bei einem Rücksprung aus der Methode analog ein `monitorexit` ausgeführt wird.

3.2.4 Arbeitsweise des Bytecode Parsers

Der Bytecode Parser besteht aus zwei elementaren Teilen:

- einem Parser zum Einlesen einer Klassendatei (nach der Definition in [LiYe97], siehe auch ’Das Java Class File Format’ auf Seite 50) und Ablegen der gelesenen Informationen, sowie
- einem übergeordneten System zum Anstoßen des Parsierungsprozesses und zur Auflösung daraus resultierender Klassenabhängigkeiten (Stapel- bzw. Batch- Parser).

Die Arbeitsweise der beiden Teilsysteme und deren Verknüpfungen untereinander veranschaulicht das folgende Ablaufdiagramm:

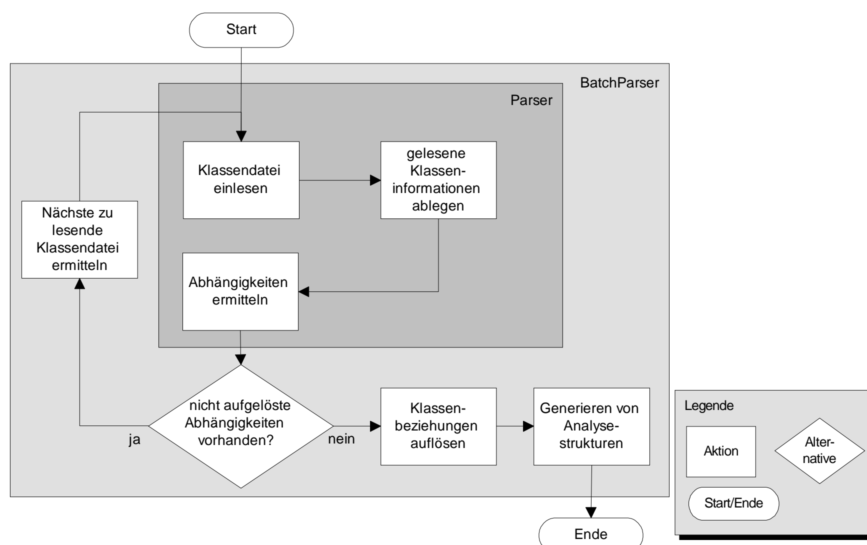

Abbildung 3-20: Arbeitsweise des Bytecode Parsers

Im den nachfolgenden Abschnitten betrachten wir den Vorgang der Stapelparsierung, das Einlesen der Klassendateien und die daraus resultierenden Ablagestrukturen im einzelnen.

3.2.4.1 Stapelparser (Batchparser)

Ein Java- Projekt besteht in der Regel aus einer Reihe von Klassen, wobei in den meisten Fällen zusätzlich Gebrauch von Java- API Komponenten gemacht wird. Aus diesem Grund muß es für den eigentlichen Bytecode- Parser ein Steuerungssystem geben, daß den Überblick über die bereits eingelesenen und die noch einzulesenden Klasse behält.

Hierzu wurde eine Stapelparser entwickelt, der über eine Tabelle verfügt, in der die bereits geladenen Klassen abgelegt werden. Des weiteren besteht die Möglichkeit, von jeder eingelesenen Klasse zu ermitteln, welche anderen Klassen sie (z.B. als Superklasse, Superinterface oder in einer anderen Form) referenziert.
Der Stapelparsierer prüft nun anhand eines Abgleichs mit der Tabelle, ob alle Abhängigkeiten einer Klasse erfüllt sind, und lädt gegebenenfalls noch benötigte Klassen nach.

Den rekursiv ablaufenden Vorgang zeigt folgende Methodendefinition:

```
private void batchParse(String fileName,String classPath[])
  throws java.io.IOException {
  BinaryClass currentClass;
  String referencedClasses[];
  // parse a class
  currentClass=parseFile(fileName,classPath);
  // get referenced components of current class
  referencedClasses=currentClass.getReferencedClasses();
  // put loaded class into hashTable
  loadedClasses.put(currentClass.getName(),currentClass);
  // check, if there is need to load a component of
  // the current class
  for(int i=0;i<unresolvedClasses.length;i++) {
    if(loadedClasses.get(referencedClasses[i])==null) {
      // unloaded class found, load it
      batchParse(referencedClasses[i],classPath);
    }
  }
}
```

Abbildung 3-21: Stapelparsierungsvorgang

3.2.4.2 Parsiervorgang

Folgendes Ablaufdiagramm zeigt die Sequenz der einzelnen Vorgänge, die beim Einlesen einer Klassendatei stattfinden. Hierbei spiegeln die Elemente des Diagramms die Elemente der Klassendateistruktur aus 3.2.1 (Seite 50) wider. Das Diagramm gibt zusätzlich die einzelnen Datenstrukturen an, die durch die jeweilige Einleseaktion erzeugt wurden, und durch ein Objekt der Klasse `BinaryClass` aggregiert werden. Diese wird am Ende eines Parsierungsvorgangs durch den Parser zurückgegeben.

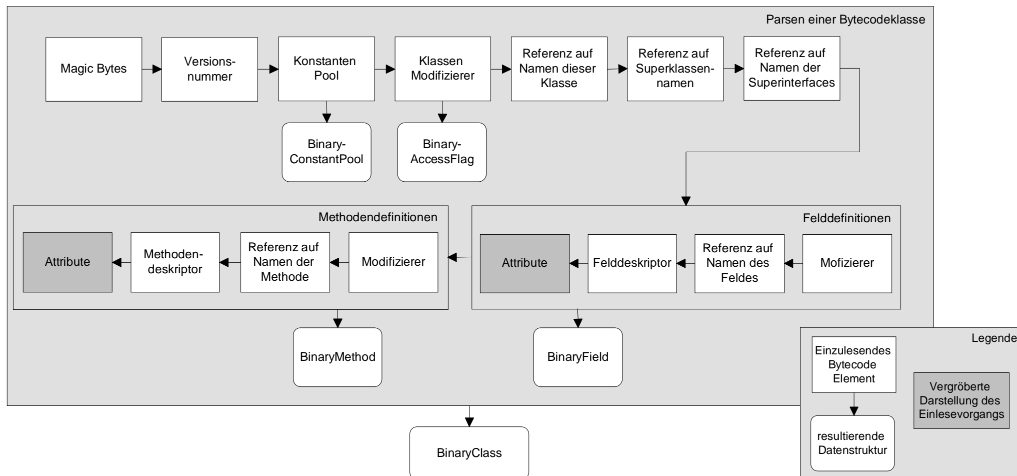

Abbildung 3-22: Aktionen beim Parsieren einer Klasse

Der Ablauf der Parsierung von Attributen ist so umfangreich, daß er im Detail in Abbildung 3-22 keinen Platz mehr findet. Da dieser Vorgang jedoch das Einlesen der essentiellen Daten zur Analyse beinhaltet, wird er in folgendem Ablaufdiagramm eingehend dargestellt.

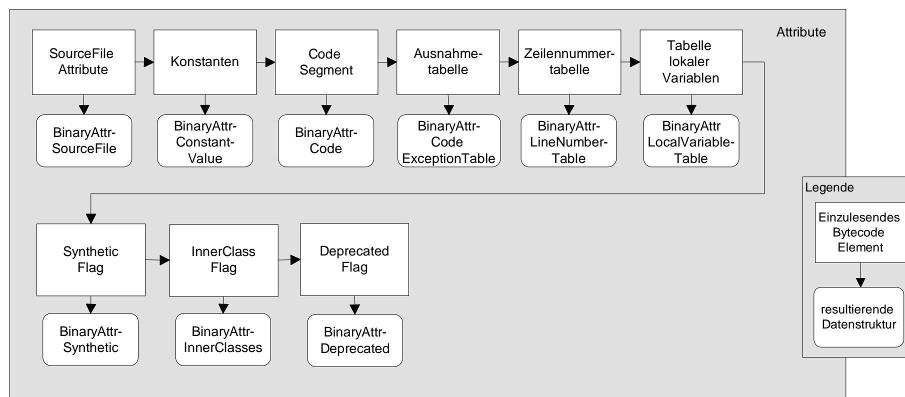

Abbildung 3-23: Aktionen beim Parsieren von Attributen

Am Ende einer erfolgreichen Parsierung steht immer ein Objekt der Klasse `BinaryClass`, daß die vollständige Beschreibung der Klassendatei enthält.

3.2.5 Erzeugung von Analysestrukturen

Die durch den Parsierungsprozeß erzeugten Konstrukte besitzen eine sehr flache Verzweigungshierarchie und benutzen für die Strukturierung lediglich namentliche Referenzierungen. Mit diesen Informationen läßt sich im Hinblick auf eine tiefergehende Analyse schlecht arbeiten. Aus diesem Grund werden die durch den Parsierungsprozeß erzeugten Informationen in eine analysefreundlichere Form gebracht.

3.2.5.1 Bildung von Analysestrukturen

Um eine besser zu handhabende Analysestruktur aus den Parserstrukturen zu erhalten, wird jedes relevante Element aus der Parsierungsstruktur in ein oder mehrere Elemente der Analysestruktur überführt.

Das folgende Diagramm zeigt die einzelnen Elemente der Parserstruktur und die daraus resultierenden Elemente der Analysestruktur.

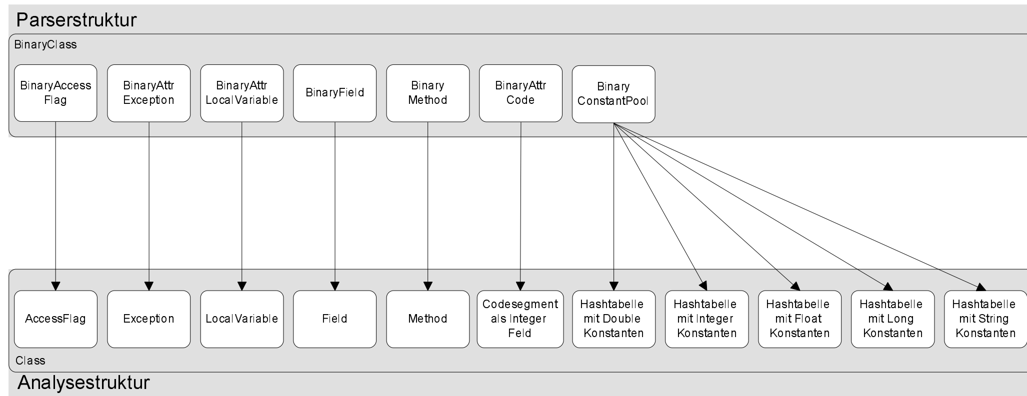

Abbildung 3-24: Bildung von Analysestrukturen

Hierbei ist zu beachten, daß der Konstantenpool in fünf einzelne Hashtabellen übersetzt wird, die jeweils alle Zeichenkettenkonstanten und numerischen Konstanten entsprechend ihres Typs enthalten. Dies ist nötig, da in der Parserstruktur numerische Zeiger auf die Konstanten festgelegt sind, auf die in dem letztendlich zu untersuchenden Codesegment der Klassendatei referenziert wird, so daß diese Information bis zum Abschluß der Analyse des Codesegmentes zur Verfügung stehen muß.

Die einzelnen Elemente, die nach dem Abschluß der Wandlungsphase in Analysestruktur vorliegen, sind wie folgt in Klassen (*Classes*), Methoden (*Methods*) und Attributen (*Fields*) strukturiert:

Element von Method	Kardinalität des Elements
AccessFlag	1
LocalVariable	n
Exception	n
Codesegment	1

Abbildung 3-25: Elemente einer Methode in Analyseform

Element von Field	Kardinalität des Elements
AccessFlag	1

Abbildung 3-26: Elemente eines Feldes in Analyseform

Element von Class	Kardinalität des Elements
AccessFlag	1
Konstantendefinition	jeweils eine Hashtable für Double-, Integer-, Float-, Integer-, Long- und Stringkonstanten
Fields	n
Methods	n

Abbildung 3-27: Elemente einer Klasse in Analyseform

3.2.5.2 Behandlung von Typen

Alle eingelesenen und in Analyseform gebrachten Klassen werden in einer Typentabelle abgelegt. Über diese Typentabelle, in der zunächst alle primitiven Typen (`int`, `byte`, `char`, `float`, `short`, `boolean`, `double`, `long`) abgelegt werden, sind alle benötigten Typeninformationen effizient abrufbar.

Jede gelesene Klassendatei führt einen neuen Typen ein, der die jeweilige Klasse namentlich repräsentiert. Die Felder (Arrays), die innerhalb einer Klassendatei benutzt werden führen ebenfalls neue Typen (Array- Typen) ein.

3.2.5.3 Auflösen der Klassenabhängigkeiten

In der Binärcodedarstellung sind die Beziehungen zwischen den einzelnen Typen der Klassen, Methoden, Attributen und lokalen Variablen lediglich namentlich referenziert. Nach der Umwandlung der Parserstrukturen in die entsprechenden Analysestrukturen werden die namentlich referenzierten Abhängigkeiten in direkte Verknüpfungen über Referenzen umgewandelt. Dieser Vorgang läuft rekursiv über alle geladenen Klassen ab. Dabei wird jede Klasse angewiesen, die Abhängigkeiten für ihre Komponenten aufzulösen.

Die einzelnen Komponenten, die über aufzulösende Beziehungen verfügen, sind im folgenden mit den einzelnen beziehungstragenden Elementen aufgeführt:

- **Klassen**: Typ der Superklasse, Typen der Superinterfaces
- **Klassenattribute**: Typ des Attributes
- **Methoden**: Typ des Rückgabewerts, Typ der Parameter
- **Lokale Variablen**: Typ der lokalen Variable

3.3 Arbeitsweise der Codesegmentanalyse

Dieser Abschnitt beschreibt die Erweiterung der Analysestrukturen um abstrakte Syntaxbäume, die aus dem Codesegment einer Java Klasse hervorgehen. Hierbei wird das Bytcodesegment in jeder Java Klasse, das als Array von ganzzahligen Werten vorliegt gemäß [LiYe97] interpretiert. Der resultierende Syntaxbaum enthält die Informationen, die den Programmablauf festlegen in einer leicht zu verarbeitenden, homogenen Form.

3.3.1 Methoden, spezielle Methoden und lokale Funktionen

Methoden

Die zu untersuchenden Codesegmente befinden sich ausschließlich in Methoden. Zu jeder Methode existiert nach Abschluß der Codesegmentanalyse genau ein Syntaxbaum, wobei die Zugriffsinformationen der Methode nicht im Syntaxbaum, sondern in der Methode selbst gehalten werden.

Zu diesen Informationen gehören unter anderem:

- Methodenname,
- Zugriffsmodifizierer (`public`, `static`, etc.),
- eine Referenz auf den Typen (die Klasse), der die Methode eingeführt hat,
- Zeiger auf lokale Variablen der Methode,
- der Syntaxbaum der Methode,
- Syntaxbäume lokaler Funktionen innerhalb der Methode,
- Anzahl der Parameter und deren Typen, sowie
- der Typ des Rückgabewerts

Spezielle Methoden

Unter *speziellen Methoden* sind Funktionen zu verstehen, die die Abbildung von Eigenschaften eines Java Programms in CSP/FDR realisieren. Diese werden nicht aus dem Bytecode gelesen, sondern sind feste Bestandteile des Übersetzers. Spezielle Methoden

überschreiben eingelesene Methoden. So werden zum Beispiel `java/lang/Thread/start`, `java/lang/Object/wait` und `java/lang/Object/notify` durch spezielle Methoden überschrieben, die die jeweils dafür vorgesehenen Ereignisse in der CSP/FDR Abstraktion ausführen (siehe auch 'Verifizierbares Verhalten von Threads' auf Seite 38).

Folgende spezielle Methoden sind implementiert:

- `java/lang/Thread/start`
- `java/lang/Thread/stop`
- `java/lang/Thread/destroy`
- `java/lang/Thread/join`
- `java/lang/Object/wait`
- `java/lang/Object/notify`
- `java/lang/Object/notifyAll`
- `java/lang/Throwable/<init>`
- `monitorenter`
- `monitorenter`
- `new`

Hierbei ist zu beachten, daß die Monitoroperationen, sowie `new` keine bestehenden Methoden überschreiben, sondern lediglich direkte Bytecodebefehle in Methodenaufrufe umsetzen.

Lokale Funktionen

Innerhalb von Codesegmenten existieren nicht- bedingte Sprünge, die direkt den Programmzähler (*Program Counter*) der virtuellen Java Machine modifizieren, um beispielsweise Schleifen nachzubilden.

Da solche direkten Sprünge innerhalb von CSP/FDR Abstraktionen nicht zulässig und in Syntaxbäumen nicht elegant zu handhaben sind, werden sie in separaten Blöcken gekapselt, die in *lokalen Funktionen* abgelegt werden. Eine genaue Beschreibung des Aufbaus solcher Funktionen ist in "Auflösung von Schleifen" auf Seite 69 beschrieben.

3.3.2 Aufbau von Syntaxbäumen aus Codesegmenten

Dieser Abschnitt entwickelt den aufwendigsten Teil der gesamten Bytecodeverarbeitung, in dem Codesegmente in Syntaxbäume umgewandelt werden. Der Aufbau eines Syntaxbaumes aus einem Codesegment erfordert die Nachbildung eines Java Laufzeitsystems auf abstrakter Ebene. Hierbei soll das Codesegment nicht ausgeführt, sondern in eine abstrakte Form gebracht werden, die sein Verhalten unabhängig von konkreten Variablenbelegungen widerspiegelt. Dieser Vorgang kann als eine Art "Disassemblierung" verstanden werden.

Als Beispiel betrachten wir folgendes Fragment eines Codesegments:

```
0 iload_1  #Inhalt der lokalen Variable #1 auf Stack legen
1 iconst_3 #Integer Konstante 3 auf Stack legen
2 iadd     #Addition der beiden Werte, Ergebnis auf Stack legen
3 iconst_2 #Integer Konstante 2 auf Stack legen
4 imul     #Multiplikation beider Werte, Ergebnis auf Stack legen
5 istore_2 #Ablegen des Ergebnisse in lokaler Variable #2
```

Wie unschwer zur erkennen ist, stellt dieses Fragment eine einfache Addition und Zuweisung der Form `y=(x+3)*2` dar. Als Syntaxbaum einer abstrakten Syntax würde dies folgendermaßen aussehen:

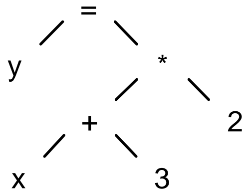

Abbildung 3-28: Fragment eines Codesegments als Syntaxbaum

Ziel der hier beschriebenen Phase ist es nun, eine Transformation des Codesegments einer Methode in eine abstrakte Syntax zu bewerkstelligen.

3.3.2.1 Codesegmentstrukturierung

Um eine effiziente Analyse des Codesegments zu gewährleisten, ist es hilfreich, sich vor Augen zu führen, wie ein Java Compiler das Codesegment erzeugt. Hierbei wird so vorgegangen, daß Verzweigungen in mehreren getrennten Blöcken im Bytecode abgelegt werden. Ein `if- then- else` Konstrukt wird z.B. in Form von zwei Blöcken im Bytecode abgelegt, wobei jeweils der Code, der nach Überprüfung der Bedingung ausgeführt werden kann, in einem einzelnen Block untergebracht ist.

Als Beispiel betrachten wir folgendes Java- Code -Fragment:

```
if(x==1) x++;
else x--;
```

Im Bytecode würde dieses Fragment folgendermaßen aussehen:

```
0 iload_1
1 iconst_1
2 if_icmpne 11
5 iinc 1 1    #true- block (iinc inkrementiert lokale Variable 1)
8 goto 14
11 iinc 1 -1  #false- Block (iinc dekrementiert lokale Variable 1)
14 return
```

In diesem Beispiel sind die Bereiche zwischen Adresse 5 und 8, sowie zwischen 11 und 14 separate Blöcke, da sie jeweils den Code für den true- und den false- Fall enthalten.

Die Analyse eines Blocks läßt sich dabei in folgende Bestandteile (analog zum Abschnitt “Code- Segmente” auf Seite 57) untergliedern:

- **Strukturoperationen**: bedingte Sprünge, Methodenaufrufe und -rückkehr (führen neue Blöcke ein)
- **Sprungoperationen**: nicht bedingte Sprungoperationen (führen neue Blöcke ein)
- **Stackoperationen**: direkte Stackmanipulation (`pop`, `push`, etc.)
- **Operationen auf Referenzen**: Manipulation von Inhalten lokaler Variablen
- **numerische Operationen**: Konvertierungsfunktionen numerischer Typen, numerische und binäre Operationen (`add`, `sub`, `shl`, etc.)
- **Feldoperationen**: Manipulation der Inhalte von Feldern (Arrays)
- **Attributoperationen**: Manipulation von Attributinhalten

3.3.2.2 Auflösung von Schleifen

Schleifen werden im Java Bytecode durch Sprungoperationen modelliert. Hierbei besteht beispielsweise eine Endlosschleife der Form `while(true){}` aus einem nicht bedingten Sprung: `0 goto 0`.

Eine for- Schleife der Form `for(int i=0;i<10;i++){}` wird folgendermaßen abgelegt:

```
 0 iconst_0    # Konstante 0 auf Stack legen
 1 istore_2    # erstes Stackelement in lokale Var. #2 speichern
 2 goto 8      # Sprung
 5 iinc 2 1    # Inhalt der lokalen Var.#2 inkrementieren
 8 iload_2     # Inhalt der lokalen Var. #2 auf Stack legen
 9 bipush 10   # Konstante 10 auf den Stack legen
11 if_icmplt 5 # Bedingter Sprung
```

Das Problem bei der Analyse von Schleifen liegt darin, daß herausgefunden werden muß, ob die Analyse endlos in einem Codefragment hängen bleiben kann, wenn es sich um eine Endlosschleife handelt.

Aus diesem Grund wird jeder Block mit seiner Startadresse in einer Tabelle abgelegt. Wird nun eine Sprungoperation an eine Adresse ausgeführt, wird ermittelt, ob zu dieser Adresse bereits ein Block innerhalb der Tabelle existiert. Ist dies nicht der Fall, wird der Block analysiert und in die Tabelle eingetragen. Andernfalls wird lediglich ein Verweis auf den anzuspringenden Block angelegt.

Das vorgestellte Beispiel würde also folgende Blockstruktur entwickeln:

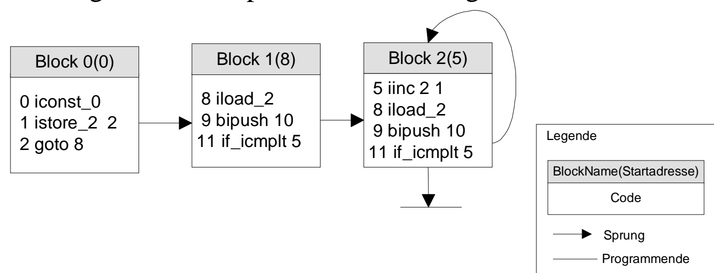

Abbildung 3-29: Blockstruktur

3.3.2.3 Laufzeitstrukturen

Dieser Abschnitt stellt die Datenstrukturen vor, mit denen das Java Laufzeitsystem für die Codesegmentanalyse simuliert wird.

Typen

Alle Elemente, die während der Laufzeitanalyse benutzt werden sind getypt. Hierbei gibt es die folgenden drei Arten von Typen:

- **PrimitiveType**: Eine Instanz dieser Typgruppe bezeichnet jeweils einen primitiven Java Datentypen. Hierfür kommen `byte`, `short`, `int`, `long`, `float`, `double` oder `boolean` in Frage.
- **Type**: Unter diese Kategorie fallen alle im Java- Bytecode definierten Klassen. Jede Klasse führt hierbei einen neuen Typ ein.
- **ArrayType**: Arraytypen legen die Dimensionierung eines Feldes und dessen Elementtyp fest. Elementtypen von Feldtypen (ArrayTypes) können wiederum von jeder beliebigen Typart sein. Mehrdimensionale Felder können so durch eine Verschachtelung von Arraytypen dargestellt werden.

Operandenstack

Wie bereits im Abschnitt “Laufzeitstrukturen” auf Seite 56 erläutert, arbeitet die virtuelle Java Maschine stackorientiert, weshalb es nötig ist, einen Keller zu implementieren, der die selben Eigenschaften hat wie der einer Virtual Machine. Die Implementierung einer virtuellen Java Machine benutzt einen nicht getypten Stack, auf dem sich jeweils Worte befinden. Diese Worte stellen entweder primitive Werte dar oder beinhalten Referenzen auf komplexere Daten.

