Institut für Informatik

Softwaretechnik und Programmiersprachen

Universitätsstr. 1 D-40225 Düsseldorf

Ein Parser für CSP_M in Java

Robin Bially

Bachelorarbeit

Beginn der Arbeit:	17. Mai 2016
Abgabe der Arbeit:	17. August 2016
Gutachter:	Prof. Dr. Michael Leuschel
	Prof. Dr. Jörg Rothe

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 16. August 2016 _____

Robin Bially

Zusammenfassung

Diese Arbeit dokumentiert die Entwicklung des neuen CSP_M-Parsers *cspmj*, der für das Validierungstool *ProB* des Lehrstuhls für Softwaretechnik und Programmiersprachen entwickelt wurde.

Der Hauptgrund für die Entwicklung eines neuen Parsers ist, dass der bisherige Parser *cspmf* im Jahr 2011 in der nur sehr wenig verbreiteten funktionalen Programmiersprache Haskell verfasst wurde und somit die zukünftige Fehlerbehandlung und Fortentwicklung behindern könnte.

Eine Herausforderung des Projekts ist die Erfassung und Abbildung aller wesentlichen Konstrukte der Spezifikationsalgebra CSP in einen Java-Parser, der mithilfe des Parsergenerators SableCC erzeugt wird. Der Entwicklungsaufwand ist dabei als besonders hoch einzustufen, was anhand der Grammatik des *FDR*-Parsers *libcspm* schnell zu erkennen ist. Die vielen Besonderheiten der Prozessalgebra gegenüber höheren Programmiersprachen sind vor der Implementierung differenziert und umsichtig zu durchdenken.

Im Vordergrund des Projekts steht der Nachweis, dass die Anbindung an *ProB*, sowie die Funktionalität und praxisnahe Laufzeit gewährleistet sind. Die Softwarequalität wird mithilfe der Open-Source Software *Gradle* und *jUnit* überprüft und optimiert. Dabei kommt *Gradle* beim automatisierten Kompilieren von Java-Dateien bzw. dem Herunterladen aller benötigten Tools und der automatisierten Projekterzeugung zum Einsatz. *jUnit* testet den Ausgabequellcode von *cspmj* gegenüber *cspmf* und ermöglicht so eine exakte Angleichung des Verhaltens unter Verwendung des *Reverse-Engineering*-Prinzips. Ebenfalls wird die kontinuierliche Integration erleichtert, bei der es darum geht, bereits bestehende Funktion zu validieren und im laufenden Entwicklungsprozess permanent automatisiert überwachen zu können.

Die Erstellung einer automatisierten Laufzeit-Testumgebung hat gezeigt, dass die Übersetzungszeit von praxisnahen Beispieldateien nicht wesentlich von der Zeit abweicht, die *cspmf* benötigt. Dazu wurden 130 CSP_M-Dateien geparst und die Zeiten beider Parser miteinander verglichen.

Die wesentlichen Vorteile von *cspmj* gegenüber *cspmf* sind:

- Größerer Funktionsumfang
- Bekanntere Programmiersprache mit potentiell höherer Weiterentwicklungsbereitschaft
- Plattformunabhängige Unterstützung durch die *Java Virtual Machine*
- Optimierte Prolog-Darstellung durch Behebung einiger Anzeigebugs
- Interpretation und Überschreiben aller eingebauten Funktionen und Konstanten auf die gleiche Weise

Das Projekt wird mithilfe von Github versioniert. Das Repository kann unter folgendem Link aufgerufen werden:

`https://github.com/RobinBia/CSPMJ`

Inhaltsverzeichnis

1. Einleitung und Motivation

1.1 Gliederung

In den Kapiteln 2 bis 6 werden Methoden vorgestellt, die in der Arbeit zum Einsatz kommen. Zuerst wird die Hauptmotivation für die Entwicklung eines neuen CSP_M-Parsers für das Tool ProB, erläutert. Kapitel 2 enthält eine kurze Beschreibung und Einführung in die Struktur, Funktionsweise und Anwendungspraxis der Prozessalgebra CSP_M. Kapitel 3 ist eine Einführung in die Grundlagen des Parsergenerators SableCC, der für die automatisierte Erzeugung der Java-Klassen verwendet wurde. Kapitel 4 benennt die Motivation und Spezifikation für die Verwendung des Build-Management-Automatisierungstools Gradle. In Kapitel 5 wird erläutert, wie durch eine automatisierte Testumgebung die Überprüfung der Integrität von CSP_M gewährleistet werden kann. Kapitel 6 ist eine Beschreibung der Architektur und sämtlicher Implementierungsdetails des neuen Parsers *cspmj*. Zum Schluss folgt in Kapitel 7 eine Analyse und Einschätzung der aktuellen Funktionalität von *cspmj* und eine Auflistung zukünftig relevanter Entwicklungsschritte.