Der hier benutzte Operandenstack benutzt im Gegensatz dazu eine getypte Struktur, die nur Referenzen enthält, aus denen die jeweiligen Nutzdaten direkt extrahiert werden können.

Hierbei ist zu beachten, daß dadurch die Einträge auf dem Stack anders organisiert sind als bei der Verwendung eines ungetypten Stacks. So belegt bei der nicht getypten Version die Ablage eines Wertes des Datentyps `long` oder `double` jeweils zwei Wörter des Operandenstacks, während die getypte Version nur einen Eintrag zur Ablage einer Referenz benutzt, die den entsprechenden `long`- oder `double`- Wert beinhaltet.

Aus diesem Grund wurden die Operationen zur direkten Stackmanipulation (`pop`, `dup`, etc.) entsprechend angepaßt. Folgendes Beispiel, bei dem eine `double`- Konstante vom Stack genommen wird, soll dies verdeutlichen:

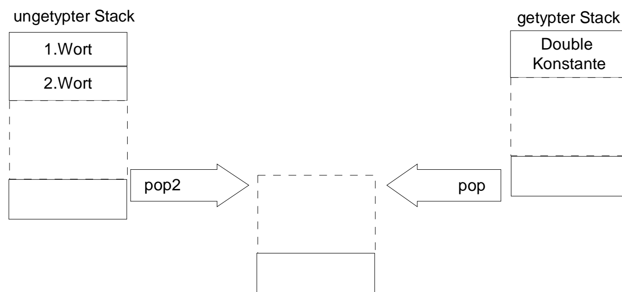

Abbildung 3-30: Getypte und nicht- getypte Stacks

Referenzen und Konstanten des Operandenstacks

Während des Ablaufs der Laufzeitanalyse werden unterschiedliche Konstrukte zur Belegung des Operandenstacks benutzt. Diese Elemente sind analog zu denen in “Code- Segmente” auf Seite 57 aufgeführten. Sie lassen sich in Konstanten und Referenzen unterteilen:

- **PrimitiveConstant**: Konstante vom Typ `byte`, `short`, `int`, `long`, `float`, `double` oder `boolean`.
- **ObjectConstant**: Konstante eines beliebigen nicht primitiven Typen; dies beinhaltet ebenfalls Feld- (Array-) Typen
- **ArrayRef**: Referenz auf ein Feld (Array) beliebigen Typs.
- **ObjectRef**: Referenz auf beliebige nicht- Feld- Typen. Die Objekte, die dieses Element referenzieren kann sind entweder vom Typ String oder beziehen sich auf Klassen, die primitive oder zusammengesetzte Typen aggregieren.
- **PrimitiveRef**: Referenz auf einen Wert primitiven Typs (`byte`, `short`, `int`, `long`, `float`, `double` und `boolean`).
- **ReturnAddress**: Spezielle Referenz, die vom Laufzeitsystem für Rücksprünge verwendet wird.

`ArrayRef`, `ObjectRef`, `PrimitiveRef`, sowie `ObjectConstant` und `PrimitiveConstant` lassen sich jeweils in Elemente der abstrakten Syntax umwandeln, wobei aus den Referenzen Syntaxbaumelemente vom Typ `ASTVariable` und aus Konstanten Syntaxbaumelemente vom Typ `ASTConstant` erzeugt werden (siehe ’Abstrakte Syntax’ auf Seite 72).

3.3.2.4 Abstrakte Syntax

Nachdem wir nun die Struktur des Codesegmentes und seinen logischen Aufbau betrachtet haben, ist es an der Zeit, die abstrakte Syntax festzulegen, in die das Codesegment überführt werden soll.

Das folgende Diagramm zeigt die einzelnen Elemente der abstrakten Syntax und deren Beziehungen untereinander.

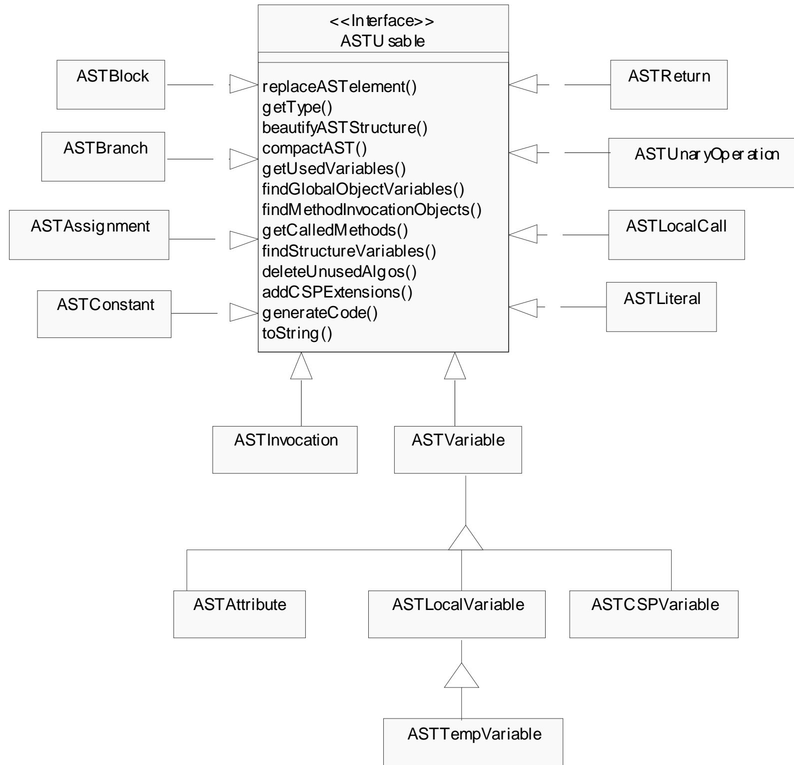

Abbildung 3-31: Klassenabhängigkeiten der abstrakten Syntax

Hierbei ist `ASTUsable` ein Interface, daß angibt, welche Methoden durch alle Elemente der abstrakten Syntax zu implementieren sind.

Betrachten wir nun die einzelnen Elemente der abstrakten Syntax detailliert:

Name	Funktion	Unterelemente und Elementtyp
`ASTAssignment`	Zuweisung	Zielvariable (`ASTVariable`), Quelle (`ASTUsable`)
`ASTAttribute`	Attributvariable	keine
`ASTBlock`	Block (Menge von Opertionen)	Liste von Operationen (`ASTUsable`)
`ASTBranch`	Bedingte Verzweigung	Bedingung (`ASTDyadicOperation`), True- Zweig (`ASTBlock`), False- Zeig (`ASTBlock`)
`ASTConstant`	Konstante Werte	keine
`ASTCSPVariable`	CSP Hilfsvariable	keine
`ASTDyadicOperation`	Dyadische Operation	erstes Argument (`ASTUsable`), zweites Argument (`ASTUsable`)
`ASTInvocation`	Methodenaufruf	aufzurufende Methode, Objektvariable (`ASTVariable`), Liste von Argumenten (`ASTUsable`)
`ASTLiteral`	Einfaches Literal (Zeichenkette oder Zahlenwert)	keine
`ASTLocalCall`	Aufruf lokaler Funktion	aufzurufende lokale Funktion
`ASTLocalVariable`	Lokale Variable	keine
`ASTReturn`	Methodenrücksprung	Rückgabewert (`ASTUsable`)
`ASTTempVariable`	Temporäre Hilfsvariable (wird im Laufe der Analyse eliminiert)	keine
`ASTUnaryOperation`	Unäre Operation	Argument (`ASTUsable`)

Abbildung 3-32: Elemente der abstrakten Syntax

Aus den Abhängigkeiten der einzelnen Elemente der abstrakten Syntax läßt sich folgende Grammatik entwickeln:

ASTAssignment:
    ASTVariable ASTUsable
ASTBlock:
    *ASTUsable**
ASTBranch:
    DyadicOperation ASTBlock ASTBlock
ASTDyadic:
    ASTUsable ASTUsable **OperationType**
ASTInvocation:
    Method [*ASTVariable*] *ASTUsable**
ASTLocalCall:
    LocalFunction
ASTReturn:
    ASTUsable
ASTUnary:
    ASTUsable

ASTUsable:
    ASTAssignment
    ASTBlock
    ASTBranch
    ASTConstant
    ASTDyadicOperation
    ASTInvocation
    ASTLiteral
    ASTLocalCall
    ASTReturn
    ASTUnaryOperation
    ASTVariable
ASTVariable:
    ASTCSPVariable
    ASTLocalVariable
    ASTTempVariable
    ASTAttribute

Abbildung 3-33: Abhängigkeiten der abstrakten Syntax als Grammatik

Variablen

Wie leicht an den verschiedenen Arten von Variablen (`ASTAttribute`, `ASTLocalVariable` `ASTTempVariable` und `ASTCSPVariable`) zu erkennen ist, stellt diese einen besonders wichtiges Element der abstrakten Syntax dar.

`ASTVariable` stellt die Basisklasse für alle Variablen (Attribute, lokale Variablen und CSP- Variablen) dar, ist aber niemals direkter Bestandteil eines abstrakten Syntaxbaumes. Sie beinhaltet lediglich alle Methoden, die für die einzelnen Variablentypen dieselben sind. Die einzelnen Variablenarten im Syntaxbaum und deren Eigenschaften sind im folgenden aufgeführt:

- **`ASTLocalVariable`**: Die Abbildung von lokalen Variablen in der abstrakten Syntax wird mittels `ASTLocalVariable` bewerkstelligt. Lokale Variablen zeichnet aus, daß sie an die Methoden gebunden sind, in denen sie definiert wurden.
- **`ASTAttribute`** ist analog zu `ASTLocalVariable` an eine Klasse gebunden. Alle Methoden einer Klasse können auf das Attribut zugreifen. Außerdem kann von außen auf ein Attribut zugegriffen werden, wenn die Attributmodifizierer (siehe ’Struktur des Java Class File Modells’ auf Seite 51) entsprechend gesetzt sind.
- **`ASTCSPVariable`** ist ein Syntaxbaumelementtyp, der erst nach der Analysephase eingeführt wird, da er seitens des Java Bytecodes nicht existent ist. Das Auslesen und Zuweisen von Werten kann in CSP/FDR nicht direkt über Variablenreferenzen realisiert werden. Vielmehr ist es hierzu nötig, spezielle Variablen zu definieren, die die Werte der Kanäle in einem Zustand beinhalten. Diese Variablen werden durch Elemente des Typs `ASTCSPVariable` dargestellt.
- **`ASTTempVar`**: Im Codesegment des Java Bytecodes existieren zur Realisierung von Vergleichen und arithmethischen Berechnungen lediglich unäre und dyadische Operationen, die nicht geschachtelt werden dürfen. Innerhalb des Codesegments wird dabei so vorgegangen, daß jeweils eine Operation durchgeführt und das Ergebnis als Referenz auf den Stack gelegt wird. Anschließend wird mit den jeweiligen Ergebnis weitergearbeitet. Die Ergebnisreferenzen sind an keine im Java Bytecode definierten Variablen gebunden, so daß sie im abstrakten Syntaxbaum durch einen anonyme (temporäre Variable) repräsentiert werden.
  Da das Vorhandensein von temporären Variablen zu einer nicht sehr performanten (da ungeschachtelten) Struktur des Syntaxbaumes führt, ist es sinnvoll, den Baum nach dessen Aufbau von temporären Variablen zu befreien. Dieses vorgehen wird in Abschnitt “Syntaxbaumoptimierung” auf Seite 76 erläutert.

3.3.2.5 Abbildung von Referenzen auf Variablen im Syntaxbaum

Jede Referenz ist während der Bytecodeanalyse an eine Variable gebunden. Dies ist wichtig, da der aufgebaute Syntaxbaum nur Variablen und keine Stackreferenzen enthalten darf. Die Variablen gehen direkt aus den Referenzen der Bytecodeanalyse hervor. Würde eine Referenz keiner Variablen zugeordnet sein, würde dies zum Fehlen von Syntaxbaumelementen führen.
Als Variablenelemente im Syntaxbaum kommen `ASTAttribute`, `ASTLocalVariable` und `ASTTempVariable` in Frage. Hierbei ist darauf zu achten, daß eine Referenz mehreren Variablen zugewiesen sein kann.
Für Abhängigkeiten von Referenzen zu Variablen gelten folgende Regeln:

- Wird eine Referenz neu erzeugt, ist sie an eine neu erzeugte temporäre Variable gebunden.
- Wird eine Referenz einer lokalen Variablen oder einem Attribut zugewiesen, wird eine Kopie dieser Referenz erstellt, die an die Variable oder das Attribut gebunden ist.

3.3.2.6 Abbildung der Codesegmentoperationen auf Elemente der abstrakten Syntax

Nachdem wir nun die Elemente kennen, die innerhalb der Laufzeitanalyse und des Syntaxbaumes benutzt werden, ist es nun an der Zeit zu definieren, welche Operationen im Bytecode zur Einführung welcher Elemente im Syntaxbaum führen.

Methodenaufruf/ -rückehr

Jeder Methodenaufruf über die Bytecodeoperationen `INVOKEVIRTUAL`, `INVOKESPECIAL`, `INVOKESTATIC` und `INVOKEINTERFACE` führt ein Syntaxbaumelement vom Typ `ASTInvocation` ein, dem die aufzurufende Methode und die zu übergebenden Parameter zugeordnet sind. Ist die aufzurufende Methode nicht statisch, muß zusätzlich noch das Objekt angegeben werden, auf dem die Methode ausgeführt wird (*this- reference*).

Gibt eine Methode einen Wert zurück, so wird die Rückgabe innerhalb des Java Bytecodes durch die Operation `RETURN` durchgeführt. Diese Operation führt ein Objekt des AST- Elementes `ASTReturn` in den Syntaxbaum ein.

Monitorenter/exit

Die zum gegenseitigen Ausschluß auf kritische Abschnitte vorgesehenen Bytecodeoperationen werden durch den Aufruf einer entsprechenden *speziellen Methode* (über `ASTInvocation`) im Syntaxbaum abgebildet. Das System stellt hierfür eine Reihe von *speziellen Methoden* (siehe 'Methoden, spezielle Methoden und lokale Funktionen' auf Seite 67) zur Verfügung, die das jeweilige Verhalten direkt in CSP/FDR Notation darstellen.

Synchronized Methoden

Methoden, die mit dem Qualifizierer `synchronized` definiert wurden werden so behandelt, als würde zu Beginn der Methode ein `MONITORENTER` und vor dem Rücksprung ein `MONITOREXIT` auf die aktuelle Instanz aufgerufen werden (siehe 'Verwendung des synchronized Schlüsselworts' auf Seite 21).
Im Bytecode sind diese Aufrufe jedoch nicht explizit eingebracht, so daß jeweils ein Element `ASTMonitorEnter` zu Beginn und `ASTMonitorExit` vor dem Rücksprung aus einer synchronisierten Methode dem Syntaxbaum zugefügt werden.

Bedingte Sprünge

Jeder bedingte Sprung führt eine Element vom Typ `ASTBranch` in den Syntaxbaum ein, wobei die jeweilige Bedingung durch eine dyadische Operation (`ASTDyadicOperation`) mit den Vergleichsoperationen ausgedrückt wird. Die jeweiligen Codesegmente für den true- und false- Fall werden durch jeweils ein Element des Typs `ASTBlock` dargestellt.

Numerische Operationen

Bei den numerischen Operationen kann zwischen dyadischen und unären Operationen unterschieden werden.

Die Funktion `NEG` im Java Bytecode ist die einzige unäre, so daß der Aufruf dieser Funktion ein Syntaxbaumelement des Typs `ASTUnaryOperation` zufügt.

Alle anderen numerischen Operationen sind dyadisch, so daß hier ein Element des Typs `ASTDyadicOperation` in den Syntaxbaum eingefügt wird.

Damit eine Beziehung zwischen den Operationen und den späteren Zuweisungen an Variablen bewerkstelligt werden kann, werden alle unären und dyadischen AST- Elemente mittels einer Zuweisung (`ASTAssignment`) an eine neu generierte temporäre Variable gebunden.

STORE, PUTFIELD und PUTSTATIC- Operationen

Diese Operationen führen ein neues Zuweisungselement (`ASTAssignment`) in den Syntaxbaum ein. Das Ziel dieser Zuweisung ist dabei die lokale Variable (bei `STORE`) oder das Attribut (bei `PUTFIELD` oder `PUTSTATIC`), dem der aktuelle Wert des Stacks zugewiesen werden soll. Die Quelle ist entweder eine Variable (`ASTVariable`) oder eine Konstante (`ASTConstant`), die an die Referenz des obersten Stackelements gebunden ist.

LDC/LDC_W/LDC2_W

Diese Bytecodeoperationen dienen dazu, Konstanten aus dem Konstantenpool des Bytecodes direkt auf den Stack zu legen. Diese Konstanten können primitiv oder vom Typ String sein.

Alle drei Operationen führen dazu, daß ein neues Objekt des Syntaxbaumelementtyps `ASTConstant` mit dem entsprechenden Typ und dem Wert aus dem Konstantenpool erzeugt wird.

NEW, NEWARRAY, ANEWARRAY

Die `NEW`- Operationen führen jeweils eine neue Instanz von `ASTTempvariable` ein. Auf diese Variable wird anschließend ein Objekt der Klasse `ASTInvocation` in den Syntaxbaum eingebracht, die auf die *spezielle Methode* für `NEW` zeigt.

ARRAYLENGTH

Die Operation `ArrayLength` führt eine String- Konstante (`ASTConstant`) mit dem Wert `MAX_ARRAY_LENGTH` ein. Dies ist dadurch bedingt, daß diese Funktion nicht direkt in CSP/FDR übesetzt werden kann und die Länge von Feldern in CSP/FDR Abstraktionen durch diese Textkonstante festgelegt wird (siehe 'Abstraktionsmodell' auf Seite 43).

3.3.2.7 Syntaxbaumoptimierung

Wie in “Code- Segmente” auf Seite 57 erläutert, existieren in einem Bytecodesegment nur unäre und dyadische Operationen, die auf Referenzen arbeiten. Diese Referenzen dürfen nicht geschachtelt werden, so daß für eine geschachtelte Operation temporäre Referenzen, die von temporären Variablen gehalten werden, angelegt werden (siehe auch 'Abbildung von Referenzen auf Variablen im Syntaxbaum' auf Seite 74). Dies führt dazu, daß z.B. die Berechnung des Terms `y=(x+3)*2` in zwei Schritte unterteilt wird:

```
1) tempVar=(x+3)
2) y=tempVar*2
```

Innerhalb des Syntaxbaumes sind solche temporären Variablen, die kein äquivalent im Java Quellcode haben, unerwünscht.

Aus diesem Grund wird bei der Erstellung eines Syntaxbaumelements für eine Zuweisung, dessen Ziel eine temporäre Variable ist, das zuzuweisende Element in einer Hashtabelle abgelegt. Als Schlüssel dient hierbei die temporäre Variable, als Nutzdaten das Element, das zugewiesen werden soll.

Der so entstandene Nutzdatenanteil kann wiederum ebenfalls temporäre Variablen enthalten, so daß nach Abschluß der Syntaxbaumerstellung innerhalb der Tabelle alle temporären Variablen in den Nutzdatenanteilen rekursiv durch die zugehörigen temporärvariablen- freien Terme ersetzt werden.

Im Anschluß daran werden alle temporären Variablen im Syntaxbaum durch die entsprechenden Temporärvariablen- freien Syntaxbaumelemente ersetzt.

Hierdurch wird für unser Beispiel die folgende Struktur im Syntaxbaum durch eine optimierte Version ersetzt, die sich in Abbildung 3-28, „Fragment eines Codesegments als Syntaxbaum,“ auf Seite 69 befindet.

Abbildung 3-34: Syntaxbaum vor der Syntaxbaumoptimierung

3.3.3 Syntaxbaumanalyse

Die bisherigen Schritte der Übersetzung dienten lediglich dazu, ein eingelesenes Programm in eine gut zu handhabende Form (den Syntaxbaum) zu bringen. Nun ist es an der Zeit, die eigentlich für die Übersetzung interessanten Aspekte (siehe ’Auswertung paralleler Java Programme’ auf Seite 40) der Implementierung zu untersuchen.

3.3.3.1 Ermittlung von Thread- und Synchronisationsvariablen im Syntaxbaum

Threadvariablen

Für die Ermittlung der verwendeten Kontrollfäden wurde ein Algorithmus analog dem in “Ermittlung von Kontrollfäden” auf Seite 40 vorgestellten Vorgehen implementiert. Hierzu wird eine Liste von Start- Methoden und Threadvariablen geführt. Die Liste der Startmethoden ist initial mit der Methode `java/lang/Thread/start()` belegt.

Daraufhin werden die Syntaxbäume der einzelnen Methoden auf Aufrufe (`ASTInvocation`) der Start- Methoden untersucht. Ruft eine Methode eine Startmethode auf einem ihrer Parameter aus, wird sie in die Methodenliste eingetragen. Ruft sie eine Start- Methode mit einer Variable auf, die nicht als Parameter deklariert wurde, wird diese Variable in die Threadvariablenliste eingetragen. Der resultierende Algorithmus sieht folgendermaßen aus:

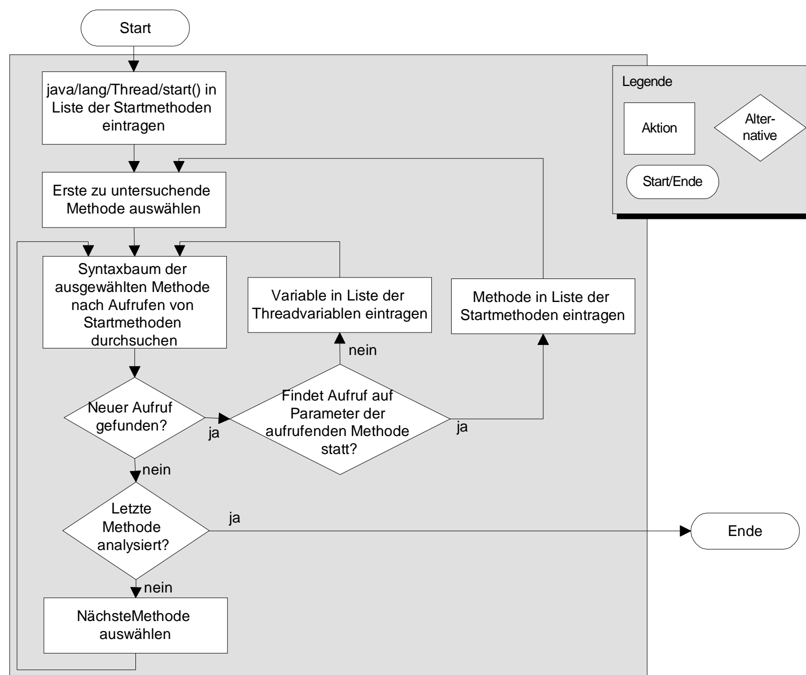

Abbildung 3-35: Algorithmus zur Ermittlung von Kontrollfäden im Syntaxbaum

Synchronisationsvariablen

Analog zum Auffinden der Threadvariablen wird die Ermittlung von Synchronisationsvariablen realisiert. Synchronisationsvariablen sind hierbei die Variablen, auf die die Methoden `java.lang.Object.wait()`, `java.lang.Object.notify()`, sowie `MONITORENTER` und `MONITOREXIT` aufgerufen werden. Aus diesem Grund kann der oben angegebene Algorithmus genutzt werden, wobei statt der Start- Methode, die Synchronisationsmethoden angegeben werden.

3.3.3.2 Ermittlung von Strukturvariablen im Syntaxbaum

Die Ermittlung von Strukturvariablen wurde ebenfalls durch eine Analyse der Syntaxbäume der eingelesen Methoden realisiert. Der im Abschnitt “Ermittlung benötigter Strukturvariablen” auf Seite 42 beschriebene Algorithmus wird dabei durch einen rekursiven

Abstieg in den Syntaxbaum der Methoden realisiert. Jedes Element wird daraufhin auf seine Abhängigkeit von Thread- und Synchronisationsvariablen untersucht. Die Bedingungen für die Übernahme der einzelnen im Syntaxbaum benutzten Variablen in die Liste der Strukturvariablen ist in folgender Aufzählung, geordnet nach Syntaxbaumelementen, dargestellt:

- **`ASTAssignment`**: Ist das Ziel der Zuweisung eine Thread-, Synchronisations- oder Strukturvariable, so ist die Quelle eine Strukturvariable.
- **`ASTBranch`**: Beinhaltet einer der beiden Ereignisblöcke der Verzweigung Thread-, Synchronisations- oder Strukturvariablen, so sind die in der Bedingung benutzten Variablen Strukturvariablen.
- **`ASTInvocation`**: Wird ein Parameter eines Methodenaufrufs mit einer Thread- , Synchronisations- oder Strukturvariable versehen, so ist der Parameter eine Strukturvariable.
- **`ASTReturn`**: Wird eine Variable zurückgegeben, so ist diese eine Strukturvariable.

Wurden alle Syntaxbäume nach diesem Verfahren behandelt, liegt die komplette Menge der Strukturvariablen vor.

3.3.3.3 Löschen nicht benötigter Algorithmen im Syntaxbaum

Nachdem alle relevanten Variablen ermittelt wurden, können die nicht benötigten Algorithmen aus den Syntaxbäumen gelöscht werden. Hierbei ist zu beachten, daß der JAVA2CSP- Übersetzer darüber hinaus die Möglichkeit gibt, unerwünschte Klassen zu definieren, die nicht in der zu erzeugenden CSP/ FDR Spezifikation auftauchen, um die Spezifikation möglichst effizient zu gestalten (siehe Anhang A). Die Verwendung unerwünschter Klassen muß ebenfalls durch diesen Analyseschritt berücksichtigt werden.

Das Vorgehen hierzu kann in drei Teile aufgeteilt werden:

- Jedes Syntaxbaumelement, das ausschließlich Variablen benutzt, die weder Thread- noch Synchronisations-, oder Strukturvariablen sind, wird entfernt.
- Alle Variablen und Methoden, die vom Typ unerwünschter Klassen sind oder in unerwünschten Klassen definiert wurden, werden gelöscht.
- Alle Methoden, deren Syntaxbaum nach Ausführung der ersten beiden Schritte leer ist, werden entfernt.

Im Anschluß an diesen Analyseschritt sind nur noch für die Übersetzung relevante Methoden und relevante Elemente in den Syntaxbäumen der Methoden vorhanden.

3.4 Codegenerierung

In diesem Abschnitt wird die Erzeugung einer CSP/ FDR- Spezifikation aus den erzeugten Analysestrukturen vorgestellt.

3.4.1 Erweiterung des Syntaxbaumes um CSP Elemente

Die nach der Übersetzung vorliegenden Syntaxbäume enthalten alle Informationen des Kontrollflusses der einzelnen Methoden im Format des Ausgangscodes. So sind beispielsweise Zuweisungen der Form `dest=src` vorhanden, die in dieser Form nicht direkt in CSP übersetzt werden können, da in CSP nur Kanäle zur Speicherung von Werten benutzt werden können. Aus dem Quellkanal müssen die Werte zunächst über den Umweg einer CSP- Variable ausgelesen und dann in den Zielkanal übertragen werden (siehe auch ’Kommunikation in CSP’ auf Seite 28). Dies würde für unser Beispiel folgendermaßen aussehen: `src?x -> dest!x`.

Des weiteren werden die konkreten Belegungen von Kanälen für Vergleiche, dyadische und unäre Operationen sowie Methodenaufrufen mit primitiven Parametern benutzt. Dies erfordert die Einführung von CSP- Variablen in den Syntaxbaum (`ASTCSPVariable`).

Die Implementierung der Erweiterung des Syntaxbaums um CSP- Elemente ist dabei als Rekursion über alle Syntaxbaumelemente realisiert. Jede Klasse der abstrakten Syntax besitzt dabei eine Methode, die sich und alle Kindelemente der jeweiligen Instanz um CSP-Elemente erweitert.

Im folgenden entwickeln wir die CSP- Erweiterungen, geordnet nach betroffenen Syntaxbaumelementen.

ASTAssignment

Wird innerhalb einer Zuweisung direkt auf eine Variable zugegriffen, wird die ursprüngliche Zuweisung durch zwei Zuweisungen ersetzt, wobei eine neue CSP- Variable erzeugt wird, die erste Zuweisung den Quellkanal in eine CSP- Variable liest und die zweite den Wert der CSP- Variablen in den Zielkanal schreibt.

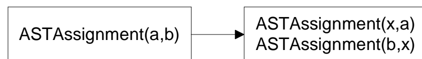

Abbildung 3-36: CSP Variablen in Zuweisungen

ASTDyadicOperation/ ASTUnaryOperation

Eine dyadische oder unäre Operation kann direkte Zugriffe auf Variablen enthalten. In diesem Fall wird eine neue CSP- Variable erzeugt, dem der Wert der entsprechenden Variable zugewiesen wird. Dies kann bei dyadischen Operationen für beide Elemente gelten. Für eine dyadische Operation, die zwei Variablen vergleicht, würde die Transformation folgendermaßen aussehen:

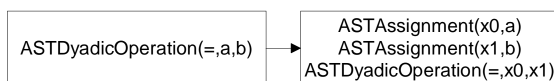

Abbildung 3-37: CSP Variablen bei dyadischen Operationen

ASTInvocation

Alle primitiven Parameter eines Methodenaufrufs werden by- value übergeben. Aus diesem Grund muß jeder primitive Variablenwert, der bei einem Methodenaufruf übergeben wird, in eine CSP- Variable geschrieben werden.
Für den Aufruf einer Methode mit zwei primitiven Parametern ergibt sich somit folgende Transformation:

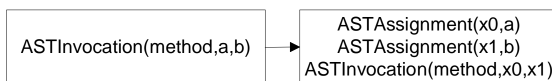

Abbildung 3-38: CSP Variablen bei Methodenaufrufen

3.4.2 Abkürzen von Bezeichnern

Die in der Spezifikation verwendeten Bezeichner für Variablen-, Attribut-, und Methodennamen sind in der Regel sehr lang, da sie jeweils voll qualifiziert sind. Diese Form der Ausgabe ist zwar für die maschinelle Bearbeitung der Abstraktion nicht von Nachteil, jedoch sind lange Bezeichner für einen menschlichen Leser nicht sonderlich intuitiv, um die Bezeichner in der Abstraktion im Java- Quellcode wiederzufinden.

Um hier Abhilfe zu schaffen, werden alle verwendeten Bezeichner soweit wie möglich gekürzt. Hierbei ist darauf zu achten, daß keine zwei Objekte den selben Bezeichner besitzen und das kein Bezeichner für Variablen-, Attribut-, und Methodennamen mit einem vordefinierten Bezeichner des Übersetzungssystems (siehe Anhang B) kollidiert.

Der hierfür angewendete Algorithmus sieht folgendermaßen aus:

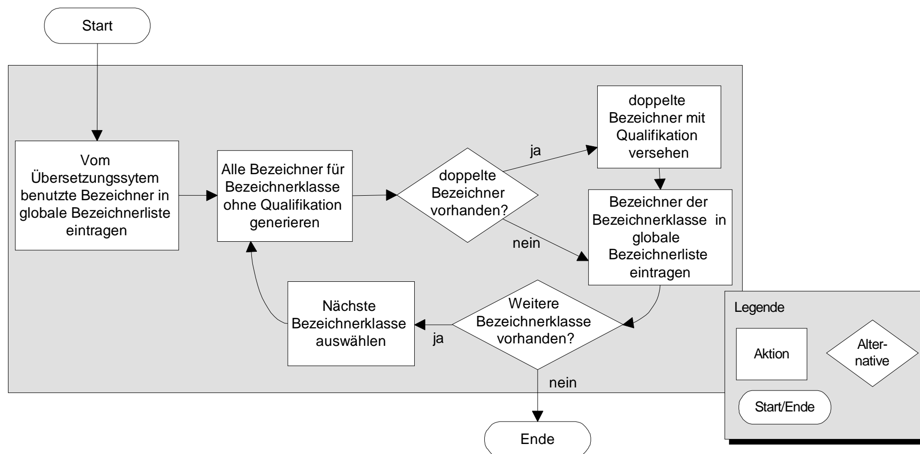

Abbildung 3-39: Algorithmus zur Kürzung von Bezeichnern

Als Bezeichnerklassen kommen lokale Variablen, Klassen, Methoden und Attribute in Frage. Die folgende Tabelle zeigt, womit die einzelnen Bezeichnerklassen qualifiziert werden können:

Bezeichnerklasse	Qualifizierende Bezeichner
Attribute	Paketname, Klassenname
Klassen	Paketname
Lokale Variablen	Paketname, Klassenname, Methodenname
Methoden	Paketname, Klassenname

Abbildung 3-40: Qualifizierung von Bezeichnerklassen

3.4.3 Nachbildung arithmetischer Funktionen

Der Java Bytecode verfügt über eine Reihe von arithmetischen Funktionen, die nicht direkt durch FDR bereitgestellt werden. Hierzu gehören das bitweise AND, OR und XOR sowie die Funktionen für Linksshift, Rechtsshift und Rechtsschift ohne Vorzeichenübertrag (siehe Abbildung 3-9, „Arithmethische Befehle im Java Class File Format," auf Seite 58). Die jeweiligen Algorithmen und Implementierungen zur Nachbildung dieser Funktionen werden im folgenden erläutert. Diese Funktionen sind Bestandteile der JAVA2CSP- Bibliothek, die in Anhang C auf Seite 105 aufgeführt ist.

Für die Nachbildung von `AND`, `OR` und `XOR` werden die jeweiligen Zahlenwerte zunächst in Sequenzen von binären Werten umgesetzt, wobei das erste Bit einer Sequenz jeweils das Vorzeichen beinhaltet (positiv=0, negativ=1). Analog dazu existiert eine Funktion, die die Binärcodierung wieder in einen Zahlenwert umsetzt.

- **Linksshift**: Die Operation "Linksshift um s Positionen" kann sehr einfach durch eine Multiplikation mit 2^s bewerkstelligt werden.
- **Rechtsshift**: Analog zum Linksshift läßt sich der Rechtsshift durch eine Division durch 2^s realisieren.
- **Rechtsshift ohne Vorzeichenübertrag**: Der Rechtsshift ohne Vorzeichenübertrag läßt sich durch die Neutralisierung des Vorzeichens nach der Ausführung der schon eingeführten Rechtsshiftoperation nachbilden.
- **Bitweises ODER**: Das bitweise ODER wird als "Veroderung" der jeweiligen Elemente zweier Sequenzen der Binärdarstellung der jeweiligen Zahlenwerte bewerkstelligt. Ist eine Sequenz länger als die andere, wird der Übertrag an die Ergebnissequenz angehängt, da dies das bitweise ODER auf die einzelnen Elemente der Restsequenz mit 0 darstellt.

- **Bitweises UND**: Das bitweise UND ist analog zum bitweisen ODER implementiert. Etwaige Restsequenzen, die durch unterschiedlich lange Bitsequenzen entstehen, werden eliminiert, da dies der "Verundung" mit 0 entspricht.
- **Bitweises XOR**: Die Funktionsweise der Nachbildung des bitweisen XOR ist analog zum bitweisen OR. Wenn eine Sequenz von Binärwerten länger ist als die andere, wird auf jedes Element der Restsequenz die XOR Operation mit 0 angewendet. Die XOR- Operation auf zwei Bitwerten ist dabei als modulo 2 Addition definiert.

3.4.4 Erzeugung einer textuellen Repräsentation

Nachdem wir nun die Transformationsregeln und die daraus resultierende Traversierung der Syntaxbäume eingeführt haben, läßt sich aus den vorliegenden Informationen eine FDR/CSP Spezifikation in ein textuelles Format für die direkte Verwendung durch FDR2 erzeugen.
Das hier vorgestellte vorgehen implementiert dabei genau das Abstraktionsmodell aus Abschnitt 2.6.
Die in der Repräsentation enthaltenen Elemente und deren Reihenfolge zeigt die folgende Abbildung:

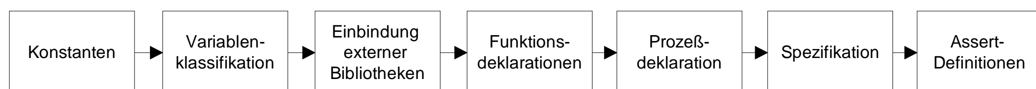

Abbildung 3-41: Struktur der Codegenerierung

3.4.4.1 Generierung von Konstanten

In der erzeugten Spezifikation werden eine Reihe von Konstanten verwendet, die dazu dienen, den Experimentraum einzugrenzen, um eine möglichst effiziente Spezifikation zu erhalten. Die einzelnen Konstantentypen, deren Bezeichner und Bedeutung werden im folgenden erläutert.

Bereich numerischer Werte

Der Spezifikation werden die Konstanten `MIN_VAL_NUMBER` und `MAX_VAL_NUMBER` zugefügt. Diese legen für alle numerischen Kanäle die minimalen und maximalen Werte fest.

Objektkardinalitäten

Für jede Klasse, die innerhalb der Spezifikation benutzt wird, wird eine Konstante `MAX_OBJ_` mit angehängten Klassennamen angelegt, die die Anzahl der benutzten Instanzen dieser Klasse festlegt.

Bereich der Objektwerte

Anhand der Objektkardinalitäten werden Konstanten für die Menge der möglichen Objekt- IDs für Instanzen einer Klasse festgelegt. `MIN_RANGE_<Klasse>` und `MAX_RANGE_<Klasse>` geben damit die minimalen und maximalen Werte für Objekt-IDs der Klasse an.

Zusätzlich wird die Konstante `MAX_OBJ` in die Spezifikation geschrieben, die angibt, wieviele Klasseninstanzen insgesamt in der Spezifikation erzeugt werden können. Der Wert dieser Konstante ist identisch mit der Konstante `MAX_RANGE` der letzten angegebenen Klasse.

Klassen- und Variablenbezeichner

Für jede Klasse wird eine ID vergeben. Hierzu wird jeder Klassename als Konstante abgelegt. Die Werte der Konstanten werden hierbei von 0 beginnend durchnumeriert.

Analog zu den Klassenbezeichnern wird jeder Variablenname als Konstante in die Spezifikation geschrieben.

3.4.4.2 Generierung von Variablenklassifikationen und Kanaldefinitionen

Um die Anzahl der benötigten Kanäle möglichst gering zu halten, werden alle verwendeten Variablen in Variablenklassen unterteilt, auf die jeweils über dieselben Kanäle zugegriffen werden kann.

Die Klassifikation richtet sich dabei danach, ob Werte in den Variablen abgelegt werden können (Strukturvariablen), oder ob sie lediglich für die Indizierung von Ereignissen verwendet werden (Thread-, Monitor- und Synchronisationsvariablen).

Strukturvariablen

Strukturvariablen lassen sich unter dem Gesichtspunkt der Indizierbarkeit aufteilen, so daß die Abstraktion von statischen und nicht- statischen Variablen unterschiedliche Zugriffskanäle benutzen. Des weiteren werden Feld- und nicht- Feldtypen unterschieden (siehe auch 'Abstraktionsmodell' auf Seite 43).

Die folgende Tabelle zeigt alle Variablentypen und deren Abbildung auf Kanäle:

Variablentyp	Variablen, die der Typ beinhaltet	zugeordnete Kanäle
nicht- statische Objektvariablen	nicht- statische Variablen, deren Typen weder primitiv, noch Felder sind	`readObj, writeObj`
nicht- statische Feldobjektvariablen	nicht- statische Variablen, deren Typen nicht- primitive Felder sind	`readObjArr, writeObjArr`
statische Objektvariablen	statische Variablen, deren Typen weder primitiv, noch Felder sind	`readSobj, writeSobj`
statische Feldobjektvariablen	statische Variablen, deren Typen nicht- primitive Felder sind	`readSobjArr, writeSobjArr`
nicht- statische numerische Variablen	nicht- statische Variablen, deren Typen numerisch und keine Felder sind	`readNum, writeNum`
nicht- statische numerische Feldvariablen	nicht- statische Variablen, deren Typen numerische Felder sind	`readNumArr, writeNumArr`
statische numerische Variablen	statische Variablen, deren Typen numerisch und keine Felder sind	`readSnum, writeSnum`
statische numerische Feldvariablen	statische Variablen, deren Typen numerische Felder sind	`readSnumArr, writeSnumArr`
nicht- statische boolesche Variablen	nicht- statische Variablen, die vom booleschen Typ, und keine Felder sind	`readBool, writeBool`
nicht- statische boolesche Feldvariablen	nicht- statische Variablen, deren Typen boolesche Felder sind	`readBoolArr, writeBoolArr`
statische boolesche Variablen	statische Variablen, die vom booleschen Typ, und keine Felder sind	`readSbool, writeSbool`
statische boolesche Feldvariablen	statische Variablen, deren Typen boolesche Felder sind	`readSboolArr, writeSboolArr`

Abbildung 3-42: Strukturvariablentypen und zugeordnete Kanäle

Kanäle für die Abbildung nicht- statischer Variablen werden über die möglichen Objekt-IDs indiziert, die sie instanziieren. Dies zeigt folgendes Beispiel:

```
ObjectIDs= {
  objId0.Thread1_obj
  | objId0<-{0..MAX_OBJ},
}
channel readObj, writeObj: ObjectIDs.{0..MAX_OBJ}
```

Hierdurch ist es möglich, verschiedene Instanzen von Klassen zu erstellen (siehe auch 'Abstraktionsmodell' auf Seite 43).

Felder werden analog dazu über die Anzahl der Feldelemente indiziert, wobei die Angabe des Index' hinter der Variablen-ID angegeben wird:

```
StaticObjectArrIDs= {
  fork.arrSize0
  | arrSize0<-{0..MAX_ARRAY_LEN_demo_DiningPhils_Phil_fork}
}
channel readSobjArr,writeSobjArr: StaticObjectArrIDs
```

Thread-, Monitor- und Synchronisationsvariablen

Für die Thread-, Monitor- und Synchronisationsvariablen muß im Vergleich zu den Strukturvariablen weniger Aufwand getrieben werden, da hier keine Kommunikation stattfindet, sondern lediglich die zu observierenden Ereignisse ausgeführt werden und somit die möglichen Belegungen der einzelnen Kanäle nicht durch einen *rectangular*- Datentyp (siehe 'Sprachkonstrukte von FDR' auf Seite 32) beschrieben werden müssen.

Variablentyp	Variablen, die der Typ beinhaltet	zugeordnete Kanäle
Threadvariablen	Variablen vom Typ `java/lang/Thread`	`startT, stopT, destroyT, resumeT, suspendT, joinT`
Monitorvariablen	Variablen, auf die die Bytecodefunktionen `monitorenter` oder `monitorexit` aufgerufen werden	`monitorenter, monitorexit`
Synchronisations-variablen	Variablen, auf die die Methoden `java/lang/Object/wait`, `java/lang/Object/notify` oder `java/lang/Object/notifyAll` aufgerufen werden.	`waitT, notifyT, notifyAllT`

Abbildung 3-43: Kanäle für Thread-, Monitor- und Strukurvariablentypen

3.4.4.3 Einbindung externer Bibliotheken

Innerhalb jeder Spezifikation wird die Datei "java2cspLib.fdr2" eingebunden, die eine Reihe von Bibliotheksfunktionen zur Verfügung stellt.
Dies wird durch die Erweiterung der Spezifikation um die Zeichenkette `include "java2cspLib.fdr2"` bewerkstelligt.

3.4.4.4 Generierung von Funktionsdefinitionen

Eine Funktionsdefinition besteht in CSP/FDR aus Methodennamen, optional angegebenen Parametern und dem Funktionsrumpf.

Der Funktionskopf wird direkt aus den Informationen erzeugt, die über die im Quellprogramm verwendeten Methoden zur Verfügung stehen. Hierbei ist zu beachten, daß, wie im Abschnitt “Abstraktionsmodell” auf Seite 43 erläutert, Objekte durch jeweils zwei Parameter (Variablenname und Objekt- ID) übergeben werden und Rückgaben durch einen zusätzlichen Parameter realisiert werden.

Der Rumpf der Funktion wird anschließend direkt aus dem zugehörigen Syntaxbaum generiert. Jedes Element der abstrakten Syntax besitzt hierzu eine Methode `generateCode`, die das jeweilige Objekt dazu veranlaßt, die von ihm gehaltenen Daten inklusive Kindknoten als Stringrepräsentation auszugeben.

Verfügt eine Funktion über lokale Funktionsdefinitionen, werden diese innerhalb der FDR- Klausel `let... within` in die Spezifikation eingebracht.

3.4.4.5 Erzeugung von Prozessen

Hauptprozeß

Innerhalb erzeugter Spezifikationen, wird der Hauptprozeß (Java- Main- Thread) durch den Bezeichner `PM` beschrieben. Er ruft immer die main- Funktion einer Spezifikation auf, und hat somit folgende Form:

```
PM = main(mainArgs,0)
```

Die Argumente von `main` sind dabei vordefinierte, aber keiner Variablengruppe zugeordnete Variablen- IDs, sowie deren Objekt- IDs, welche immer den Wert 0 enthalten.

Abbildung von Threads auf Prozesse

Die einzelnen im Quellprogramm ablaufenden Threads werden durch die Bezeichner `P0` bis `Pn` deklariert. Hierbei ist darauf zu achten, daß alle Instanzen einer Thread- Klasse in einem Prozeß zusammengefaßt werden. Folgendes Codesegment soll hierzu als Beispiel dienen:

```
P0 = ( ||| i0 : {MIN_RANGE_Test..MAX_RANGE_Test} @
   startT.i0.Test_t1 ->
   readObj.i0.Test_t1?t ->
   run(i0.Test_t1, t) )
```

In diesem Beispiel existiert die Threadvariable `Test_t1`, von denen es mehrere Instanzen geben kann. Hierzu werden die möglichen Instanzobjekte über die Variable `i0` iteriert. Für jede der Instanzen wird erwartet, daß das Hauptprogramm, oder ein anderer Thread das Ereignis `startT` ausgeführt hat. Ist dies der Fall, wird die Objekt ID der Threadvariable ausgelesen und der Rumpf des Threads (Funktion `run`) gestartet. Alle resultierenden Instanzen werden dann durch den Paralleloperator (|||) als parallele Prozesse spezifiziert.

Analog hierzu ist die Benutzung von Feldern als Threadvariablen. Hierbei werden nicht nur die einzelnen Instanzen der Threadvariable betrachtet, sondern auch die Elemente des Feldes.

3.4.4.6 Systemerzeugung

Das Gesamtsystem setzt sich aus den einzelnen erzeugten Prozessen zusammen, die sich über die Ereignisse `start`, `stop`, `resume` und `suspend` synchronisieren.

Als Beispiel hierfür kann folgendes System gelten, bei dem der Hauptthread, den Prozeß `P1` startet:

```
SYSTEM= PM [|{|start,stop,resume,suspend|}|] P1
```

Es ist jedoch nicht ausreichend, lediglich die einzelnen Threads sinnvoll zusammenzustellen, da hier noch keine Hilfsprozesse für Strukturvariablen und Kontrollprozesse für das Threadverhalten berücksichtigt wurden.

Hilfsprozesse für Strukturvariablen

Variablen können in CSP/FDR nur in Form von Prozessen abstrahiert werden, die parallel zu den untersuchenden Prozessen laufen, und diese mit den benötigten Variablenbelegungen "versorgen". Für alle in Abschnitt 3.4.4.2 dargestellten Strukturvariablentypen existieren solche Hilfsprozesse, die innerhalb der JAVA2CSP- Bibliothek implementiert sind (siehe Anhang C). Diese Hilfsprozesse werden für alle in der Spezifikation vorhandenen Variablen angestoßen und mit dem Gesamtsystem synchronisiert. Das folgende Beispiel illustriert dies:

```
SYSTEM=
   ( ( PM [| {| startT, stopT, suspendT, resumeT |} |] P0 )
   [| {| readNum, writeNum |} |]
      ( ( ||| i0 : {MIN_RANGE_Thread1..MAX_RANGE_Thread1} @
           VarNum(i0.gaga,0, MIN_NUM, MAX_NUM) ) )
   [| {| readSobj, writeSobj |} |]
      ( VarSobjID(local1) ||| VarSobjID(bottleNeck) ) )
```

Hierbei existieren die Prozesse `PM` und `P0`, die auf die nicht- statische Variable `gaga` und die statischen Variablen `local1` und `bottleNeck` zugreifen. Die Hilfsprozesse werden innerhalb von `SYSTEM` mit Initialwerten gestartet und über die Zugriffskanäle `readNum`, `writeNum`, `readSobj` und `writeSobj` synchronisiert.