1.2 ProB

Das 2007 von Michael Leuschel und Michael Butler vorgestellte Validierungstool ProB wurde zu dem Zweck der Überprüfung und Veranschaulichung von Spezifikationen der B-Methode entwickelt. Die B-Methode ist die Theorie und Methodologie für die formale Entwicklung von Computersystemen und kommt in der Industrie schwerpunktmäßig zur Steuerung von Bahnsignalen zum Einsatz [LB07]. Neben der B-Methode wurden die Sprachen Event-B, TLA+, und Z unterstützt [Mi16]. Die Integration von der Spezifikationssprache CSP_M erfolgte erst später mit der Absicht, CSP und B- Spezifikationen zu kombinieren. Eine B-Maschine kann als reaktives System betrachtet werden, das kontinuierlich Operationen unabhängig voneinander ausführt. Neben dem Vorteil der optimalen Modellierung paralleler Aktivität hat B jedoch den Nachteil, dass es sich nicht für die Modellierung von sequenzieller Aktivität eignet. CSP_M hingegen besitzt Operatoren wie die sequentielle Komposition und unterstützt mehrere synchrone Kommunikationsarten. Auf diesem Weg ergänzen sich B und CSP optimal und ermöglichen eine komplementäre Spezifikation von Systemen. Die Synchronisation von CSP und B konnte durch eine Prolog-Implementierung der Interpreter beider Spezifikationssprachen und der Verwendung von Prolog-Unifikationen erreicht werden [LF08].

1.3 Motivation und Projektziel

Die Analyse von sicherheitskritischen Systemen hinsichtlich ihrer sequentiellen Aktivität ist in Zeiten zunehmender Softwarekomplexität und der von digitalen Systemen ausgehenden Verantwortung von großer Bedeutung. Das Hauptziel der Arbeit war die Entwicklung eines neuen CSP_M-Parsers in einer Java-Umgebung für die Nutzung in ProB. Für die Entwicklung des bisher integrierten Parsers *cspmf*, der im Rahmen seiner Dissertation von Marc Fontaine im Jahr 2011 vorgestellt wurde, kam die funktionale Programmiersprache Haskell zum Einsatz [Fo11]. Das Front-End jedes anderen ProB-Werkzeugs ist bisher in Java implementiert. Die Parser der restlichen Spezifikationssprachen wurden mit Hilfe von SableCC erzeugt. Der Popularitätsgrad von Haskell ist im Jahr 2016 gegenüber Java als sehr gering einzuschätzen. Dem aktuellen TIOBE Programming Community Index zufolge ist Haskell mit 0,0304% gegenüber Java etwa 65-mal mal weniger populär.[1] Es muss damit gerechnet werden, dass die Identifikation von Fehlern und die Weiterentwicklung von *cspmf* nur eingeschränkt möglich ist. Aus diesem Grund sollte ein neuer Parser in einer weit verbreiteten Programmiersprache entwickelt werden. Die Wahl fiel auf Java, um das ProB Front-End zu vereinheitlichen. Eine weitere Motivation bestand darin, den Funktionsumfang der in ProB überprüfbaren Spezifikationen zu erweitern und an den Entwicklungsstand von *libcspm* anzugleichen. *libcspm* ist der Parser des Analysetools FDR, das Ursprünglich 1991 von dem Unternehmen *Formal Systems (Europe) Ltd* [Gi16c]. veröffentlicht und anschließend von der Universität Oxford weiterentwickelt wurde. Es diente während der Entwicklung von *cspmf* als Leitfaden für die Übersetzung der formalen Prozessalgebra CSP [Ho78] in den maschinell lesbaren Dialekt CSP_M [Ro05]. Sowohl in Anlehnung an die CSP_M-Dokumentation von FDR als auch die Übersicht der von ProB unterstützten Spezifikationen wurde *cspmj* als Zusammenfassung der wichtigsten Komponenten beider Parser entworfen. Da die Bearbeitungszeit von drei Monaten verglichen mit der einer Dissertation nur sehr kurz ist, muss die Entwicklungsherausforderung als besonders groß eingeschätzt werden. Auch die vielen komplizierten Eigenschaften der Sprache CSP_M, die in Kapitel 5.2 genauer erklärt werden, erschweren die Vervollständigung des Parsers. Es ist damit zu rechnen, dass auch in Zukunft noch einige Arbeiten notwendig sein werden, um den Parser mit der aktuellen FDR Version vollständig kompatibel zu machen. Aus diesem Grund reduziert sich die Erwartung nur auf die wesentlichen Bestandteile der Sprache, die das FDR-Front-End unterstützt.