Hilfsprozesse für Objekt- IDs

Wie bereits im Abschnitt “Abstraktionsmodell” auf Seite 43 beschrieben, werden innerhalb von Spezifikationen dynamisch Objekt- IDs vergeben. Diese dynamische Generierung übernimmt pro verwendeter Java- Klasse ein eigener Prozeß (`NewObjectIDGenerator`), der als rekursiver Zähler implementiert ist (siehe Anhang C).

Alle vorhandenen Objektzähler synchronisieren sich über den Kanal `readNewObjectID`, der zu jeder Klasse den entsprechend aktuellen Objektwert enthält.

```
SYSTEM=
   ( ( PM [| {| startT, stopT, suspendT, resumeT |} |] P0 )
   [| {| readNewObjectID |} |]
     NewObjectIDGenerator(Test, MIN_RANGE_Test, MAX_RANGE_Test) )
)
```

Kontrollprozesse für Threadverhalten

Die Kontrollprozesse für Threadvariablen spiegeln das korrekte Verhalten der zu untersuchenden Eigenschaften des Systems wider. Eine genaue Diskussion dieser Eigenschaften befindet sich im Abschnitt “Verifizierbares Verhalten von Threads” auf Seite 38.

Analog zu den Hilfsprozessen für Strukturvariablen werden die Kontrollprozesse für Threadverhalten über die in Abbildung 3-43 dargestellten Kanäle in die Spezifikation integriert.

Folgendes Beispiel zeigt das schon bekannte System mit der Monitor- und Synchronisationsvariable `obj` und der Threadvariable `t1`.

```
SYSTEM=
   ( ( ( ( PM [| {| startT, stopT, suspendT, resumeT |} |] P0 )
   [| {| monitorenter, monitorexit |} |]
     ( ( ||| i0 : {MIN_RANGE_Thread1..MAX_RANGE_Thread1} @
       Monitor(i0.Thread1_obj) ) ) )
   [| {| startT, stopT, suspendT, resumeT |} |]
     ( ||| i0 : {MIN_RANGE_Test..MAX_RANGE_Test} @
       Thread(i0.t1) ) ) )
   [| {| waitT, notifyT |} |]
     ( ( ||| i0 : {MIN_RANGE_Object..MAX_RANGE_Object} @
       WaitNotify(i0) ) ) )
```

3.4.4.7 Assert Definitionen

Die Assert- Definitionen einer Spezifikation dienen dazu festzulegen, welche Eigenschaften des Systems überprüft werden sollen. In unserem Fall sind dies die Deadlock- und Livelock- Eigenschaften, so daß die beiden Definitionen

```
assert SYSTEM :[deadlock free [F]]
assert SYSTEM :[livelock free [F]]
```

in der Spezifikation dazu führen, daß das FDR- Werkzeug diese direkt als mögliche überprüfbare Eigenschaften zur Verfügung stellt.

Kapitel 4

Schlußbetrachtung

4.1 Verwandte Arbeiten

Wie bereits in der Einleitung erwähnt, ist das Gebiet der automatischen Softwareverifikation noch nicht sehr populär und deshalb auch nicht erschöpfend bearbeitet. Wahrscheinlich gibt es aus diesem Grund nur sehr wenige vergleichbare Werkzeuge, die im folgenden vorgestellt werden.

Verisoft

Verisoft (vgl. [God97]) ist ein Werkzeug für die systematische Analyse des Zustandsraumes von Softwaresystemen, die aus einer endlichen Menge konkurrierender Prozesse zusammengesetzt sind und über eine endliche Menge von Kommunikationsobjekten (Message Queues, Semaphoren, gemeinsam genutzter Speicher, TCP/IP- Verbindungen, etc.) verfügen. Der Zustandsraum eines Systems ist dabei ein gerichteter Graph, der das kombinierte Verhalten aller am System beteiligten Komponenten darstellt. *Verisoft* untersucht den Zustandsraum eines Systems, indem es die Ausführung der einzelnen Komponenten beobachtet und kontrolliert. Es sucht nach Koordinationsproblemen wie z.B. Verklemmungen oder Livelocks zwischen konkurrierenden Prozessen und Verletzungen benutzerspezifischer Bedingungen. Es besitzt des weiteren eine Option, um Nichtdeterminismus zu simulieren.

Interessant hierbei ist, daß das Produkt über keine eigene Sprache verfügt, sondern ein variables Übersetzer- Frontend besitzt, so daß Programme, die z.B. in Sprachen wie C, C++, Java oder anderen imperativen oder objektorientierten Sprachen geschrieben sind, analysiert werden können.

Im Vergleich zu dieser Arbeit besitzt *Verisoft* den Vorteil, daß eine Reihe von Programmiersprachen unterstützt werden, während der JAVA2CSP- Übersetzer lediglich Java Bytecode untersuchen kann. Außerdem wird ein eigener Model- Checker benutzt, so daß hier keine Bindung an ein weiteres Produkt, wie dem FDR- Werkzeug vorliegt.

Der hauptsächliche Nachteil von *Verisoft* gegenüber dieser Arbeit liegt darin, daß alle zu untersuchenden Softwaresysteme im Quellcode vorliegen müssen, während das Ergebnis dieser Arbeit lediglich den Bytecode eines Projekts zur Analyse benötigt, was somit ein Novum darstellt.

Spin

Spin (vgl. [Spin]) ist ein Werkzeug zur Überprüfung der logischen Konsistenz verteilter Systeme im Bezug auf Kommunikationsprotokolle. Ein Kommunikationssystem wird durch die C- ähnliche Metasprache *Promela* (Process or Protocol Meta Language) beschrieben. Die Sprache erlaubt die dynamische Generierung nebenläufiger Prozesse, wobei die Kommunikation über Nachrichtenkanäle synchron mittels ADA- Rendezvous´ (vgl. [WaWiFi87]) oder asynchron mittels gepufferten Kanälen erfolgen kann.

Mit einer gegebenen Spezifikation in *Promela* kann *Spin* zufällige oder interaktive Simulationen der Systemausführung erzeugen. Des weiteren ist es in der Lage C- Programme aus der *Promela* Spezifikation zu erzeugen, die eine schnelle und weitreichende Verifikation des Zustandraumes des Systems erlaubt.
Während der Simulation und Verifikation überprüft *Spin* das Vorhandensein von Verklemmungen, Livelocks, unbenutzten Codeabschnitten und nicht spezifizierter Kommunikation auf den Kommunikationskanälen. Des weiteren unterstützt das System die Verifikation linearer zeitlicher Eigenschaften, die entweder in *Promela* oder durch die direkte Formulierung in einer temporären Logik festgelegt werden können.

Der Hauptunterschied zwischen JAVA2CSP und *Spin* liegt darin, daß alle zu prüfenden Implementierungen in einer spezifischen Sprache geschrieben sein müssen, so daß die Verifikation von bereits existenten Systemen sehr aufwendig ist.

Der hauptsächliche Vorteil des Systems ist eng mit dessen hauptsächlichen Nachteil verbunden und liegt darin, daß das System stark implementierungsorientiert ist, da die verwendete Spezifikationssprache *Promela* stark an C angelegt ist und somit eine leichte Abbildung gefundener Fehler in der *Promela*- Spezifikation auf das C- Programm, auf das die Spezifikation basiert, möglich ist. Im Vergleich zum JAVA2CSP- Übersetzer lassen sich Implementierungen von Kommunikationsprotokollen (siehe auch 'Interprozeßkommunikation' auf Seite 90) und Anforderungen an das Zeitverhalten definieren.

4.2 Kritik & Ausblick

Dieser Abschnitt stellt eine kritische Auseinandersetzung mit der Funktionalität des erstellten Systems dar und beschreibt mögliche Erweiterungen, die jedoch den Rahmen dieser Diplomarbeit überschreiten und deshalb nicht implementiert wurden.

4.2.1 Fließkommawerte

Das vorliegende System beherrscht die Abbildung von booleschen und ganzzahligen Werten, sowie Objektidentitäten als Strukturelemente. Hierbei ist die Verwendung von Fließkommazahlen außer acht gelassen worden, da deren Benutzung für die Steuerung von Threads einem relativ geringen Gewicht zufällt.
Des weiteren wird der Experimentraum eines Model- Checkers durch die Verwendung von Fließkommawerten sehr stark ausgedehnt. Hierbei bleibt es fraglich, ob sich CSP/ FDR- Abstraktionen, die über eine Reihe von Kanälen auf Fließkommabasis verfügen, überhaupt oder nur mit einem exorbitanten zeitlichen Aufwand prüfen lassen.

Fließkommazahlen sind indes in FDR sehr leicht zu simulieren, indem zwei ganzzahlige Werte aneinander gebunden werden, wobei einer den ganzzahligen Anteil und der andere den Nachkommastellenanteil annimmt. Diese Methode verfügt jedoch nur über eine begrenzte Genauigkeit. Des weiteren müßten hierfür alle arithmetischen Ganzzahloperationen auf Fließkommawerte übertragen werden.

Die Verwendung von Fließkommazahlen ist bereits in FDR2 enthalten (vgl. [Formal97]), jedoch nur experimentell implementiert und nicht dokumentiert, so daß hier von deren Verwendung abgesehen wurde.

4.2.2 Zeichenketten

Zeichenketten werden durch FDR nicht unterstützt. Roscoe rechtfertigt dies in [Formal97] damit, daß FDR keine Programmiersprache ist, in der Eingaben und Ausgaben auf String- Basis unnötig zur Komplizierung führen. Er beschreibt ebenfalls, daß es für einige Problemstellungen durchaus nützlich wäre, Zeichen und Zeichenketten zu benutzen, es jedoch keine Beispiele gibt, in denen dies wirklich unabdingbar und nicht anders abbildbar wäre.

In Java und Java Bytecode werden Zeichenketten aus Werten vom primitiven Typ `char` zusammengesetzt, so daß alle String- Operationen auf einzelne Zeichen abgebildet werden. Bei der Übersetzung von Java nach CSP/FDR geht hierbei die Semantik der Funktionen auf Zeichenketten nicht verloren, so daß es denkbar ist, Zeichenketten und Operationen auf Zeichenketten als Strukturvariablen zu benutzen.

Allerdings werden die hierdurch erzeugten Spezifikationen sehr groß und komplex, da alle genutzten Methoden, die auf Strings arbeiten in CSP/FDR übersetzt werden müssen. Dieser Umstand führt zur Überlegung, daß hier eine weitere Abstraktion auf die Ebene von Strings angebracht wäre, um die Größe der erzeugten Spezifikationen zu minimieren.

Eine Vereinfachung könnte zunächst durch die Abbildung von Strings auf Sequenzen von Zeichen erfolgen, wobei jede Zugriffsfunktion auf Zeichenketten durch FDR- Operationen auf diese Zeichensequenzen übersetzt wird. Hierbei ist darauf zu achten, daß Java Zeichen im Unicode Format darstellt (siehe ’Dateiformat’ auf Seite 50), wobei jedes Zeichen 16- bit groß sein kann. Hierdurch würde eine große Anzahl von Zuständen pro Zeichen entstehen, so daß der Zahlenbereich zusätzlich eingegrenzt werden müßte. Die Einhaltung dieser beiden Maßnahmen würde zu kompakten und effizienten Spezifikationen führen.
Eine weitere Alternative würde darin bestehen, höhere Abstraktionsgrade einzuführen (siehe ’Qualität erzeugter CSP/FDR Abstraktionen’ auf Seite 90).

4.2.3 Synchronisation mit Timeout

Die vorliegende Version des Model- Checkers FDR stellt keine direkte Möglichkeit zur Verfügung, Ereignisse in einem zeitkritischen Kontext zu behandeln. Aus diesem Grund können die Java Synchronisationsmethoden `wait(timeout)` und `join(timeout)`, die über Zeitüberschreitungswerte Verfügen nicht umgesetzt werden. Zwar ist es möglich, nicht zeitgesteuerte Eigenschaften von "getimeten" Prozessen zu prüfen (vgl. [Rosecoe97], Kapitel 14), jedoch erweitert dies den Experimentraum eines Model- Checkers enorm, so daß es fraglich ist, ob sich eine Spezifikation mit zeitgesteuerten Ereignissen noch in verträglicher Zeit überprüfen läßt. Es existieren jedoch Ansätze, CSP durch Zeitverhalten zu erweitern. Als weiterführende Literatur seien hier die Werke von Davies ([Dav93], [DJR+92]) genannt.

4.2.4 Rekursion

Die Nachbildung von Rekursion läßt sich mit der vorliegenden Version des FDR- Werkzeugs ebenfalls nur sehr aufwendig bewerkstelligen. Der Grund hierfür liegt darin, daß Kanäle nur auf oberster Ebene (*top- level*) und nicht innerhalb von Funktionen in einer Spezifikation deklariert werden dürfen (vgl. [Formal97]). Der JAVA2CSP- Übersetzer führt für jede benutzte lokale Variable in einem Java- Programm genau eine entsprechende Kanaldefinition ein. Ruft sich nun eine übersetzte Methode selbst auf, führt dies bei der Überprüfung der erzeugten Spezifikation dazu, daß der Wert im Kanal nach Beendigung einer Rekursion nicht auf den der vorigen Rekursionsstufe zurückgesetzt wird.
Aus diesem Grund sollten die zu überprüfenden Bestandteile übersetzter Java Programme frei von Rekursion sein.
Um Rekursion innerhalb von CSP/FDR zu ermöglichen, müßte jede potentiell innerhalb von Rekursionen benutzte Java- Variable durch eine Feld von Kanälen übersetzt werden, deren Länge die maximale Rekursionstiefe darstellt. Jede Rekursionsstufe würde somit eine eigene Variablemengen erhalten, die die zugehörigen Werte korrekt speichern könnte.
Die Nachteile dieses Vorgehens liegen jedoch darin, daß für jede rekursive Funktion klar sein müßte, wie hoch deren Rekursionstiefe ist (dies ist seitens des Übersetzungssystems kaum mit vertretbarem Aufwand möglich), sowie in der Tatsache, daß jedes neu eingeführte Variablenkonstrukt den Zustandsraum des Systems vergrößert und damit die Verifikation verlangsamt.

4.2.5 Schnittstelle zu Isabelle/HOL

Isabelle/ HOL ist ein Spezifikations- und Verifikationssystem. Isabelle stellt ein generisches System dar, mit dem sich logische Formalismen implementieren lassen. Zusätzlich dazu ist HOL eine Spezialisierung von Isabelle. HOL (Higer Order Logic, Logik höherer Ordnung) benutzt dabei zur Darstellung mathematischer Probleme, Konzepte der funktionalen Programmierung und verbindet diese mit logischen Elementen. Die komplette Referenz zu Isabelle stellt [Paulson98] dar, während in [Nipkow98] eine Einführung gegeben wird.
Am Fachbereich Informatik der Universität Bremen wurde ein System entwickelt, daß die Verwendung von CSP innerhalb von Isabelle/ HOL erlaubt, und somit zusätzliche Einsatzmöglichkeiten für das System bietet.
In diesem Zusammenhang ist es denkbar, eine Schnittstelle zwischen dem JAVA2CSP-Übersetzer und dem Theorembeweiser Isabelle/ HOL zu schaffen, so daß direkt aus Java Programmen erzeugte Spezifikationen in den Theorembeweiser einfließen könnten.
Es wäre so denkbar, den JAVA2CSP- Übersetzer anzuweisen, Spezifikationen in Form der abstrakten CSP Syntax für die konkrete Isabelle/ HOL- Syntax zu erzeugen, so daß eine direkte Integration des Übersetzers möglich wäre.

Dieses Vorgehen wurde im Rahmen dieser Diplomarbeit bereits angedacht, jedoch wieder verworfen, da die Unterschiede zwischen dem von FDR genutzten CSP- Dialekt und der CSP- Schnittstelle von Isabelle/ HOL zu groß sind.

4.2.6 Interprozeßkommunikation

Es ist es weiterhin denkbar, sich nicht lediglich auf die Analyse von Tasks zu beschränken, die innerhalb eines Prozesses ablaufen, sondern die Kommunikation einzelner Prozesse untereinander zu analysieren.

Hierbei ist es zunächst einmal nötig, die einzelnen Arten der Interprozeßkommunikation, wie z.B. TCP/IP- oder UDP- Kommunikation bzw. entfernte Unterprogrammaufrufe (*Remote Procedure Calls*, vgl. [Tan95]) zu analysieren, und festzulegen, welche Möglichkeiten bestehen, daß ein Prozeß durch deren Verwendung in einen Verklemmungszustand gerät.

Als Anwendung hierfür läßt sich beispielsweise die Verifikation von implementierten Übertragungsprotokollen auf Verklemmungsfreiheit anführen. So ließen sich beispielsweise komplexe Client/ Server- Implementierungen auf TCP/IP- und UDP- Ebene auf Verklemmungen überprüfen.

Analog dazu ließe sich für entfernte Unterprogrammaufrufe übersprüfen, ob es eine Situation gibt, in der der aufrufende Prozeß in einem Dauerschlafzustand (siehe ’Verhungern, Dauerschlaf und vorzeitige Beendigung’ auf Seite 14) gelangen kann. Dies kann sehr leicht passieren, da der aufrufende Prozeß nach dem Remote Procedure Call in die Sleep Queue des Betriebssystems eingereiht, und z.B. im Falle eines Implementierungsfehler in der aufgerufenen Funktion nicht mehr in einen ausführungsbereiten Zustand gebracht wird.

4.2.7 Qualität erzeugter CSP/FDR Abstraktionen

Wie bei allen Übersetzern, deren Ausgabeformat wiederum Quellcode ist, muß sich auch das hier entwickelte Werkzeug an der Qualität der erzeugten Ausgabe messen lassen. Aus diesem Grund widmet sich dieser Abschnitt der Betrachtung des Qualitätsaspektes im Kontext der Verständlichkeit und Handhabbarkeit für menschliche Leser und der Effizienz im Hinblick auf die Verifikation der Abstraktionen mittels Model- Checkern.

Komplexität von Bezeichnern

In der Ausgabe des JAVA2CSP- Übersetzers werden immer möglichst kurze Bezeichner angegeben, um die Übersichtlichkeit zu fördern. Nichtsdestotrotz ist dieser Aspekt stark an den übersetzten Programmcode gebunden, so daß viele Überladungen von Methoden dazu führen, daß die Methodennamen immer gleich sind, und so die Bezeichner in der re-

sultierenden Abstraktion um Paketnamen erweitert werden. Dasselbe gilt für oft benutzte lokale Variablen, wie z.B. dem obligaten Iterator `i` oder der "Universalvariable" `x`. Andererseits ist durch dieses Vorgehen eine genaue Abbildung auf den Ausgangscode möglich, um aufgespürte Lebendigkeitsausfälle zu beheben. Aus diesem Grund wurde auch die Überlegung verworfen, lediglich unqualifizierte Variablen und Methodennamen zu benutzen und diese durchzunumerieren. Dieses Vorgehen sorgt zwar für sehr kompakte Abstraktionen, läßt aber kaum mehr eine Abbildung auf den Originalcode zu.

Für eine weitere Version des Werkzeug wäre aus diesem Grund eine Option sinnvoll, mit der sich zwischen beiden Modi umschalten läßt.

Effizienz im Hinblick auf Model- Checking

Das größte Problem, das alle automatisch generierten CSP/FDR- Abstraktionen haben, dürfte die Effizienz sein, mit der sie sich mittels Model- Cheking prüfen lassen. Der hier gewählte Ansatz der Übersetzung (siehe 'Abstraktionsmodell' auf Seite 43) verfolgt das Ziel, möglichst gut lesbare und zugleich auch effizient prüfbare Abstraktionen zu erzeugen.

Der größte Aufwand bei der Erstellung von Abstraktionen ist hierbei die Aufgabe der Eingrenzung des Experimentraums der verwendeten Model- Checker. Es sollten hierzu möglichst wenige Kanäle und Prozesse erzeugt werden. Außerdem sollten alle Kanalprotokolle so gewählt sein, daß sie möglichst wenige Zustände besitzen. Die Entscheidung welche Werte hierfür optimal geeignet sind, sind durch eine automatisierte Analyse nicht immer möglich, so daß eine Reihe von Werten manuell anzugeben sind.

Wichtig ist jedoch hierbei, daß die Anzahl der Kanäle in der Abstraktion direkt an die Anzahl der Variablen im Quellcode gekoppelt ist. Lediglich Methodenrückgaben und der Objekt- ID- Erzeugungsmechanismus des Abstraktionsmodells führen zusätzliche Kanäle ein. Nicht benutzte oder unerwünschte Variablen (siehe 'Löschen nicht benötigter Algorithmen im Syntaxbaum' auf Seite 78), werden vom Übersetzungssystem nicht in Kanäle übersetzt.

Hierzu ist allerdings zu bemerken, daß sich die Entwicklung von effizienten Modelchekkern gerade erst in den Anfängen befindet und immer effizientere Systeme zu erwarten sind. Des weiteren hängt die Geschwindigkeit, mit der Abstraktionen überprüft werden können, stark von der eingesetzten Hardware ab. Auch hier sind in den nächsten Jahren Leistungssteigerungen zu erwarten, so daß es immer weniger darauf ankommen wird, wie erschöpfend die zu überprüfenden Spezifikationen abstrahiert sind.

Nichtsdestotrotz ist der Einsatz von weiterführenden Abstraktionsmodellen möglich. Hier sei zum einen die dynamische Prozeß- und Kanalerzeugung nach dem π - *Kalkül* (vgl. [Milner91]) genannt, mit dem sich Prozesse und Kanäle höherer Ordnung erstellen lassen, die zur Laufzeit des Systems mit Typen versehen werden können. Dieses Kalkül existiert jedoch nur als Theorie und wird deshalb hier für eine konkrete Umsetzung nicht betrachtet.

Des weiteren besteht die Möglichkeit weiterführender Abstraktionen, wie sie z.B. in [Rosecoe97] (Kapitel 12: Abstractions) dargestellt wird. Jedoch spielt bei der Durchführung solcher Abstraktionen die Semantik des übersetzten Systems eine große Rolle, so daß sich dies nur mit einem extrem großen Aufwand in die Übersetzerimplementierung integrieren läßt.

Manuell anzugebene Werte

Jede erzeugte Abstraktion sollte nachbearbeitet werden, um die Effizienz im Hinblick auf Model- Checking zu optimieren. Um diese Einstellungen möglichst optimal zu gestalten, muß man das Programm kennen, um die Werte für die Anzahl der Objekt- IDs und der Feldlängen zu setzen. Dies ist einerseits ein Nachteil, da es somit erforderlich ist, daß der Entwickler des Ausgangsprogramms (oder eine Person, die sich damit gut auskennt) die erstellte Abstraktion selber bearbeitet. Andererseits können Fehler, die durch die Analyse eines Model- Checkers zu Tage gefördert werden in der Regel selbst nur von den Ent-

wicklern behoben werden, so daß dieser Aspekt als eher unkritisch anzusehen ist. Er bedeutet allerdings einen Komfortverlust gegenüber der Zielsetzung der vollautomatischen Übersetzung.

Leider ist eine vollautomatische Übersetzung mit dem Aspekt der Effizienz durch Model- Checking nicht vereinbar, da nur durch die manuelle Einbringung von Objekt- ID- Mengen eine effizient überprüfbare Abstraktion zu erreichen ist. Der Grund hierfür liegt darin, daß eine aufwendige dynamische Analyse des zu übersetzenden Programms notwendig wäre, um die Quantität aller erzeugter Objektinstanzen und Bereiche vorkommender numerischer Werte zu ermitteln.

Es wäre allerdings für eine weitere Version des JAVA2CSP Werkzeugs möglich, zumindest die Feldlängen automatisch zu bestimmen. Hierzu müßten nicht nur (wie bisher), die explizit mit `new` erzeugten Felder auf deren Länge überprüft werden, sondern auch alle Feld- Parameter mit deren möglichen Belegungen im Programm. Die größte Feldlänge der übergebenen Objekte wäre somit zugleich die für den Parameter anzunehmende Feldlänge.

4.3 Erfahrungen

Als ich begann, mir Gedanken über diese Diplomarbeit zu machen, dachte ich, daß die größten Schwierigkeiten im Bereich des Compilerbaus liegen würden, da mich meine Erfahrungen aus dem studentischen Projekt BALI (vgl. [BALI99]) lehrten, daß die Umsetzung von Java Bytecode in effizient zu analysierende Strukturen keineswegs trivial ist.
Es stellte sich jedoch im Nachhinein heraus, daß die meisten Hürden auf dem Weg lagen, eine Abbildung von Java Programmen in eine äquivalente und performante CSP/ FDR- Abstraktion zu entwickeln.

Die Arbeit an dem vorliegenden Modell hat mich um viele Erfahrungen im Zusammenhang mit dem Model- Checking- Werkzeug FDR, und um einige graue Haare reicher gemacht.

Nichtsdestotrotz bin ich zum Schluß gelangt, daß diese Arbeit ein nicht unwichtiger Schritt zur Verbesserung der Vorraussetzungen zur Erstellung sicherer Systeme ist.

Literaturverzeichnis

[Baillie98] Jean Baillie: Introduction to CSP- Course Material. Department of Computer Science, University of Hertfordshire, UK, 1998. (http://www.cs.herts.ac.uk/~jean/notes99/notes99.html)

[BALI99] Studentisches Projekt BALI: Projektbericht des studentischen Projekts BALI. Universität Bremen, 1999. (http://www.tzi.de/~bali)

[BKPS97] Bettina Buth, Michael Kouvaras, Jan Peleska, Hui Shi: Deadlock Analysis for a Fault- Tolerant System. In Michael Johnson (Ed.): Algebraic Methodology and Software Technology. Proceedings of the AMAST 1997. Sidney, Australien, Dezember 1997, Springer LNCS 1349 (1997), pp. 60-75.

[BMI92] Der Bundesminister des Innern der Bundesrepublik Deutschland: Planung und Durchführung von IT- Vorhaben- Vorgehensmodell. Koordinierung- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung (KBSt), 1992.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler: Making the future safe for the past: Adding Genericity to the Java Programming Language. Submitted to OOPSLA98, 1998.

[Brinch75] P. Brinch Hansen: The Programming Language Concurrent Pascal. IEEE Trans. on Software Engineering, vol. SE-1, Seite 199-207, Juni 1975.

[CoElSh71] E.G. Coffman, M.J. Elphik, A. Shoshani: System Deadlocks, Computing Serveys, vol3, Seite 67-78, Juni 1971.

[Court+71] P.J.Courtois, F.Heymans, D.L.Parnas: Concurrent Control with Readers and Writers, Commun. of the ACM, vol.10, Seite.667-668, Oktober 1971.

[Dav93] J.W.M. Davies: Specification and proof in real-time CSP. Cambridge University Press, 1993.

[Dijk65b] E.W. Dijkstra: Co- operating Sequential Processes. In Genuys, F. (Ed): Programming Languages. Academic Press, London,1965.

[DJR+92] J.W.M. Davies, D.M. Jackson, G.M. Reed, A.W. Roscoe, S.A. Schneider: Timed CSP: theory and applications. In Real time: theory and practice, Springer LNCS 600. 1992

[Flan96] David Flanagan: Java in a Nutshell. O´Reilly & Associates, Inc, Sebastopol, USA, 1996.

[Formal95] Formal Systems: Failures Divergence Refinement, FDR User Manual and Tutorial (Version 1.42). Formal Systems (Europe) Ltd., 1995.

[Formal97] Formal Systems: Failures Divergence Refinement, FDR2 User Manual (Version 2.28). Formal Systems (Europe) Ltd., 1997.

[FrKi96] M. Franz, T. Kistler: Slim Binaries, Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine, Juni 1996. (http://www.ics.uci.edu/~oberon/research.html)

[God97] Patrice Godefroid: Model Checking for Programming Languages using Veri-Soft. In Proceedings of the 24th ACM Symposium on Principles of Programming Languages. Paris, France, Januar 1997 (http://www.bell-labs.com/project/verisoft/)

[GoJoSt96] James Gosling, Bill Joy, Guy Steele: The Java Language Specification. Addison- Wesley- Longman, Reading, Massachussets, USA, 1996.

[GrKn97] Mark Grand, Jonathan Knudsen: Java Fundamental Classes Reference. O´Reilly & Associates, Inc, Sebastopol, USA, 1997.

[HaPe98] Anne E. Haxthausen, Jan Peleska: Formal Development and Verification of a Distributed Railway Control System. In Proceedings of the 1st FMERail Workshop, Utrecht, The Netherlands, Juni 1998.

[HeHo94] Guido Herrtwich, Günter Hommel: Nebenläufige Programme. Springer- Verlag, Berlin, 1994

[Hoare74] C.A.R Hoare: Monitors, An Operating System Structuring Concept. Commun. of the ACM, vol. 17, p.549-557, Oct 1974; Erratum in Commun. of the ACM, vol18, Seite 95, Februar 1975.

[Hoare85] C.A. Hoare. Communicating Sequential Processes. Prentice-Hall International, Englewood Cliffs, New Jersey, USA, 1985.

[IEEE754] IEEE Standard for Binary Floating-Point Arithmetic. ANSI/IEEE Std. 754-1985. Available from Global Engineering Documents, 15 Inverness Way East, Englewood, Colorado 80112-5704 USA, 1985.

[KSS95] Steve Kleiman, Devang Shah, Bart Smalders: Programming With Threads. Prentice-Hall International, Englewood Cliffs, New Jersey, USA, 1995 .

[Lea97] Doug Lea: Concurrent programming in Java- Entwurfsprinzipien und Muster. Addison- Wesley- Longman, Bonn, 1997.

[LiYe97] Tom Lindholm, Frank Yellin: The Java Virtual Machine Specification. Addison-Wesley- Longman, Reading, Massachussets, USA, 1997.

[MeyDo97] Jon Meyer, Troy Downing: Java Virtual Machine. O´Reilly & Associates, Inc, Sebastopol, USA, 1997.

[Milner91] Robin Milner: The Polyadic π - Calculus: a tutorial. Laboratory for Foundations of Computer Science Department, University of Edinburgh, 1991.

[Naur60] P. Naur (Ed). Report on the algorithmic language Algol 60. Comm. ACM, 3 (1960), 299-314, und Comm. ACM, 6, 1 (1963), 1-17.

[Nipkow98] Tobias Nipkow: Isabelle/ HOL- The Tutorial (Draft). Technische Universität München, Institut für Informatik, 1998. (http://www4.in.tum.de/~isabelle/dist/)

[OaWo99] Scott Oaks, Henry Wong: Java Threads, Second Edition. O´Reilly & Associates, Inc, Sebastopol, USA, 1999.

[Open93] Open Group: File System Safe UCS Transformation Format (FSS_UTF), X/Open Preliminary Specification. X/Open Company Ltd., Document Number P316, Juni 1993. (http://www.opengroup.org/publications/catalog/p316.htm)

[Paulson98] Lawrence C. Paulson: The Isabelle Reference Manual. Computer Laboratory, University of Cambridge, 1998. (http://http://www4.in.tum.de/~isabelle/dist/)

[Rational98] Rational Software Corporation: Rational Rose 98- Using Rational Rose. Rational Software Corporation, Cupertino, USA, 1998.

[Roscoe95] A.W. Roscoe: CSP and determinism in security modelling. In IEEE Symposium on Security and Privacy, 1995.

[Rosecoe97] A.W. Roscoe: The theory and practice of concurrency. Prentice Hall International, Englewood Cliffs, New Jersey, USA, 1998.

[RWW94] A.W. Roscoe, J.C.O. Woodcock, L. Wulf. Non- interference through Determinism. In European Symposium on Research in Computer Security, vol.875 of Lecture Notes in Computer Science, Seite 33-53, 1994.

[Spin] On-The-Fly, LTL Model Checking with SPIN (http://cm.bell-labs.com/cm/cs/what/spin/index.html)

[Tan95] Andrew S. Tanenbaum: Moderne Betriebssysteme. Hanser Verlag, München, 1995.

[Unicode92] The Unicode Consortium: The Unicode Standard, Worldwide Chracater Encoding, Version 1.0, Volume2. Addison- Wesley- Longman, 1992. (http://www.unicode.org)

[Watt96] David A. Watt: Programmiersprachen: Konzepte und Paradigmen. Hanser Verlag, München, Wien, 1996.

[WaWiFi87] David A. Watt, Brian A. Wichmann, William Findlay: Ada : Language and Methodology. Prentice Hall International, Englewood Cliffs, New Jersey, USA, 1987.

[Wirth95] Niklaus Wirth: Algorithmen und Datenstrukturen. Pascal Version. Teubner Verlag, Stuttgart, 1995.

Anhang A

Benutzung des Übersetzers

Dieser Anhang gibt einen Überblick über die Installation und die Benutzung des JAVA2CSP- Übersetzers und erläutert die einzelnen Übersetzungsoptionen des Systems. Des weiteren wird eine einfache Benutzungsoberfläche für das kommandozeilenorientierte System vorgestellt.

Installation

Als System für den Betrieb des Übersetzers kommt jede beliebige Plattform in Frage, für die das Java Development Kit (JDK) in der Version 1.1 oder höher vorliegt. Bei der Verwendung anderer Java Laufzeitsysteme kann keine Gewähr für die einwandfreie Funktion übernommen werden.
Die Verteilung des JAVA2CSP- Systems erfolgt in Form einer Zip- Datei namens "java2csp.zip". Für die Installation muß lediglich der Java Klassenpfad (Classpath) um den Pfad und den Namen dieser Zip- Datei erweitert werden.

Der JAVA2CSP *Übersetzer*

Der JAVA2CSP- Übersetzer ist ein kommandozeilenorientiertes System, daß mittels des Kommandos `"java java2csp.java2cspConverter"` innerhalb des betriebssystemspezifischen Kommandozeileninterpreters (z.B. "bash" unter Unix oder "CMD" unter Microsoft Windows) aufgerufen wird.
Als Kommandozeilenparameter ist der Klassenname der Hauptklasse (Klasse, die die Main- Methode enthält) des Programms notwendig, das zu analysieren ist. Die Angabe dieser Klasse erfolgt in paketorientierter Form, wie dies z.B. beim Java Interpreter der Fall ist (z.B. `java.lang.String` oder `java2csp.java2cspConverter`).
Der Aufruf des Programms ohne Kommandozeilenparameter führt zur Ausgabe von Benutzungsinformationen, die an dieser Stelle vertiefend erläutert werden.

- **-c<n> (comments)**: Diese Option veranlaßt den Übersetzer, die Ausgabe von Kommentaren in der zu erstellenden Spezifikation einzuschalten (-c1), bzw. zu unterdrücken (-c0). Im Kopf der resultierenden Spezifikation werden bei eingeschalteten Kommentarmodus alle Java Dateien angegeben, die zur Übersetzung benutzt wurden. Des weiteren werden für alle übersetzten Methoden die vollen Methodennamen als Kommentar vor die daraus resultierenden Funktionen in die Abstraktion geschrieben. Diese Maßnahmen dienen der besseren Zuordnung der Elemente der Spezifikation zum Ausgangscode. Der Kommentarmodus ist standardmäßig abgeschaltet.
- **-s<n> (short identifier mode)**: Die Option -s1 weist den Übersetzer an, Bezeichner für Klassen und Variablen soweit wie möglich, durch das Weglassen von Paket-, Klassen- bzw. Methodennamen, zu kürzen. Diese Option ist standardmäßig aktiviert.
- **-o<n> (optimization mode)**: Die Option -o1 sorgt dafür, daß nur die Algorithmen übersetzt werden, die das Synchronisationsverhalten beeinflussen. Diese Option ist standardmäßig deaktiviert.

- **-am<n> (analyse monitor behaviour)**: Diese Option steuert die Analyse des Monitorverhaltens. Bei der Angabe von -am1 wird das Monitorverhalten innerhalb der Zielspezifikation modelliert, bei -am0 wird dies unterlassen. Diese Option ist standardmäßig eingeschaltet.
- **-aw<n> (analyse wait/notify behaviour)**: Diese Option steuert analog zur "am"- Option die Analyse des wait/notify Verhaltens. Diese Option ist standardmäßig eingeschaltet.
- **-minVal<n>**: Mittels "minVal" läßt sich der minimale Wert für numerische Kanäle in der resultierenden Spezifikation einstellen. <n> kann hierbei einen beliebigen Wert zwischen -2147483647 und 2147483647 annehmen. Ist diese Option beim Programmaufruf nicht angegeben, wird der Wert 0 angenommen.
- **-maxVal<n>**: Analog zur Definition von minVal beschreibt diese Option den maximalen Wert, den numerische Kanäle in der Zielspezifikation annehmen können. Der Standardwert beträgt 10.
- **-maxObj<n>**: Diese Option steuert die maximale Anzahl der Objekte einer Klasse. Alle Kanäle, die Objektwerte enthalten können minimal den Wert 0 und maximal den angegebenen Wert <n> annehmen. Ist diese Option nicht angegeben, wird der Wert 1 angenommen.
- **-i<Klasse$_1$;Klasse$_2$;...;Klasse$_n$> (classes to ignore)**: Diese Option legt fest, welche Klassen bei der Analyse ignoriert werden. Hierbei sollten die Klassen angegeben werden, die zwar innerhalb des zu analysierenden Projektes über Synchronisationseigenschaften verfügen, jedoch für die Spezifikation nicht von Interesse sind, damit eine möglichst kompakte Spezifikation erzeugt werden kann. Die Klassennamen werden hierbei in paketorientierter Form angegeben und durch Semikola voneinander getrennt (z.B. `java.lang.Thread;java.lang.ThreadGroup`). Standardmäßig werden die Klassen `java.lang.Thread`, `java.lang.ThreadGroup`, `java.lang.String`, `java.lang.StringBuffer` und `java.lang.SecurityManager` ausgeblendet.
- -**f<Dateiname> (output filename)**: Diese Option legt den Namen der zu erzeugenden Ausgabedatei fest. Ist die Option "-f" nicht angeben, wird der Name der zu analysierenden Datei benutzt, wobei das Dateinamensuffix durch ".fdr2" ersetzt wird.

Optimierung des Java Bytecodes

Mit dem JAVA2CSP- System lassen sich alle erzeugten Bytecodedarstellungen von Java Programmen übersetzen. Ein besonders schönes Ergebnis erreicht man, wenn das zu übersetzende Projekt mit Debugging- Informationen in die Class- Datei Form übersetzt wurde. Bei den JDK- Compilern läßt sich dies über die Angabe der Option "-g" bei der Übersetzung erreichen.
Die Debugging- Informationen liefern z.B. die Namen lokaler Variablen, die von JAVA2CSP benutzt werden können, um sie als Bezeichner in die zu erzeugende Abstraktion einzubringen, um so deren Lesbarkeit zu erhöhen.

Optimierung erzeugter CSP/FDR Abstraktionen

Nach der erfolgreichen Übersetzung eines Java- Projektes in eine CSP/FDR Spezifikation, sollte diese nachbearbeitet werden, um ein möglichst effizientes Model- Checking des Systems zu ermöglichen. Der Aufwand des Model- Checkings ist direkt an die Größe des durch die Spezifikation erzeugten Experimentraumes des FDR- Werkzeugs gebunden, woraus folgt, daß eine möglichst kompakte Spezifikation mit wenigen Zuständen schneller zu Ergebnissen führt.
Um dies zu erreichen werden für jede Spezifikation eine Reihe von Konstanten angelegt, die nachträglich verändert werden sollten, da sie zur Übersetzungszeit nicht ermittelt werden können. Dies betrifft z.B. die Anzahl möglicher erzeugbarer Objekte innerhalb einer Spezifikation, die zur Laufzeit festgelegt wird, jedoch statisch in einer CSP/FDR Spezifi-

kation vorliegen muß. Die folgende Tabelle zeigt die einzelnen Konstanten, deren Bedeutung, und die Werte, die optimalerweise hierfür eingetragen werden.

Konstante	Bedeutung	Optimaler Wert
`MAX_NUM`	maximaler Wert, für numerische Variablen	Der größte numerische Wert in der Spezifikation, der einer Variablen zugewiesen wird.
`MIN_NUM`	minimaler Wert, für numerische Variablen	Der kleinste numerische Wert in der Spezifikation, der einer Variablen zugewiesen wird.
`MAX_LEN_` `<FeldVariable>`	maximale Länge des Feldes mit der Bezeichnung "FeldVariable"	Für alle bekannten Felder, die im Quellprogramm mittels `new Typ[n]` erzeugt wurden, werden die Längen durch den Übersetzer korrekt angegeben. Über Felder, die z.B. als Parameter von Methoden benutzt werden, sind keine Informationen über die Feldlänge vorhanden. Der optimale Wert ist für diese Felder also die maximale Feldlänge, mit der der entsprechende Parameter der Methoden im Quellprogramm aufgerufen wird.
`MAX_OBJ_` `<Klasse>`	maximale Anzahl der Objekte vom Typ "Klasse"	Den optimalen Wert für diese Konstante stellt die maximale Anzahl der im Java Programm mittels `new` erzeugten Objekte des Typs dar.

Abbildung A-1: Einstellbare Konstanten in erzeugten Spezifikationen

Das JAVA2CSP *Frontend*

Um die Handhabung des Übersetzers zu erleichtern, existiert eine einfache Benutzungsoberfläche. Diese kann mittels des Kommandozeilenaufrufs `"java java2cspFront.java2cspFrontend"` gestartet werden. Das Frontend bietet die im folgenden dargestellte Oberfläche:

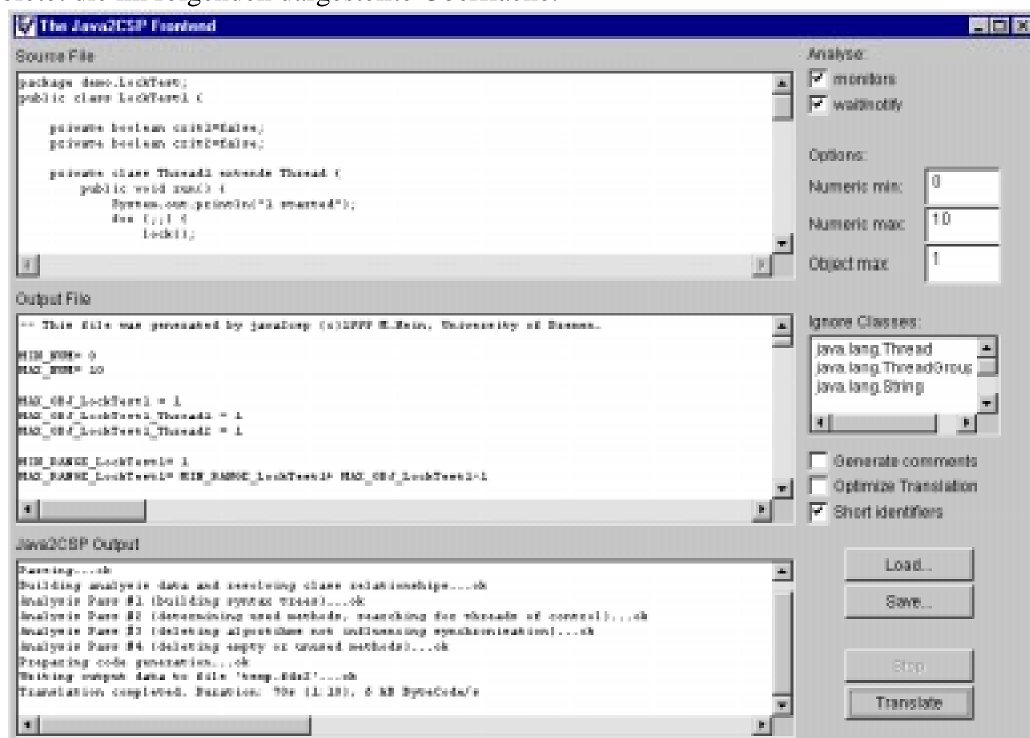

Abbildung A-2: Das JAVA2CSP *Frontend*

Einen Überblick über die einzelnen Elemente der Oberfläche und deren Funktionen gibt folgende Aufzählung:

- **Ausgabefelder**: Das Programm verfügt über die Textfelder `Source File`, `Output File` und `Java2CSP Output`, die die zu übersetzende Java Datei, die erzeugte Spe-

zifikation nach Ablauf der Übersetzung und die Ausgaben, die während des Übersetzungsvorgangs erzeugt wurde, anzeigen.

- **Analyse:** Mittels der Checkboxen `Analyse- monitors` und `Analyse- wait/notify` lassen sich die Optionen "-am" und "-aw" des Übersetzers steuern.
- **Options**: Die Eingabefelder `Numeric Min`, `Numeric Max` und `Object Max` legen die Parameter für `minVal`, `maxVal` und `maxObj` des Compilers fest. Im Textfeld `Ignore Classes` werden die Parameter der Option "-i" festgelegt. Die Checkboxen `Generate Comments`, `Short Method Names` und `Optimize Translation` steuern die Optionen "-c", "-s" und "-o".
- **Load:** Mittels des Buttons `Load` läßt sich eine Java Datei laden, die im Textfeld `Source File` angezeigt wird. Diese dient dann als Ausgangspunkt für die Übersetzung.
- **Save**: Der Button `Save` dient nach einer erfolgreichen Übersetzung dazu, die erzeugte Spezifikation zu speichern.
- **Stop**: Die Betätigung des `Stop`- Buttons bricht eine gestartete Übersetzung ab.
- **Translate**: Der Button `Translate` startet den Übersetzungsprozeß. Hierbei wird der Name der zu analysierenden Klasse aus der geladenen Quellcodedatei ermittelt und die entsprechende Klassendatei zur Übersetzung herangezogen.

Anhang B

Verwendete Bezeichner

Das Übersetzungssystem benutzt, zusätzlich zu den Schlüsselwörtern von FDR, eine Reihe von reservierten Bezeichnern. Diese Bezeichner dürfen in den Java- Programmen, die übersetzt werden sollen verwendet werden, da diese bei Bedarf durch Paket oder Methodennamen qualifiziert werden (siehe auch 'Erzeugung einer textuellen Repräsentation' auf Seite 81). Alle Bezeichner, die sich auf unterschiedliche Objekte innerhalb einer Abstraktion beziehen, können so immer eindeutig sein.
Im folgenden werden alle vom System verwendeten Bezeichner aufgelistet, wobei diese nach suffixlosen und suffixbehafteten Bezeichnern getrennt dargestellt sind.