Das Projekt wurde mithilfe von Github versioniert wurde das Projekt mithilfe und kann jederzeit unter folgendem Link heruntergeladen werden:

`https://github.com/RobinBia/CSPMJ`

[1] http://www.tiobe.com/tiobe-index/

2. CSP_M

2.1 Definition

CSP_M ist ein maschinell interpretierbarer Dialekt der Prozessalgebra CSP (= Communicating Sequential Processes), der die Interaktion zwischen kommunizierenden Prozessen beschreibt. Er ermöglicht es, Prozesse zu definieren und ihr Verhalten präzise und mathematisch zu steuern [Ro05] [Fr].

2.2 Verwendung

Bereits in den 80er-Jahren wurde CSP in Mikroprozessorarchitekturen wie dem INMOS T9000 Transputer eingesetzt um unter Anderem dessen Pipeline-Befehle zu verifizieren [Ba95]. In der Industrie wird CSP heute, wie auch die B-Methode zur Modellierung sicherheitskritischer Systeme eingesetzt. So entwarfen das Bremen Institute for Safe Systems und Daimler-Benz Aerospace ein Störungsmanagementsystem und eine 23000 Zeilen Code umfassende Avionikschnittstelle für die internationale Raumstation (ISS). Durch die Analyse mit CSP konnten Fehler gefunden werden und Verklemmungen verhindert werden [Bu97] [BPS99]. Auch in der Softwareentwicklung wird CSP eingesetzt. So half die Analyse einer Smart-Card-Architektur der Firma *Altran Praxis* deren Schutz vor unerlaubten Zugriffen zu gewährleisten. Ein weiteres Anwendungsszenario ist die Überprüfung von kryptographischen Verfahren. Mit Hilfe von FDR konnte in dem Needham-Schroeder-Protokoll, einem Verfahren für sicheren Datenaustausch in dezentralen Netzwerken, eine Sicherheitslücke entdeckt und behoben werden [Lo96].

2.3 Aufbau eines CSP-Programms

Eine CSP_M-Datei besteht aus einer Liste von Instruktionen. Dies können Definitionen, Deklarationen (z.B. `datatype`, `channel`), Kurzbefehle zur Steuerung des Codes wie `include`, `transparent` und `print` sein, aber auch Behauptungen (`assert`) zur schnellen Überprüfung von Aussagen. In dem folgenden Abschnitt werden die wichtigsten CSP-Konstrukte und Operatoren vorgestellt, die für ein anschließendes Beispiel benötigt werden, das die Funktionsweise von CSP demonstrieren soll. Eine Vollständige Übersicht und Erklärung bietet die CSP_M-Dokumentation für FDR, die bei der Entwicklung des Parsers als Leitfaden diente.

2.4 Dateizusammenfassung

Um mehrere Dateien zu einer größeren zusammenzufassen, eignet sich der Aufruf von `include "dateipfad"`. Dies kann sehr nützlich sein, wenn man bestimmte Teile eines Programms testen möchte um diese möglicherweise später hinzufügen zu können oder um große Dateien aufzuteilen und eine gewisse Übersichtlichkeit herzustellen.

2.5 Transparente/Externe Funktionen

Transparente und externe Funktionen sind bereits in den Interpreter von ProB eingebaut und werden nach ihrem Aufruf nur als solche erkannt, wenn sie vorher in einer Spezifikation explizit angegeben werden: `transparent <Bezeichnerliste>` bzw. `external <Bezeichnerliste>`. Transparente Funktionen werden grundsätzlich als Prozess interpretiert. Externe Funktionen hingegen können zusätzlich im Kontext aller anderen Ausdrücke genutzt werden. Es ist allerdings anzumerken, dass sich diese aufgrund ihrer nicht semantikerhaltenden Eigenschaften

nur bedingt für den Einsatz in sicherheitskritischen Systemen eignen [Gi16a].

2.6 Definitionen

Ausdrücke

Ein Ausdruck kann in CSP entweder ein Prozessausdruck oder ein Nicht-Prozessausdruck sein. Erstere werden durch Prozessatome (z.B. STOP, SKIP, CHAOS) oder Auswahloperatoren generiert, die in *Kapitel 2.10* genauer erklärt werden.
Nicht-Prozessausdrücke werden hingegen durch arithmetische oder boolesche Operatoren generiert, welche auf Ausdrucksatome (z.B. Zahlen, Sequenzen, Mengen, Ereignisse) angewandt werden.