Bezeichner	Bedeutung
`BAND`	Name der bitweisen AND Funktion in Java2CSPLib.fdr2
`BOR`	Name der bitweisen OR Funktion in Java2CSPLib.fdr2
`BXOR`	Name der bitweisen XOR Funktion in Java2CSPLib.fdr2
`arrSize`	Bezeichner für die Feldlänge eines Feldes in ID- Definitionen
`BitSeqToInt`	Name einer Hilfsfunktion in Java2CSPLib.fdr2 zur Erzeugung eines ganzzahligen Werts aus einer Bitsequenz
`BoolArrIDs`	Mengenbezeichner für alle booleschen nicht statischen Feldvariablen
`BoolIDs`	Mengenbezeichner für alle booleschen nicht statischen Variablen
`BSHL`	Name der bitweisen Linksshift- Funktion in Java2CSPLib.fdr2
`BSHR`	Name der bitweisen Rechtsshift- Funktion in Java2CSPLib.fdr2
`BUSHR`	Name der bitweisen nicht vorzeichenbehafteten Rechtsshift Funktion in Java2CSPLib.fdr2
`BXOR`	Name der bitweisen XOR Funktion in Java2CSPLib.fdr2
`destroyT`	Kanalbezeichner für das Ereignis"destroy"
`IntToBitSeq`	Name einer Hilfsfunktion in Java2CSPLib.fdr2 zur Erzeugung einer Bitsequenz aus einem ganzzahligen Wert
`joinT`	Kanalbezeichner für das Ereignis "join"
`MAX_ARRAY_LEN`	Konstante für die maximale Länge von Feldern
`MAX_NUM`	Konstante für die maximale Belegung numerischer Kanäle in einer Spezifikation
`MAX_OBJ`	Konstante für die maximale Anzahl benutzter Objekte in einer Spezifikation
`MIN_NUM`	Konstante für die minimale Belegung numerischer Kanäle in einer Spezifikation
`Monitor`	Funktion, die korrektes Monitorverhalten definiert (Bestandteil von Java2CSPLib.fdr2)
`monitorEnter`	Name der Monitor Enter- Funktion in der Spezifikation
`monitorenter`	Kanalbezeichner für das Ereignis "monitor enter"
`monitorExit`	Name der Monitor Exit- Funktion in der Spezifikation

Bezeichner	Bedeutung
monitorexit	Kanalbezeichner für das Ereignis "monitor exit"
MonitorIDs	Mengenbezeichner für alle Monitorvariablen
NewObjectIDGenerator	Name der Funktion zum Erzeugen neuer Objekt- IDs in Java2CSPLib.fdr2
notifyAllT	Kanalbezeichner für das Ereignis "notify All"
notifyT	Kanalbezeichner für das Ereignis "notify"
NumArrIDs	Mengenbezeichner für alle numerischen nicht statischen Feldvariablen
NumIDs	Mengenbezeichner für alle numerischen nicht statischen Variablen
ObjectArrIDs	Mengenbezeichner für alle objektwertigen nicht statischen Feldvariablen
ObjectIDs	Mengenbezeichner für alle objektwertigen nicht statischen Variablen
objId	Bezeichner für die Menge der Objekt- IDs einer Variable in ID- Definitionen
PM	Prozeßbezeichner des Main- Prozesses in Spezifikationen
put	Funktion zum Speichern von booleschen und numerischen Werten in Kanäle (Bestandteil von Java2CSPLib.fdr2)
putArr	Funktion zum Speichern von booleschen und numerischen Werten in Feld- Kanäle (Bestandteil von Java2CSPLib.fdr2)
putObj	Funktion zum Speichern von Objekt- IDs in Kanäle (Bestandteil von Java2CSPLib.fdr2)
putObjArr	Funktion zum Speichern von Objekt- IDs in Feld- Kanäle (Bestandteil von Java2CSPLib.fdr2)
readBool	Auslesen eines nicht statischen Kanals booleschen Typs
readBoolArr	Auslesen eines nicht statischen Feld- Kanals booleschen Typs
readNewObjectID	Kanal zur Übergabe neu erzeugter Objekt- IDs
readNum	Auslesen eines nicht statischen Kanals numerischen Typs
readObj	Auslesen eines nicht statischen Kanals, der Objekt- IDs enthält
readObjArr	Auslesen eines nicht statischen Feld- Kanals, der Objekt- IDs enthält
readSbool	Auslesen eines statischen Kanals booleschen Typs
readSboolArr	Auslesen eines statischen Feld- Kanals booleschen Typs
readSnum	Auslesen eines statischen Kanals numerischen Typs
readSobj	Auslesen eines statischen Kanals, der Objekt- IDs enthält
readSobjArr	Auslesen eines statischen Feld- Kanals, der Objekt- IDs enthält
resumeT	Kanalbezeichner für das Ereignis "resume"
return	Parametername für Übergabe von Rückgabeewerten
startT	Kanalbezeichner für das Ereignis"start"
StaticBoolArrIDs	Mengenbezeichner für alle booleschen statischen Feldvariablen
StaticBoolIDs	Mengenbezeichner für alle booleschen statischen Variablen
StaticNumArrIDs	Mengenbezeichner für alle numerischen statischen Feldvariablen
StaticNumIDs	Mengenbezeichner für alle numerischen statischen Variablen
StaticObjectArrIDs	Mengenbezeichner für alle objektwertigen statischen Feldvariablen
StaticObjectIDs	Mengenbezeichner für alle objektwertigen statischen Variablen
stopT	Kanalbezeichner für das Ereignis"stop"
suspendT	Kanalbezeichner für das Ereignis"suspend"
SynchroIDs	Mengenbezeichner für alle Synchronisationsvariablen
this	Parameterbezeichner für "this"- Variable
Thread	Funktion, die korrektes Threadverhalten definiert (Bestandteil von Java2CSPLib.fdr2)
ThreadArr	Funktion, die korrektes Threadverhalten für Felder definiert (Bestandteil von Java2CSPLib.fdr2)

Bezeichner	Bedeutung
`ThreadIDs`	Mengenbezeichner für alle Threadvariablen
`VarBool`	Funktion, die eine boolesche nicht- statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarBoolArr`	Funktion, die ein boolesches nicht- statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarNum`	Funktion, die eine numerische nicht- statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarNumArr`	Funktion, die ein numerisches nicht- statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarObj`	Funktion, die eine objektwerige nicht- statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarObjArr`	Funktion, die ein objektwertiges nicht- statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSbool`	Funktion, die eine boolesche statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSboolArr`	Funktion, die ein boolesches statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSnum`	Funktion, die eine numerische statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSnumArr`	Funktion, die ein numerisches statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSobj`	Funktion, die eine objektwertige statische Variable emuliert (Bestandteil von Java2CSPLib.fdr2)
`VarSobjArr`	Funktion, die ein objektwertiges statisches Feld emuliert (Bestandteil von Java2CSPLib.fdr2)
`WaitNotify`	Funktion, die korrektes Synchronisationsverhalten definiert (Bestandteil von Java2CSPLib.fdr2)
`waitT`	Kanalbezeichner für das Ereignis"wait"
`writeBool`	Schreiben auf einen nicht- statischen Kanal booleschen Typs
`writeBoolArr`	Schreiben auf einen nicht- statischen Feld- Kanal booleschen Typs
`writeNum`	Schreiben auf einen nicht- statischen Kanal numerischen Typs
`writeObj`	Schreiben auf einen nicht- statischen Kanal, der Objekt- IDs enthält
`writeObjArr`	Schreiben auf einen nicht- statischen Feld- Kanal, der Objekt- IDs enthält
`writeSbool`	Schreiben auf einen statischen Kanal booleschen Typs
`writeSboolArr`	Schreiben auf einen statischen Feld- Kanal booleschen Typs
`writeSnum`	Schreiben auf einen statischen Kanal numerischen Typs
`writeSobj`	Schreiben auf einen statischen Kanal, der Objekt- IDs enthält
`writeSobjArr`	Schreiben auf einen statischen Feld- Kanal, der Objekt- IDs enthält

Abbildung B-1: Suffixlose Bezeichner

Bezeichner	Suffix	Bedeutung
`i`	0..n	allgemeine CSP Iterations- und Indexvariable
`local`	0..n	Bezeichner für lokale Variablen einer Funktion
`localFun`	0..n	Name lokaler Funktionen
`MAX_LEN_`	Feldbezeichner	Konstante für maximale Länge eines Feldes
`MAX_OBJ_`	Klassenbezeichner	Konstante für maximale Anzahl von Objekten einer Klasse
`MAX_RANGE_`	Feldbezeichner	Konstante für maximale Anzahl von Objekt IDs einer Klasse

Bezeichner	Suffix	Bedeutung
MIN_RANGE_	Feldbezeichner	Konstante für minimale Anzahl von Objekt IDs einer Klasse
objectID_	Klassenbezeichner	Bezeichner für Objekt ID- Parameter einer Funktion
P	0..n	Bezeichner für Definitionen einzelner Spezifikationsprozesse
param	0..n	Bezeichner für primitive Parameter einer Funktion
returned	Methodenname	Kanalbezeichner für Rückgabewerte
t	0..n	CSP Variable für Objekt IDs
x	0..n	allgemeine CSP Variable

Abbildung B-2: Suffixbehaftete Bezeichner

Anhang C

Java2CSP Bibliothek

Dieser Anhang enthält die komplette Bibliothek, die jede von JAVA2CSP erzeugte Abstraktion benutzt. Sie beschreibt spezielle arithmetische Funktionen, die von FDR nicht direkt zur Verfügung gestellt werden und gibt Spezifikationen für das gewünschte Verhalten von Kontrollfäden wieder.

```
----------------------------------------------------------------------
-- THE JAVA2CSP Library
-- (c) M.Hein, last changes 04.10.1999
----------------------------------------------------------------------

----------------------------------------------------------------------
-- some java bytecode bit operations, not directly supported by fdr2
----------------------------------------------------------------------

-- calcs the bit representation from a given Integer value as sequence
-- sign is coded in the first bit of each sequenze (positive=0, negative=1)
-- e.g. -27=<111011>, 22= <001101>
IntToBitSeq(val) =
let
           IntToBitSeqHelp(0,seq) = seq
           IntToBitSeqHelp(val,seq)= IntToBitSeqHelp((val/2),(seq^<(val%2)>))
within
           if(val>=0) then <0>^IntToBitSeqHelp(val,<>)
           else <1>^IntToBitSeqHelp(-val,<>)

-- calcs an integer value from a given bit sequence
-- sign is coded in the first bit of each sequenze (positive=1, negative=0)
-- e.g. <011011>=27, <101101>=-22
BitSeqToInt(seq)=
let
          BitSeqToIntHelp(<>,bitVal)  = 0
         BitSeqToIntHelp(seq,bitVal) = (head(seq)*bitVal) +
                                          BitSeqToIntHelp(tail(seq),bitVal*2)
within
   if(head(seq)==0) then BitSeqToIntHelp(tail(seq),1)
   else -BitSeqToIntHelp(tail(seq),1)


-- bitwise OR
BOR (val1, val2) =
let
   BORor(val1,val2) = if(val1==1 or val2==1) then 1
                                         else 0
   BORhelp(<>,<>)=<>
   BORhelp(<>,seq2)=seq2
   BORhelp(seq1,<>)=seq1
   BORhelp(seq1,seq2)= <BORor(head(seq1),head(seq2))>^
                                      BORhelp(tail(seq1),tail(seq2))
within
   BitSeqToInt( BORhelp(IntToBitSeq(val1), IntToBitSeq(val2)) )
```

```
-- bitwise AND
BAND (val1, val2) =
let
   BANDand(val1,val2)= if(val1==1 and val2==1) then 1
                                            else 0
   BANDhelp(<>,<>)=<>
   BANDhelp(<>,seq2) = <>
   BANDhelp(seq1,<>) = <>
   BANDhelp(seq1,seq2)= <BANDand(head(seq1),head(seq2))>^
                           BANDhelp(tail(seq1),tail(seq2))
within
   BitSeqToInt( BANDhelp(IntToBitSeq(val1), IntToBitSeq(val2)) )

-- bitwise XOR
BXOR (val1, val2) =
let
   BXORhelp(<>,<>)=<>
   BXORhelp(<>,seq2) = BXORhelp(<0>,seq2)
   BXORhelp(seq1,<>) = BXORhelp(seq1,<0>)
   BXORhelp(seq1,seq2)= <(head(seq1) +
       head(seq2))%2>^BXORhelp(tail(seq1),tail(seq2))
within
   BitSeqToInt( BXORhelp(IntToBitSeq(val1), IntToBitSeq(val2)) )

-- left shift
BSHL (val, 0)= val
BSHL (val, s)= BSHL(val*2,s-1)

-- right shift
BSHR (val,0)  = val
BSHR (val, s) = BSHR(val/2,s-1)

-- rightshift without sign extension
BUSHR (val, s) = if(BSHR(val, s)<0) then -BSHR(val,s)
              else BSHR(val, s)

-- maximum function on sequence of numbers
maxNum(seq) =
let
        maxNumHelp (<>,x) = x
        maxNumHelp (seq,x) = if ( head(seq) <=x ) then maxNumHelp(tail(seq), x)
                                          else maxNumHelp( tail(seq), head(seq) )
within
   maxNumHelp(seq,0)

----------------------------------------------------------------------
-- defintions for correct thread behaviour
----------------------------------------------------------------------

-- definition for correct behaviour of a monitor
Monitor(var) = monitorenter.var -> monitorexit.var -> Monitor(var)



-- definition for correct wait/notify behaviour of a thread
WaitNotify(var) =
let
      NotifyBehaviour(var) =( notifyT.var ->
                              ( NotifyBehaviour(var)
                                []
                                WaitNotify(var) )
                              )
                              []
                              ( notifyAllT.var ->
                              ( NotifyBehaviour(var)
                                []
                                WaitNotify(var) )
                               )
```

```
within
   waitT.var -> NotifyBehaviour(var)
   []
   NotifyBehaviour(var)

-- definition for correct start/stop, suspend/ resume behaviour of a thread
Thread(var) =
let
              ThreadPause(var) = (suspendT.var -> resumeT.var -> ThreadPause(var))
              []
              (resumeT.var -> ThreadPause(var))
              []
              SKIP
within
   startT.var ->
   ThreadPause(var);
   (
      stopT.var ->
      (
         SKIP
         []
         (joinT.var -> SKIP)
      )
      []
      destroyT.var ->
      (
         SKIP
         []
         (joinT.var -> SKIP)
      )
   )

-- definition for correct start/stop behaviour for an array
ThreadArr(var,len) = ||| i : {0..len} @ Thread(i.var)

----------------------------------------------------------------------
-- processes for variable simulation
----------------------------------------------------------------------

-- definition for non array global number variable
VarNum(var,val,min,max) =
               writeNum.var?x -> x>=MIN_NUM and x<=MAX_NUM & VarNum(var,x,min,max)
               []
               readNum.var!val -> VarNum(var,val,min,max)

-- definition for non array global boolean variable
VarBool(var,val) =writeBool.var?x -> VarBool(var,x)
                  []
                  readBool.var!val -> VarBool(var,val)

-- definition for non array global object variable
VarObj(var,val,maxVal)=writeObj.var?x -> x<=maxVal & VarObj(var,x,maxVal)
                        []
                        readObj.var!val -> VarObj(var,val,maxVal)

-- definition for static non array number variable
VarSnum(var,val,min,max) =
             writeSnum.var?x -> x>=MIN_NUM and x<=MAX_NUM & VarSnum(var,x,min,max)
             []
             readSnum.var!val -> VarSnum(var,val,min,max)




-- definition for static non array boolean variable
VarSbool(var,val) =writeSbool.var?x -> VarSbool(var,x)
                   []
                   readSbool.var!val -> VarSbool(var,val)
```

```
-- definition for static non array object variable
VarSobj(var,val,maxVal)=
                  writeSobj.var?x -> x<=maxVal & VarSobj(var,x,maxVal)
                  []
                  readSobj.var!val -> VarSobj(var,val,maxVal)

-- definition for global number array variable
VarNumArr(var,val,min,max,len) =
let
                  VarNumElt(var,val) = writeNumArr.var?x -> x>=MIN_NUM and
                                         x<=MAX_NUM & VarNumElt(var,x)
                                     []
                                     readNumArr.var!val -> VarNumElt(var,val)
within
             ||| i : {0..len} @ VarNumElt(i.var,val)

-- definition for global boolean array variable
VarBoolArr(var,val,len) =
let
   VarBoolElt(var,val) =writeBoolArr.var?x -> VarBoolElt(var,x)
                        []
                        readBoolArr.var!val -> VarBoolElt(var,val)
within
            ||| i : {0..len} @ VarBoolElt(i.var,val)

-- definition for global object array variable
VarObjArr(var,val,maxVal,len)=
let
   VarObjElt(var,val) =   writeObjArr.var?x -> VarObjElt(var,x)
                         []
                         readObjArr.var!val -> VarObjElt(var,val)
within
            ||| i : {0..len} @ VarObjElt(i.var,val)

-- definition for static number array variable
VarSnumArr(var,val,min,max,len) =
let
   VarSnumElt(var,val) =writeSnumArr.var?x ->
                              x>=MIN_NUM and x<=MAX_NUM & VarSnumElt(var,x)
                         []
                         readSnumArr.var!val -> VarSnumElt(var,val)
within
            ||| i : {0..len} @ VarSnumElt(i.var,val)

-- definition for static  boolean array variable
VarSboolArr(var,val,len) =
let
   VarSboolElt(var,val) =writeSboolArr.var?x -> VarSboolElt(var,x)
                          []
                          readSboolArr.var!val -> VarSboolElt(var,val)
within
            ||| i : {0..len} @ VarSboolElt(i.var,val)


-- definition for static object array variable
VarSobjArr(var,val,maxVal,len)=
let
   VarSobjElt(var,val)=  writeSobjArr.var?x -> VarSobjElt(var,x)
                       []
                       readSobjArr.var!val -> VarSobjElt(var,val)
within
            ||| i : {0..len} @ VarSobjElt(i.var,val)
```

```
------------------------------------------------------------------------
-- definitions for writing data into channels
-------------------------------------------------------------------------

-- definition for writing numeric and boolean data into channels
put(var,val) =
  if member(var,NumIDs) then val>=MIN_NUM and val<=MAX_NUM &
      writeNum.var!val -> SKIP
  else if member(var,BoolIDs) then val>=0 and val<=1 & writeBool.var!val -> SKIP
  else if member(var,StaticNumIDs) then
      val>=MIN_NUM and val<=MAX_NUM & writeSnum.var!val -> SKIP
  else val>=0 and val<=1 & writeSbool.var!val -> SKIP

-- definition for writing numeric and boolean data into array channels
putArr(var,index,val,maxIndex)=
  if member(index.var,NumArrIDs) then index<=maxIndex and
      val>=MIN_NUM and val<=MAX_NUM & writeNumArr.index.var!val -> SKIP
  else if member(index.var,BoolArrIDs) then index<=maxIndex and val>=0 and
      val<=1 & writeBoolArr.index.var!val -> SKIP
  else if member(index.var,StaticNumArrIDs) then index<=maxIndex and
      val>=MIN_NUM and val<=MAX_NUM & writeSnumArr.index.var!val -> SKIP
  else index<=maxIndex and val>=0 and val<=1 & writeSboolArr.index.var!val -> SKIP

-- definition for writing object data into channels
putObj(var,val,minVal,maxVal) =
   if member(var,ObjectIDs) then val<=maxVal and val>=minVal &
      writeObj.var!val -> SKIP
   else val<=maxVal and val>=minVal & writeSobj.var!val -> SKIP

-- definition for writing object data into array channels
putArrObj(var,val,minVal,maxVal,maxIndex) =
   if member(index.var,ObjectIDs) then index<=maxIndex and val>=minVal and
      val<=maxVal & writeObj.index.var!val -> SKIP
   else index<=maxIndex and val>=minVal and val<=maxVal &
      writeSobj.index.var!val -> SKIP


-------------------------------------------------------------------------
-- object id generator
-------------------------------------------------------------------------

channel readNewObjectID:CLASSES.{0..MAX_OBJ}
NewObjectIDGenerator(class,val,max) = val<=max & readNewObjectID.class!val ->
      NewObjectIDGenerator(class,val+1,max)
```
Abbildung C-1: JAVA2CSP *Bibliothek*

Anhang D

FDR Grammatik

Dieser Anhang beschreibt die komplette Grammatik von FDR 1.42 (vgl. [Formal95]). Leider ist für die aktuelle Version von FDR keine Grammatikreferenz verfügbar, so daß hier auf die der Version 1.42 zurückgegriffen werden muß. Diese bietet aber nichtsdestotrotzt einen guten Einblick in die FDR Syntax.

```
begin:
        [ \n ] linelist
linelist:
        line
        error
        line \n [ linelist ]
        error \n [ linelist ]
line:
        any == any
        PRAGMA
proc:
        justproc
        unknown
bool:
        justbool
        unknown
exp:
        justexp
        unknown
any:
        justproc
        justbool
        justexp
        unknown
justbool:
        ! bool
        bool AND bool
        bool OR bool
        bool == bool
        bool != bool
        bool <= bool
        bool < bool
        bool >= bool
unknown:
        IF bool THEN any ELSE any
        LET [linelist] WITHIN any
        \ args @ any
        simple
justproc:
        || exp : exp @ [ exp ] proc
        |~| exp : exp @ proc
        [] exp : exp @ proc
        ||| exp : exp @ proc
        proc \ exp
        proc ||| proc
        proc [| exp |] proc
        proc [ exp || exp ] proc
        proc |~| proc
        proc [] proc
        proc ; proc
        exp transfer proc
        STOP
        SKIP
simple:
        lname
        uname
        ( any [ , anys ] )
        simple args
        simple [[ [ sublist ] ]]
        simple [[ sublist | gens ]]
sublist:
        subst
        subst ( , subst)*
subst:
        any = any
targ:
        exps
```

```
transfer:
        ->
        ? exp transfer
        ? exp  ,  exp transfer
        ! exp transfer
justexp:
        exp . exp
        exp + exp
        exp - exp
        exp * exp
        exp % exp
        exp / exp
        # exp
        exp ^ exp
        < targ >
        < targ | gens >
        ( targ )
        ( targ | gens )
        ( exps )
        ( exps | gens )
        number
gens:
        gen
        gen ( , gen)*
gen:
        any = any
        bool
anys:
        any
        any ( , any)*
exps:
        exp
        exp ( , exp)*
args:
        ( anys )
```

Abbildung D-1: FDR- Grammatik

Anhang E

Klassendiagramme

Dieser Anhang enthält die Klassendiagramme der einzelnen Komponenten des Übersetzers. Sie sollen einen Eindruck über die Kapselung und die interne Organisation der Implementierung geben.

Parser

Das folgende Diagramm zeigt die Vererbungs- und Implementierungszusammenhänge der einzelnen Klassen des Parsers. Auf Aggregationen und Angabe der einzelnen Klassenattribute wurde hierbei bewußt verzichtet, um das Diagramm nicht zu komplex zu gestalten.

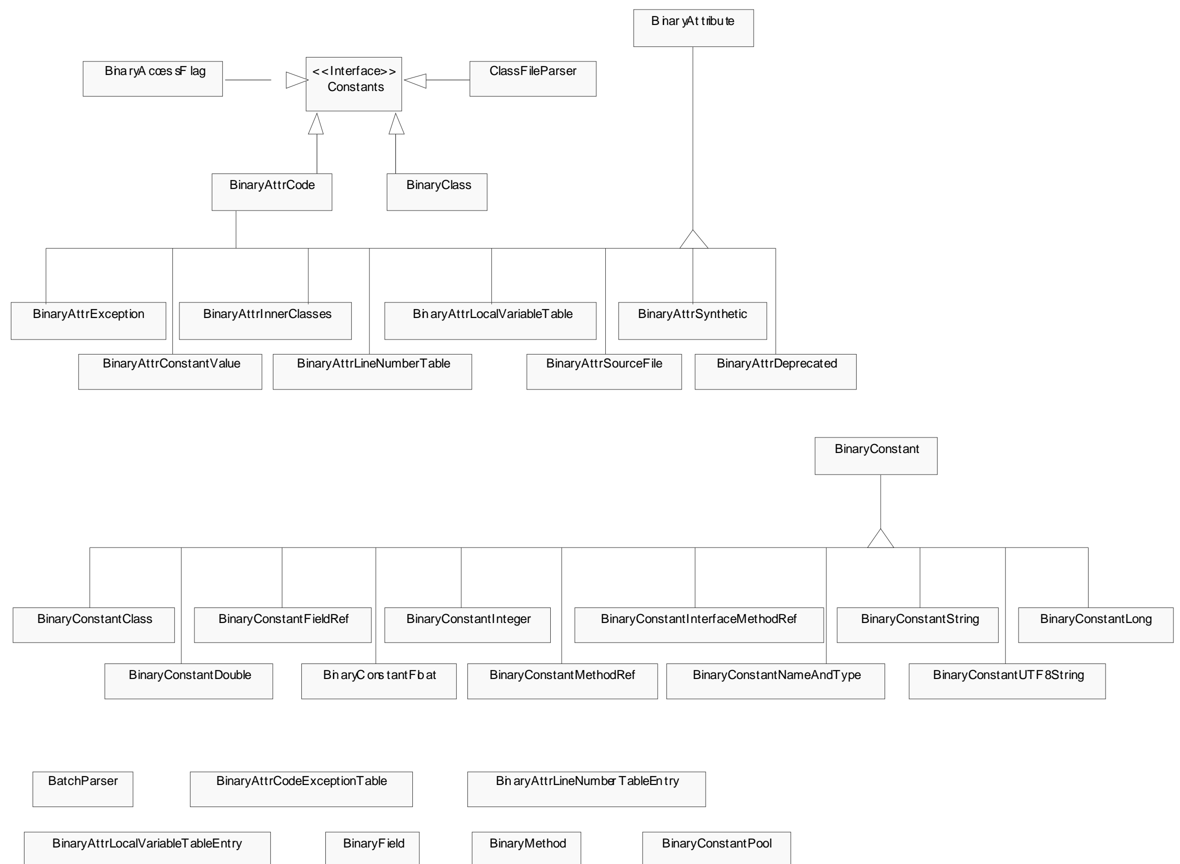

Abbildung E-1: Klassendiagramm Parser

Im Klassendiagramm spiegelt sich die Struktur des Java Bytecode Formats direkt wider. Sehr gut zu erkennen sind die beiden Basisklassen `BinaryAttribute` und `BinaryConstant`. Alle vorhandenen Klassen zur Ablage von Bytecodeattributen erben von `BinaryAttribute`, alle Klassen zur Speicherung der verschiedenen Konstanten

von `BinaryConstant`. `BinaryAttrCodeExceptionTable`, `BinaryAttrLineNumberTableEntry` und `BinaryAttrLocalVariableTableEntry` werden innerhalb des Diagramms nicht referenziert. Dies liegt daran, daß sie nicht von `BinaryAttribute` erben, sondern durch `BinaryAttrCode` aggregiert werden.
Das Interface `Constants` stellt die im Bytecode verwendeten numerischen Konstanten zur Verfügung, deren Semantik in den Klassen `BinaryAccessFlag`, `ClassFileParser`, `BinaryAttrCode` und `BinaryClass` zum tragen kommt.

Analyse und Codegenerierung

Das folgende Klassendiagramm zeigt die Zusammenhänge der Interfaces und Klassen der Analysephase und Codegenerierung des Systems. Auch hier wurde auf Aggregationen und Angabe der einzelnen Klassenattribute aus Gründen der Übersichtlichkeit verzichtet.

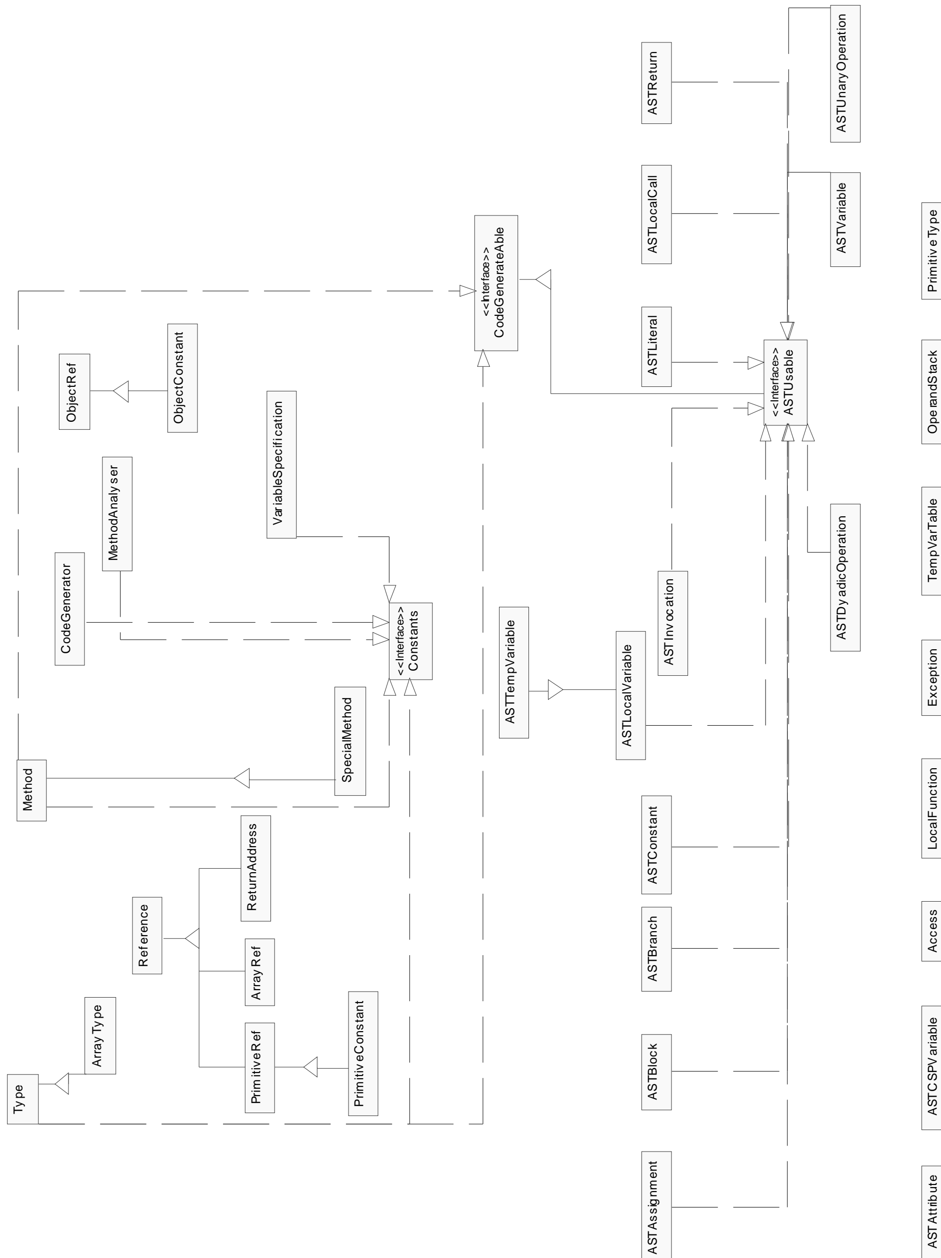

Abbildung E-2: Klassendiagramm Analyse und Codegenerierung

In diesem Diagramm wird der Aufbau von Typen sowie Referenzen und Konstanten innerhalb der Bytecodeanalysephase deutlich. Sie sind strukturell von den später im Syntaxbaum vorkommenden Elementen (alle Klassen, deren Namen mit `AST` beginnen) getrennt. Das Interface `Constants` stellt alle für die Analyse des Codesegments benötigten Opcode- Konstanten zur Verfügung. Alle Elemente, aus denen sich während der Codeerzeugungsphase Elemente der CSP/FDR Spezifikation generieren lassen, implementieren das Interface `CodeGenerateAble`, alle Elemente der abstrakten Syntax `ASTUsable`, das ebenfalls `CodeGenerateAble` erweitert. `CodeGenerateAble` stellt alle statischen Bezeichner für die Codegenerierung zur Verfügung (z.B. Schlüsselwörter, Namen für zusätzliche CSP Variablen, etc.)

Aggregationsdiagramm der abstrakten Syntax

Das folgende Diagramm zeigt die Beziehungen ausgewählter Klassen zur Darstellung der abstrakten Syntax. Hierbei sind nur die Aggregationen dargestellt, da diese einen guten Überblick über die Anordnung der Elemente der abstrakten Syntax bieten.

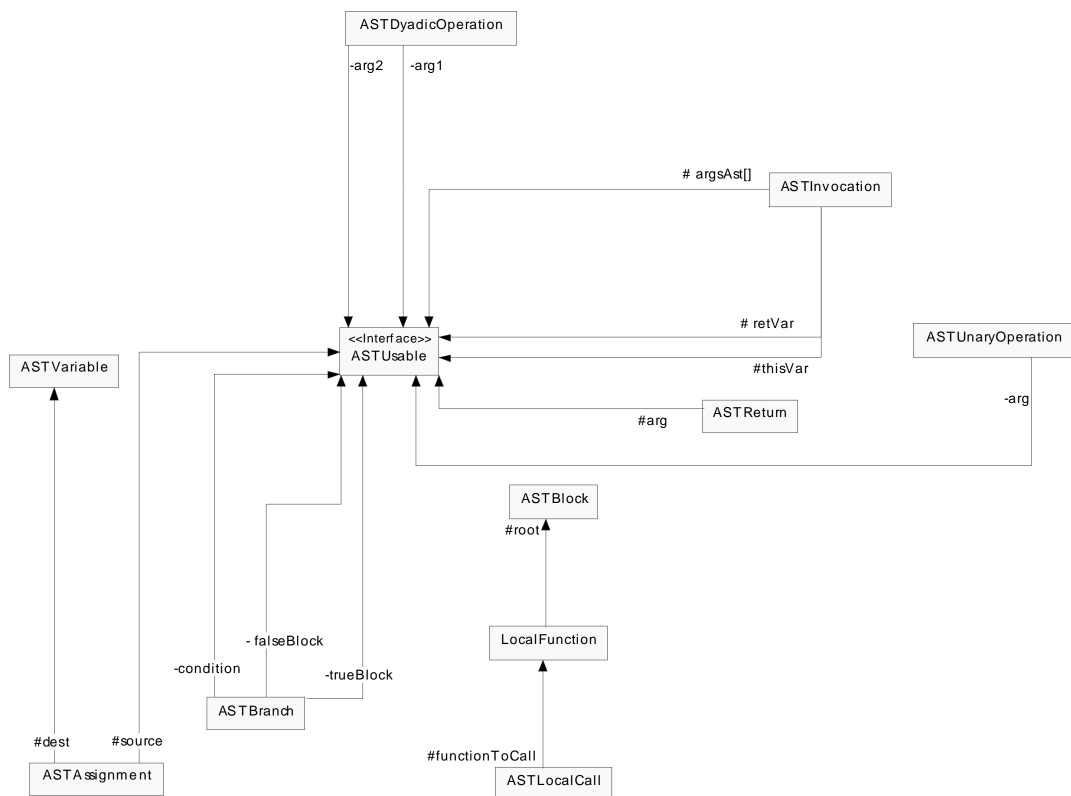

Abbildung E-3: Aggregationsdiagramm der abstrakten Syntax

Die Basis für alle Elemente der abstrakten Syntax stellt das Interface `ASTUsable` dar. Es dient zur Abtraktion der Elementtypen der einzelnen Syntaxbaumelemente, wenn es sich um ein beliebiges Element der abstrakten Syntax handeln kann. Dies ist z.B. beim Attribut `argsAst[]` der Fall, das die Argumente eines Aufrufs darstellt, die von jeden beliebigen Syntaxbaumelement sein können.
Interessant ist ebenfalls die Aggregation eines `ASTBlock` innerhalb von `LocalFuntion`. Hierdurch wird deutlich, daß jede lokale Funktion tatsächlich einen einzelnen Syntaxbaum besitzt, der aus dem Syntaxbaum der definierenden Methode extrahiert wird.

Anhang F

Beispiele

Dieser Abschnitt zeigt einige beispielhafte Übersetzungen mit dem JAVA2CSP- System für anschließende Verifikationen mit dem FDR- Werkzeug.

Leser-/Schreiber- Problem

Das folgende Beispiel zeigt die übersetzte Form des Leser-/ Schreiber Problems, wie es im Abschnitt “Beispiele” auf Seite 22 vorgestellt wurde. Es veranschaulicht, das naheliegendste Anwendungsgebiet für das JAVA2CSP- Werkzeugs: die Überprüfung der Synchronisation zwischen Threads. Eine Verifikation mit FDR2 bestätigt, daß das Beispiel im Bezug auf Threadsynchronisation tatsächlich Deadlock- frei ist.

```
-- This file was generated by java2csp (c)1999 M.Hein, University of Bremen.
-- minimum and maximum numeric values
MIN_NUM= 0
MAX_NUM= 10

-- maximum object number per class
MAX_OBJ_Buffer = 1
MAX_OBJ_Producer = 1
MAX_OBJ_Consumer = 1

-- object ranges
MIN_RANGE_Buffer= 1
MAX_RANGE_Buffer= MIN_RANGE_Buffer+ MAX_OBJ_Buffer-1
MIN_RANGE_Producer= MAX_RANGE_Buffer+1
MAX_RANGE_Producer= MIN_RANGE_Producer+ MAX_OBJ_Producer-1
MIN_RANGE_Consumer= MAX_RANGE_Producer+1
MAX_RANGE_Consumer= MIN_RANGE_Consumer+ MAX_OBJ_Consumer-1
MAX_OBJ= MAX_RANGE_Consumer

-- class type definitions
Buffer = 0
Producer = 1
Consumer = 2

CLASSES={
  Buffer,
  Producer,
  Consumer
 }

-- variable definitions
Consumer_buffer = 0
Producer_buffer = 1
available = 2
c = 3
c1 = 4
mainArgs = 5
p1 = 6
```

```
-- channel definitions
ObjectIDs= {
  objId0.Producer_buffer,
  objId1.Consumer_buffer
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ }
}
channel readObj, writeObj: ObjectIDs.{ 0..MAX_OBJ }

ObjectArrIDs= { }

StaticObjectIDs= {
  c,
  c1,
  p1
}
channel readSobj, writeSobj: StaticObjectIDs.{ 0..MAX_OBJ }

StaticObjectArrIDs= { }

NumIDs= { }

NumArrIDs= { }

BoolIDs= {
  objId0.available
  | objId0<-{ 0..MAX_OBJ }
}
channel readBool, writeBool: BoolIDs.{ 0, 1 }

BoolArrIDs= { }

StaticNumIDs= { }

StaticNumArrIDs= { }

StaticBoolIDs= { }

StaticBoolArrIDs= { }

ThreadIDs= {
  c1,
  p1
}
channel startT, stopT, destroyT, resumeT, suspendT, joinT: ThreadIDs

MonitorIDs=
  { MIN_RANGE_Buffer..MAX_RANGE_Buffer }
channel monitorenter, monitorexit: MonitorIDs

SynchroIDs=
  { MIN_RANGE_Buffer..MAX_RANGE_Buffer }
channel waitT, notifyT, notifyAllT: SynchroIDs


include "java2cspLib.fdr2"


Buffer__init_(this,objectID_this) =
  put(objectID_this.available, 0)

Producer__init_(this,objectID_this,b,objectID_b) =
  putObj(objectID_this.Producer_buffer, objectID_b,
         MIN_RANGE_Buffer, MAX_RANGE_Buffer)

Consumer__init_(this,objectID_this,b,objectID_b) =
  putObj(objectID_this.Consumer_buffer, objectID_b,
         MIN_RANGE_Buffer, MAX_RANGE_Buffer)
```

```
get(this,objectID_this) =
let
localFun0=
    monitorExit(this, objectID_this)
within
  monitorEnter(this, objectID_this) ;
  readBool.objectID_this.available?x0 ->
  if (0 != x0) then (
    put(objectID_this.available, 0) ;
    notify(this, objectID_this) ;
    localFun0
  )
  else (
    wait(this, objectID_this) ;
    put(objectID_this.available, 0) ;
    notify(this, objectID_this) ;
    localFun0
  )

main(ARRAY_args,objectID_ARRAY_args) =
  readSobj.c?t0 ->
  new(c, t0, Buffer) ;
  readSobj.c?t0 ->
  Buffer__init_(c, t0) ;
  readSobj.p1?t0 ->
  new(p1, t0, Producer) ;
  readSobj.p1?t0 ->
  readSobj.c?t1 ->
  Producer__init_(p1, t0, c, t1) ;
  readSobj.c1?t0 ->
  new(c1, t0, Consumer) ;
  readSobj.c1?t0 ->
  readSobj.c?t1 ->
  Consumer__init_(c1, t0, c, t1) ;
  readSobj.c1?t0 ->
  start(c1, t0) ;
  readSobj.p1?t0 ->
  start(p1, t0)

monitorEnter(this,objectID_this) =
  member(objectID_this,MonitorIDs) & monitorenter.objectID_this -> SKIP

monitorExit(this,objectID_this) =
  member(objectID_this,MonitorIDs) & monitorexit.objectID_this -> SKIP

new(this,objectID_this,local0) =
  readNewObjectID.local0?x ->
  if member(this,ObjectIDs) then writeObj.this!x -> SKIP
  else if member(this,ObjectArrIDs) then writeObjArr.this!x -> SKIP
  else if member(this,StaticObjectIDs) then writeSobj.this!x -> SKIP
  else writeSobjArr.this!x -> SKIP

notify(this,objectID_this) =
  member(objectID_this,SynchroIDs) & notifyT.objectID_this -> SKIP

Buffer_put(this,objectID_this) =
let
localFun0=
    monitorExit(this, objectID_this)
within
  monitorEnter(this, objectID_this) ;
  readBool.objectID_this.available?x0 ->
  if (x0 != 1) then (
    put(objectID_this.available, 1) ;
    notify(this, objectID_this) ;
    localFun0
  )
  else (
```

```
    wait(this, objectID_this) ;
    put(objectID_this.available, 1) ;
    notify(this, objectID_this) ;
    localFun0
  )

Consumer_run(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.Consumer_buffer?t0 ->
    get(objectID_this.Consumer_buffer, t0) ;
    localFun0
within
  readObj.objectID_this.Consumer_buffer?t0 ->
  get(objectID_this.Consumer_buffer, t0) ;
  localFun0

Producer_run(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.Producer_buffer?t0 ->
    Buffer_put(objectID_this.Producer_buffer, t0) ;
    localFun0
within
  readObj.objectID_this.Producer_buffer?t0 ->
  Buffer_put(objectID_this.Producer_buffer, t0) ;
  localFun0

start(this,objectID_this) =
  startT.this -> SKIP

wait(this,objectID_this) =
  member(objectID_this,SynchroIDs) & waitT.objectID_this -> SKIP

-- System specification
PM = main(mainArgs, 0)
P0 = startT.c1 -> readSobj.c1?t -> Consumer_run(c1, t)
P1 = startT.p1 -> readSobj.p1?t -> Producer_run(p1, t)

SYSTEM=  ( ( ( ( ( ( (
( PM [| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |]( P0 ||| P1 )  )
[| {| readBool, writeBool |} |]
   ( ( ||| i0 : { MIN_RANGE_Buffer..MAX_RANGE_Buffer } @
         VarBool(i0.available,0) ) ) )
[| {| readObj, writeObj |} |]
   ( ( ||| i0 : { MIN_RANGE_Producer..MAX_RANGE_Producer } @
         VarObj(i0.Producer_buffer, MIN_RANGE_Buffer, MAX_RANGE_Buffer) )
   ||| ( ||| i1 : { MIN_RANGE_Consumer..MAX_RANGE_Consumer } @
         VarObj(i1.Consumer_buffer, MIN_RANGE_Buffer, MAX_RANGE_Buffer) ) ) )
[| {| readSobj, writeSobj |} |]
   ( VarSobj(c, MIN_RANGE_Buffer, MAX_RANGE_Buffer)
   ||| VarSobj(p1, MIN_RANGE_Producer, MAX_RANGE_Producer)
   ||| VarSobj(c1, MIN_RANGE_Consumer, MAX_RANGE_Consumer) ) )
[| {| monitorenter, monitorexit |} |]
   ( ( ||| i0 : { MIN_RANGE_Buffer..MAX_RANGE_Buffer } @ Monitor(i0) ) ) )
[| {| waitT, notifyT, notifyAllT |} |]
   ( ( ||| i0 : { MIN_RANGE_Buffer..MAX_RANGE_Buffer } @ WaitNotify(i0) ) ) )
[| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |]
   ( Thread(c1) ||| Thread(p1) ) )
[| {| readNewObjectID |} |]
   ( NewObjectIDGenerator(Buffer, MIN_RANGE_Buffer, MAX_RANGE_Buffer)
   ||| NewObjectIDGenerator(Producer, MIN_RANGE_Producer, MAX_RANGE_Producer)
   ||| NewObjectIDGenerator(Consumer, MIN_RANGE_Consumer, MAX_RANGE_Consumer) )  )

assert SYSTEM :[deadlock free [F]]
assert SYSTEM :[livelock free [F]]
```

Abbildung F-1: JAVA2CSP- Übersetzung des Leser-/Schreiber- Problems

Schloßalgorithmen

Das JAVA2CSP- Werkzeug eignet sich jedoch nicht nur dazu, Übersetzungen von Java nach CSP/FDR mit dem Ziel der Überprüfung von Synchronisationseigenschaften durchzuführen, sondern ermöglicht ebenfalls die Überprüfung von funktionalen Eigenschaften von Algorithmen.

Als Beispiel hierfür betrachten wir zwei Schloßalgorithmen mit dem Ziel des gegenseitigen Ausschlusses, wie sie in [HeHo94] auf den Seiten 185 und 191 zu finden sind, wobei der erste dieser Algorithmen den gegenseitigen Ausschluß nicht durchsetzt, während der zweite dies bewerkstelligt.

Eine in CSP/FDR übersetzte Umsetzung beider Algorithmen in Java wird im folgenden mit einigen kleinen manuellen Erweiterungen auf die Erfüllung des Ausschlußkriteriums überprüft.

Das folgende Java Programm zeigt den "naiven" Schloßalgorithmus mit je einer Zustandsvariable für zwei konkurrierende Threads, die auf einen kritischen Abschnitt zugreifen wollen.

```
package demo.LockTest;
public class LockTest1 {
   private boolean crit1=false;
   private boolean crit2=false;

   private class Thread1 extends Thread {
      public void run() {
         System.out.println("1 started");
         for (;;) {
            lock();
            System.out.println("crit1");
            unlock();
         }
      }

      public void lock() {
          while(crit2);
          crit1=true;
      }

      public void unlock() {
         crit1=false;
      }
   }

   private class Thread2 extends Thread {
      public void run() {
         System.out.println("2 started");
         for (;;) {
            lock();
            System.out.println("crit2");
            unlock();
         }
      }

      public void lock() {
         while(crit1);
         crit2=true;
      }
      public void unlock() {
         crit2=false;
      }
   }

    public LockTest1() {
      Thread1 t1=new Thread1();
      Thread2 t2=new Thread2();
      t1.start();
      t2.start();
    }
```

```
    public static void main(String args[]) {
      LockTest1 l=new LockTest1();
    }
}
```

Abbildung F-2: Java Umsetzung eines unsicheren Schloßalgorithmusses

Um zu verifizieren, daß dieser Algorithmus wirklich unsicher ist, übersetzen wir ihn mit dem JAVA2CSP- Werkzeug nach CSP/FDR und fügen zusätzlich die Bedingung ein, daß beide Prozesse ihre kritischen Abschnitte nicht gleichzeitig betreten dürfen. Dies entspricht folgender Funktionsdefinition, wobei das System stoppt, wenn dieser Tatbestand erfüllt sein sollte:

```
check=
   in -> (
      out -> check
      []
      in -> STOP
   )
```

Die beiden Ereignisse `in` und `out` sind hierbei die Indikatoren für das Betreten bzw. Verlassen der kritischen Abschnitte, wobei diese ebenfalls manuell der erzeugten Abstraktion an den richtigen Stellen zugeführt werden müssen, so daß sich folgende Abstraktion ergibt, in dem das FDR- Werkzeug das Vorkommen des unsicheren Zustands `in->in` festgestellt hat:

```
-- This file was generated by java2csp (c)1999 M.Hein, University of Bremen.
-- minimum and maximum numeric values
MIN_NUM= 0
MAX_NUM= 10

-- maximum object number per class
MAX_OBJ_LockTest1 = 1
MAX_OBJ_LockTest1_Thread1 = 1
MAX_OBJ_LockTest1_Thread2 = 1

-- object ranges
MIN_RANGE_LockTest1= 1
MAX_RANGE_LockTest1= MIN_RANGE_LockTest1+ MAX_OBJ_LockTest1-1
MIN_RANGE_LockTest1_Thread1= MAX_RANGE_LockTest1+1
MAX_RANGE_LockTest1_Thread1= MIN_RANGE_LockTest1_Thread1+
MAX_OBJ_LockTest1_Thread1-1
MIN_RANGE_LockTest1_Thread2= MAX_RANGE_LockTest1_Thread1+1
MAX_RANGE_LockTest1_Thread2= MIN_RANGE_LockTest1_Thread2+
MAX_OBJ_LockTest1_Thread2-1
MAX_OBJ= MAX_RANGE_LockTest1_Thread2

-- class type definitions
LockTest1 = 0
LockTest1_Thread1 = 1
LockTest1_Thread2 = 2

CLASSES={
  LockTest1,
  LockTest1_Thread1,
  LockTest1_Thread2
 }

-- variable definitions
LockTest1_Thread1_this_0 = 0
LockTest1_Thread2_this_0 = 1
LockTest1__init__t1 = 2
LockTest1__init__t2 = 3
access_0_returned = 4
access_1_returned = 5
crit1 = 6
crit2 = 7
mainArgs = 8
main_local1 = 9
```

```
-- channel definitions
ObjectIDs= {
  objId0.LockTest1__init__t1,
  objId1.LockTest1__init__t2,
  objId2.LockTest1_Thread1_this_0,
  objId3.LockTest1_Thread2_this_0
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ },
    objId2<-{ 0..MAX_OBJ },
    objId3<-{ 0..MAX_OBJ }
}
channel readObj, writeObj: ObjectIDs.{ 0..MAX_OBJ }

ObjectArrIDs= { }

StaticObjectIDs= {
  main_local1
}
channel readSobj, writeSobj: StaticObjectIDs.{ 0..MAX_OBJ }

StaticObjectArrIDs= { }

NumIDs= { }

NumArrIDs= { }

BoolIDs= {
  objId0.crit1,
  objId1.crit2,
  objId2.access_1_returned,
  objId3.access_0_returned
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ },
    objId2<-{ 0..MAX_OBJ },
    objId3<-{ 0..MAX_OBJ }
}
channel readBool, writeBool: BoolIDs.{ 0, 1 }

BoolArrIDs= { }

StaticNumIDs= { }

StaticNumArrIDs= { }

StaticBoolIDs= { }

StaticBoolArrIDs= { }

ThreadIDs= {
  objId0.LockTest1__init__t1,
  objId1.LockTest1__init__t2
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ }
}
channel startT, stopT, destroyT, resumeT, suspendT, joinT: ThreadIDs

MonitorIDs= { }

SynchroIDs= { }

include "java2cspLib.fdr2"


LockTest1__init_(this,objectID_this) =
  put(objectID_this.crit1, 0) ;
  put(objectID_this.crit2, 0) ;
  readObj.objectID_this.LockTest1__init__t1?t0 ->
  new(objectID_this.LockTest1__init__t1, t0, LockTest1_Thread1) ;
```

```
  if (0 != objectID_this) then (
    readObj.objectID_this.LockTest1__init__t1?t0 ->
    LockTest1_Thread1__init_(objectID_this.LockTest1__init__t1, t0,
                             this, objectID_this) ;
    readObj.objectID_this.LockTest1__init__t2?t0 ->
    new(objectID_this.LockTest1__init__t2, t0, LockTest1_Thread2) ;
    if (0 != objectID_this) then (
      readObj.objectID_this.LockTest1__init__t2?t0 ->
      LockTest1_Thread2__init_(objectID_this.LockTest1__init__t2, t0,
                               this, objectID_this) ;
      readObj.objectID_this.LockTest1__init__t1?t0 ->
      start(objectID_this.LockTest1__init__t1, t0) ;
      readObj.objectID_this.LockTest1__init__t2?t0 ->
      start(objectID_this.LockTest1__init__t2, t0)
    )
    else
      SKIP
  )
  else
    SKIP

LockTest1_Thread1__init_(this,objectID_this,local0,objectID_local0) =
  putObj(objectID_this.LockTest1_Thread1_this_0, objectID_local0,
          MIN_RANGE_LockTest1, MAX_RANGE_LockTest1)

LockTest1_Thread2__init_(this,objectID_this,local0,objectID_local0) =
  putObj(objectID_this.LockTest1_Thread2_this_0, objectID_local0,
          MIN_RANGE_LockTest1, MAX_RANGE_LockTest1)

access_0(this,objectID_this,return) =
  readBool.objectID_this.crit2?x ->
  put(return, x)

access_1(this,objectID_this,return) =
  readBool.objectID_this.crit1?x ->
  put(return, x)

access_2(this,objectID_this,local0) =
  put(objectID_this.crit1, local0)

access_3(this,objectID_this,local0) =
  put(objectID_this.crit2, local0)

LockTest1_Thread2_lock(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.LockTest1_Thread2_this_0?t0 ->
    access_1(objectID_this.LockTest1_Thread2_this_0, t0,
             objectID_this.access_1_returned) ;
    readBool.objectID_this.access_1_returned?x0 ->
    if (0 != x0) then
      localFun0
    else (
      readObj.objectID_this.LockTest1_Thread2_this_0?t0 ->
      access_3(objectID_this.LockTest1_Thread2_this_0, t0, 1)
    )
within
  localFun0

LockTest1_Thread1_lock(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.LockTest1_Thread1_this_0?t0 ->
    access_0(objectID_this.LockTest1_Thread1_this_0,
            t0, objectID_this.access_0_returned) ;
    readBool.objectID_this.access_0_returned?x0 ->
    if (0 != x0) then
      localFun0
```

```
    else (
      readObj.objectID_this.LockTest1_Thread1_this_0?t0 ->
      access_2(objectID_this.LockTest1_Thread1_this_0, t0, 1)
    )
within
  localFun0

main(ARRAY_args,objectID_ARRAY_args) =
  readSobj.main_local1?t0 ->
  new(main_local1, t0, LockTest1) ;
  readSobj.main_local1?t0 ->
  LockTest1__init_(main_local1, t0)

new(this,objectID_this,local0) =
  readNewObjectID.local0?x ->
  if member(this,ObjectIDs) then writeObj.this!x -> SKIP
  else if member(this,ObjectArrIDs) then writeObjArr.this!x -> SKIP
  else if member(this,StaticObjectIDs) then writeSobj.this!x -> SKIP
  else writeSobjArr.this!x -> SKIP

LockTest1_Thread2_run(this,objectID_this) =
let
localFun0=
    out->  -- ADDED BY HAND
    LockTest1_Thread2_unlock(this, objectID_this) ;
    LockTest1_Thread2_lock(this, objectID_this) ;
    in->   -- ADDED BY HAND
    localFun0
within
  LockTest1_Thread2_lock(this, objectID_this) ;
  in->  -- ADDED BY HAND
  localFun0

LockTest1_Thread1_run(this,objectID_this) =
let
localFun0=
    out->  -- ADDED BY HAND
    LockTest1_Thread1_unlock(this, objectID_this) ;
    LockTest1_Thread1_lock(this, objectID_this) ;
    in->  -- ADDED BY HAND
    localFun0
within
  LockTest1_Thread1_lock(this, objectID_this) ;
  in->  -- ADDED BY HAND
  localFun0

start(this,objectID_this) =
  startT.this -> SKIP

LockTest1_Thread2_unlock(this,objectID_this) =
  readObj.objectID_this.LockTest1_Thread2_this_0?t0 ->
  access_3(objectID_this.LockTest1_Thread2_this_0, t0, 0)

LockTest1_Thread1_unlock(this,objectID_this) =
  readObj.objectID_this.LockTest1_Thread1_this_0?t0 ->
  access_2(objectID_this.LockTest1_Thread1_this_0, t0, 0)

-- System specification
PM = main(mainArgs, 0)
P0 = ( ||| i0 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
       startT.i0.LockTest1__init__t1 -> readObj.i0.LockTest1__init__t1?t ->
       LockTest1_Thread1_run(i0.LockTest1__init__t1, t) )
P1 = ( ||| i0 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
       startT.i0.LockTest1__init__t2 -> readObj.i0.LockTest1__init__t2?t ->
       LockTest1_Thread2_run(i0.LockTest1__init__t2, t) )


SYSTEM=  ( ( ( ( (
```

```
 PM [| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |]  ( P0 ||| P1 ) )
 [| {| readBool, writeBool |} |]
    ( ( ||| i0 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
       VarBool(i0.crit1,0) ) |||
    ( ||| i1 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
      VarBool(i1.crit2,0) ) |||
    ( ||| i2 : { MIN_RANGE_LockTest1_Thread2..MAX_RANGE_LockTest1_Thread2 } @
      VarBool(i2.access_1_returned,0) ) |||
    ( ||| i3 : { MIN_RANGE_LockTest1_Thread1..MAX_RANGE_LockTest1_Thread1 } @
      VarBool(i3.access_0_returned,0) ) ) )
 [| {| readObj, writeObj |} |]
    ( ( ||| i0 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
      VarObj(i0.LockTest1__init__t1, MIN_RANGE_LockTest1_Thread1,
             MAX_RANGE_LockTest1_Thread1) ) |||
    ( ||| i1 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
      VarObj(i1.LockTest1__init__t2, MIN_RANGE_LockTest1_Thread2,
             MAX_RANGE_LockTest1_Thread2) ) |||
    ( ||| i2 : {MIN_RANGE_LockTest1_Thread1..MAX_RANGE_LockTest1_Thread1 } @
      VarObj(i2.LockTest1_Thread1_this_0, MIN_RANGE_LockTest1,
            MAX_RANGE_LockTest1) ) |||
    ( ||| i3 : { MIN_RANGE_LockTest1_Thread2..MAX_RANGE_LockTest1_Thread2 } @
      VarObj(i3.LockTest1_Thread2_this_0, MIN_RANGE_LockTest1,
            MAX_RANGE_LockTest1) ) ) )
 [| {| readSobj, writeSobj |} |]
    ( VarSobj(main_local1, MIN_RANGE_LockTest1, MAX_RANGE_LockTest1) ) )
 [| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |]
    ( ( ||| i0 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
                Thread(i0.LockTest1__init__t1) ) |||
    ( ||| i1 : { MIN_RANGE_LockTest1..MAX_RANGE_LockTest1 } @
                Thread(i1.LockTest1__init__t2) ) ) )
 [| {| readNewObjectID |} |]
   ( NewObjectIDGenerator(LockTest1, MIN_RANGE_LockTest1, MAX_RANGE_LockTest1) |||
   NewObjectIDGenerator(LockTest1_Thread1, MIN_RANGE_LockTest1_Thread1,
         MAX_RANGE_LockTest1_Thread1) |||
   NewObjectIDGenerator(LockTest1_Thread2, MIN_RANGE_LockTest1_Thread2,
         MAX_RANGE_LockTest1_Thread2) )  )

-- ADDED BY HAND
[|{|in,out|}|] check
channel in,out
check=
   in-> (
      out->check
      []
      in->STOP
   )

assert SYSTEM :[deadlock free [F]]
assert SYSTEM :[livelock free [F]]
```

Abbildung F-3: JAVA2CSP- Übersetzung des unsicheren Schloßalgorithmusses

Das folgende Programm zeigt die Java Umsetzung eines sicheren Schloßalgorithmus für zwei konkurrierende Threads.

```
package demo.LockTest;
public class LockTest2 {
    private boolean crit1=false;
    private boolean crit2=false;
    private int favourite=1;

    private class Thread1 extends Thread {
      public void run() {
         System.out.println("1 started");
         for (;;) {
            lock();
            System.out.println("crit1");
            unlock();
         }
      }
```

```
      public void lock() {
         crit1=true;
         while(crit2) {
            if(favourite!=1) {
               crit1=false;
               while(favourite!=1);
               crit1=true;
            }
         }
      }

      public void unlock() {
         crit1=false;
         favourite=2;
      }
    }

    private class Thread2 extends Thread {
      public void run() {
         System.out.println("2 started");
         for (;;) {
            lock();
            System.out.println("crit2");
            unlock();
         }
      }

      public void lock() {
         crit2=true;
         while(crit1) {
            if(favourite!=2) {
               crit2=false;
               while(favourite!=2);
               crit2=true;
            }
         }
      }

      public void unlock() {
         crit2=false;
         favourite=1;
      }
   }

   public LockTest2() {
      Thread1 t1=new Thread1();
      Thread2 t2=new Thread2();
      t1.start();
      t2.start();
   }

   public static void main(String args[]) {
      LockTest2 l=new LockTest2();
   }
}
```

Abbildung F-4: Java Umsetzung eines sicheren Schloßalgorithmusses

Für die nach CSP/FDR übersetzte Version dieses Programms fügen wir ebenfalls wieder unsere Sicherheitsspezifikation ein, und können mittels FDR feststellen, daß der Algorithmus wirklich gegenseitigen Ausschluß für zwei konkurrierende Threads garantiert.

```
-- This file was generated by java2csp (c)1999 M.Hein, University of Bremen.
-- minimum and maximum numeric values
MIN_NUM= 0
MAX_NUM= 10

-- maximum object number per class
MAX_OBJ_LockTest2 = 1
```

```
MAX_OBJ_LockTest2_Thread1 = 1
MAX_OBJ_LockTest2_Thread2 = 1

-- object ranges
MIN_RANGE_LockTest2= 1
MAX_RANGE_LockTest2= MIN_RANGE_LockTest2+ MAX_OBJ_LockTest2-1
MIN_RANGE_LockTest2_Thread1= MAX_RANGE_LockTest2+1
MAX_RANGE_LockTest2_Thread1= MIN_RANGE_LockTest2_Thread1+
MAX_OBJ_LockTest2_Thread1-1
MIN_RANGE_LockTest2_Thread2= MAX_RANGE_LockTest2_Thread1+1
MAX_RANGE_LockTest2_Thread2= MIN_RANGE_LockTest2_Thread2+
MAX_OBJ_LockTest2_Thread2-1
MAX_OBJ= MAX_RANGE_LockTest2_Thread2

-- class type definitions
LockTest2 = 0
LockTest2_Thread1 = 1
LockTest2_Thread2 = 2

CLASSES={
  LockTest2,
  LockTest2_Thread1,
  LockTest2_Thread2
 }

-- variable definitions
LockTest2_Thread1_this_0 = 0
LockTest2_Thread2_this_0 = 1
LockTest2__init__t1 = 2
LockTest2__init__t2 = 3
access_0_returned = 4
access_2_returned = 5
access_3_returned = 6
crit1 = 7
crit2 = 8
favourite = 9
mainArgs = 10
main_local1 = 11

-- channel definitions
ObjectIDs= {
  objId0.LockTest2__init__t1,
  objId1.LockTest2__init__t2,
  objId2.LockTest2_Thread1_this_0,
  objId3.LockTest2_Thread2_this_0
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ },
    objId2<-{ 0..MAX_OBJ },
    objId3<-{ 0..MAX_OBJ }
}
channel readObj, writeObj: ObjectIDs.{ 0..MAX_OBJ }

ObjectArrIDs= { }

StaticObjectIDs= {
  main_local1
}
channel readSobj, writeSobj: StaticObjectIDs.{ 0..MAX_OBJ }

StaticObjectArrIDs= { }

NumIDs= {
  objId0.favourite,
  objId1.access_3_returned
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ }
}
channel readNum, writeNum: NumIDs.{ MIN_NUM..MAX_NUM }
```

```
NumArrIDs= { }

BoolIDs= {
  objId0.crit1,
  objId1.crit2,
  objId2.access_0_returned,
  objId3.access_2_returned
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ },
    objId2<-{ 0..MAX_OBJ },
    objId3<-{ 0..MAX_OBJ }
}
channel readBool, writeBool: BoolIDs.{ 0, 1 }

BoolArrIDs= { }

StaticNumIDs= { }

StaticNumArrIDs= { }

StaticBoolIDs= { }

StaticBoolArrIDs= { }

ThreadIDs= {
  objId0.LockTest2__init__t1,
  objId1.LockTest2__init__t2
  | objId0<-{ 0..MAX_OBJ },
    objId1<-{ 0..MAX_OBJ }
}
channel startT, stopT, destroyT, resumeT, suspendT, joinT: ThreadIDs

MonitorIDs= { }

SynchroIDs= { }


include "java2cspLib.fdr2"


LockTest2__init_(this,objectID_this) =
  put(objectID_this.crit1, 0) ;
  put(objectID_this.crit2, 0) ;
  put(objectID_this.favourite, 1) ;
  readObj.objectID_this.LockTest2__init__t1?t0 ->
  new(objectID_this.LockTest2__init__t1, t0, LockTest2_Thread1) ;
  if (0 != objectID_this) then (
    readObj.objectID_this.LockTest2__init__t1?t0 ->
    LockTest2_Thread1__init_(objectID_this.LockTest2__init__t1, t0,
                           this, objectID_this) ;
    readObj.objectID_this.LockTest2__init__t2?t0 ->
    new(objectID_this.LockTest2__init__t2, t0, LockTest2_Thread2) ;
    if (0 != objectID_this) then (
      readObj.objectID_this.LockTest2__init__t2?t0 ->
      LockTest2_Thread2__init_(objectID_this.LockTest2__init__t2, t0,
                           this, objectID_this) ;
      readObj.objectID_this.LockTest2__init__t1?t0 ->
      start(objectID_this.LockTest2__init__t1, t0) ;
      readObj.objectID_this.LockTest2__init__t2?t0 ->
      start(objectID_this.LockTest2__init__t2, t0)
    )
    else
      SKIP
  )
  else
    SKIP
```

```
LockTest2_Thread1__init_(this,objectID_this,local0,objectID_local0) =
  putObj(objectID_this.LockTest2_Thread1_this_0, objectID_local0,
         MIN_RANGE_LockTest2, MAX_RANGE_LockTest2)

LockTest2_Thread2__init_(this,objectID_this,local0,objectID_local0) =
  putObj(objectID_this.LockTest2_Thread2_this_0, objectID_local0,
         MIN_RANGE_LockTest2, MAX_RANGE_LockTest2)

access_0(this,objectID_this,return) =
  readBool.objectID_this.crit1?x ->
  put(return, x)

access_1(this,objectID_this,local0) =
  put(objectID_this.crit1, local0)

access_2(this,objectID_this,return) =
  readBool.objectID_this.crit2?x ->
  put(return, x)

access_3(this,objectID_this,return) =
  readNum.objectID_this.favourite?x ->
  put(return, x)

access_4(this,objectID_this,local0) =
  put(objectID_this.favourite, local0)

access_5(this,objectID_this,local0) =
  put(objectID_this.crit2, local0)

LockTest2_Thread2_lock(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
    access_0(objectID_this.LockTest2_Thread2_this_0,
            t0, objectID_this.access_0_returned) ;
    readBool.objectID_this.access_0_returned?x0 ->
    if (0 != x0) then (
      readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
      access_3(objectID_this.LockTest2_Thread2_this_0,
               t0, objectID_this.access_3_returned) ;
      readNum.objectID_this.access_3_returned?x0 ->
      if (x0 == 2) then
        localFun0
      else (
        readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
        access_5(objectID_this.LockTest2_Thread2_this_0, t0, 0) ;
        localFun1
      )
    )
    else
      SKIP
localFun1=
    readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
    access_3(objectID_this.LockTest2_Thread2_this_0,
            t0, objectID_this.access_3_returned) ;
    readNum.objectID_this.access_3_returned?x0 ->
    if (x0 != 2) then
      localFun1
    else (
      readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
      access_5(objectID_this.LockTest2_Thread2_this_0, t0, 1) ;
      localFun0
    )
within
  readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
  access_5(objectID_this.LockTest2_Thread2_this_0, t0, 1) ;
  localFun0
```

```
LockTest2_Thread1_lock(this,objectID_this) =
let
localFun0=
    readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
    access_2(objectID_this.LockTest2_Thread1_this_0,
            t0, objectID_this.access_2_returned) ;
    readBool.objectID_this.access_2_returned?x0 ->
    if (0 != x0) then (
      readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
      access_3(objectID_this.LockTest2_Thread1_this_0,
              t0, objectID_this.access_3_returned) ;
      readNum.objectID_this.access_3_returned?x0 ->
      if (x0 == 1) then
        localFun0
      else (
        readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
        access_1(objectID_this.LockTest2_Thread1_this_0, t0, 0) ;
        localFun1
      )
    )
    else
      SKIP
localFun1=
    readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
    access_3(objectID_this.LockTest2_Thread1_this_0, t0,
objectID_this.access_3_returned) ;
    readNum.objectID_this.access_3_returned?x0 ->
    if (x0 != 1) then
      localFun1
    else (
      readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
      access_1(objectID_this.LockTest2_Thread1_this_0, t0, 1) ;
      localFun0
    )
within
  readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
  access_1(objectID_this.LockTest2_Thread1_this_0, t0, 1) ;
  localFun0

main(ARRAY_args,objectID_ARRAY_args) =
  readSobj.main_local1?t0 ->
  new(main_local1, t0, LockTest2) ;
  readSobj.main_local1?t0 ->
  LockTest2__init_(main_local1, t0)

new(this,objectID_this,local0) =
  readNewObjectID.local0?x ->
  if member(this,ObjectIDs) then writeObj.this!x -> SKIP
  else if member(this,ObjectArrIDs) then writeObjArr.this!x -> SKIP
  else if member(this,StaticObjectIDs) then writeSobj.this!x -> SKIP
  else writeSobjArr.this!x -> SKIP

LockTest2_Thread2_run(this,objectID_this) =
let
localFun0=
    out->  -- ADDED BY HAND
    LockTest2_Thread2_unlock(this, objectID_this) ;
    LockTest2_Thread2_lock(this, objectID_this) ;
    in->  -- ADDED BY HAND
    localFun0
within
  LockTest2_Thread2_lock(this, objectID_this) ;
  in->  -- ADDED BY HAND
  localFun0

LockTest2_Thread1_run(this,objectID_this) =
let
localFun0=
```

```
    out->  -- ADDED BY HAND
    LockTest2_Thread1_unlock(this, objectID_this) ;
    LockTest2_Thread1_lock(this, objectID_this) ;
    in->  -- ADDED BY HAND
    localFun0
within
  LockTest2_Thread1_lock(this, objectID_this) ;
  in->  -- ADDED BY HAND
  localFun0

start(this,objectID_this) =
  startT.this -> SKIP

LockTest2_Thread2_unlock(this,objectID_this) =
  readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
  access_5(objectID_this.LockTest2_Thread2_this_0, t0, 0) ;
  readObj.objectID_this.LockTest2_Thread2_this_0?t0 ->
  access_4(objectID_this.LockTest2_Thread2_this_0, t0, 1)

LockTest2_Thread1_unlock(this,objectID_this) =
  readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
  access_1(objectID_this.LockTest2_Thread1_this_0, t0, 0) ;
  readObj.objectID_this.LockTest2_Thread1_this_0?t0 ->
  access_4(objectID_this.LockTest2_Thread1_this_0, t0, 2)

-- System specification
PM = main(mainArgs, 0)
P0 = ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
         startT.i0.LockTest2__init__t1 -> readObj.i0.LockTest2__init__t1?t ->
         LockTest2_Thread1_run(i0.LockTest2__init__t1, t) )
P1 = ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
         startT.i0.LockTest2__init__t2 -> readObj.i0.LockTest2__init__t2?t ->
         LockTest2_Thread2_run(i0.LockTest2__init__t2, t) )

SYSTEM=  ( ( ( ( ( (
( PM [| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |] ( P0 ||| P1 ) )
[| {| readNum, writeNum |} |]
   ( ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
       VarNum(i0.favourite,0, MIN_NUM, MAX_NUM) ) |||
   ( ||| i1 : { MIN_RANGE_LockTest2_Thread2..MAX_RANGE_LockTest2_Thread2 } @
       VarNum(i1.access_3_returned,0, MIN_NUM, MAX_NUM) ) ) )
[| {| readBool, writeBool |} |]
   ( ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
       VarBool(i0.crit1,0) ) |||
   ( ||| i1 :{ MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @ VarBool(i1.crit2,0)|||
   ( ||| i2 : { MIN_RANGE_LockTest2_Thread2..MAX_RANGE_LockTest2_Thread2 } @
       VarBool(i2.access_0_returned,0) ) |||
   ( ||| i3 : { MIN_RANGE_LockTest2_Thread1..MAX_RANGE_LockTest2_Thread1 } @
       VarBool(i3.access_2_returned,0) ) ) )
[| {| readObj, writeObj |} |]
   ( ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
       VarObj(i0.LockTest2__init__t1, MIN_RANGE_LockTest2_Thread1,
       MAX_RANGE_LockTest2_Thread1) ) |||
   ( ||| i1 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
       VarObj(i1.LockTest2__init__t2, MIN_RANGE_LockTest2_Thread2,
       MAX_RANGE_LockTest2_Thread2) ) |||
   ( ||| i2 : { MIN_RANGE_LockTest2_Thread1..MAX_RANGE_LockTest2_Thread1 } @
       VarObj(i2.LockTest2_Thread1_this_0, MIN_RANGE_LockTest2,
       MAX_RANGE_LockTest2) ) |||
   ( ||| i3 : { MIN_RANGE_LockTest2_Thread2..MAX_RANGE_LockTest2_Thread2 } @
       VarObj(i3.LockTest2_Thread2_this_0, MIN_RANGE_LockTest2,
       MAX_RANGE_LockTest2) ) ) )
[| {| readSobj, writeSobj |} |]
   ( VarSobj(main_local1, MIN_RANGE_LockTest2, MAX_RANGE_LockTest2) ) )
[| {| startT, stopT, destroyT, resumeT, suspendT, joinT |} |]
   ( ( ||| i0 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
    Thread(i0.LockTest2__init__t1) ) |||
   ( ||| i1 : { MIN_RANGE_LockTest2..MAX_RANGE_LockTest2 } @
```

```
    Thread(i1.LockTest2__init__t2) ) ) )
[| {| readNewObjectID |} |]
   ( NewObjectIDGenerator(LockTest2, MIN_RANGE_LockTest2, MAX_RANGE_LockTest2) |||
   NewObjectIDGenerator(LockTest2_Thread1, MIN_RANGE_LockTest2_Thread1,
          MAX_RANGE_LockTest2_Thread1) |||
   NewObjectIDGenerator(LockTest2_Thread2, MIN_RANGE_LockTest2_Thread2,
          MAX_RANGE_LockTest2_Thread2) )  )

-- ADDED BY HAND
[|{|in,out|}|] check

channel in,out
check=
   in-> (
      out->check
      []
      in->STOP
   )

assert SYSTEM :[deadlock free [F]]
assert SYSTEM :[livelock free [F]]
```

Abbildung F-5: JAVA2CSP- Übersetzung des sicheren Schloßalgorithmusses