Symbol	Beispiel	Bedeutung/Typ
m,n	`1`	`Zahl (-2^31,...,2^31-1)`
s	`<>`	`Sequenz (Folge von Expressions)`
a	`{}`	`Menge (Sammlung von Expressions)`
b	`true`	`Boolescher Wert`
P,Q	`P = Q [] P`	`Prozess (Definition mit Prozessoperator)`
p	`_@@'c'`	`Modell`
e	`c?x`	`Ereignis`
c	`channel c`	`Kanal`
expr	`s.o.`	`Expression mit variablem Typ`

Abbildung 1 CSP$_M$ Expressions

Diese Abbildung beschreibt, welche Symbole für atomare Ausdrücke in den folgenden Kapiteln verwendet werden und in welchem Kontext sie verwendet werden.

Modelle

Eine einfache Definition ist die Bindung einer Expression `expr` an ein Modell `p`, sofern `expr` und `p` gleiche Typen besitzen. Der Grundsätzliche Aufbau einer Definition lautet: `p = expr`. Modelle setzen sich gegebenenfalls dabei aus weiteren Modellen zusammen. Die folgende Auswahl enthält alle unterstützten Modelle und Modelloperatoren in der Reihenfolge ihrer Bindungsstärke von schwach (oben) nach stark (unten):

Doppel-Modell:	p_1`@@`p_2
Punkt-Modell:	p_1`.`p_2
Anhang-Modell:	p_1`^`p_2
Zahlen-Modell:	2^{31}`,…,`2^{31}`-1`
Variablen-Modell:	`Bezeichner*`
Literal-Modell:	`0.., true/false, 'c', "String"`
Tupel-Modell:	`(`p_1`,…,`p_n`)`
Parenthese-Modell:	`(p)`
Sequenz-Modell:	`<`p_1`,…,`p_n`>`
Mengen-Modell:	`{p}`
Platzhalter-Modell:	`_`

*Ein Bezeichner setzt sich wie folgt zusammen: Ein Buchstabe gefolgt von beliebig vielen alphanumerischen Zeichen und Unterstrichen gefolgt von beliebig vielen Prime-Zeichen, z.B. `Muster1_Bezeichner2'''`.

Funktionen

Eine weitere Definitionsart ist die Funktion: `F(p)…(p) = e.` Hierbei ist `F` ein Bezeichner, gefolgt von beliebig vielen Tupel- oder Parenthese-Modelle. Ein Beispiel ist: `F(x)(y) = x+y.` Dabei werden die Variable-Patterns innerhalb der runden Klammern als Parameter der Funktion aufgefasst und die Expressions x und y an die Variablen-Modell gebunden.

2.7 Kanäle

Ein CSP-Prozess wird ausschließlich durch die Kommunikation mit seiner Umgebung beschrieben. Um diese zu abstrahieren, werden Ereignisse definiert, welche prozessintern sequentiell und atomar ausgeführt werden. Je mehr Ereignisse definiert werden, desto höher der Abstraktionsgrad. Ein Ereignis wird genau dann kommuniziert, wenn sich alle beteiligten Prozesse darauf einigen. Zur Deklaration und Zusammenfassung von Ereignissen und deren Typ verwendet CSP_M Kanäle. Ein Kanal wird wie folgt definiert: `channel c: te`. Wobei `c` der Name des Kanals und `te` ein Typenausdruck, der die Gültigkeit des Ereignisses beschreibt. Die Kommunikation über ein Ereignis findet genau dann statt, wenn sein Aufruf abgeschlossen ist, d.h. `channel c:Int.Bool` (Int = $\{-2^{31}, \ldots, 2^{31}-1\}$) beschreibt eine Menge von gültigen Ereignissen. In diesem Fall wäre die Menge `{c.1.true, c.1.false, c.2.true, c.2.false,...}` eine Zusammenfassung aller Eingaben, die unter `c` als Ereignis erkannt werden können.

2.8 Datentypen

CSP_M erlaubt die Deklaration von eigenen Datentypen, um Bezeichner an größere Datenmengen zu binden und diese zu strukturieren.
Die allgemeine Definition lautet `datatype N = C1.te1 | C2.te2 | ... [Gi16c]`, wobei `N` der Name des zu definierenden Typs ist, C_i ein Datenkonstruktor und te_i ein Typenausdruck. Diese sind als Parameter aufzufassen und erweitern das Konstrukt `C` um gewisse Werte, die über den Dot-Operator zu erreichen sind. Ein Typenausdruck ist eine durch Punkte getrennte, beliebig lange Folge von n-Tupeln oder Mengen. Die folgende Deklaration definiert eine Menge von Elementen, denen der Typ Color zugewiesen wird: `datatype Color = Red.{1,2} | Green.{3,4} | Blue.{5,6}.` Diese Instruktion ist äquivalent mit der Definition
`Color = {Red.1, Red.2, Green.3, Green.4, Blue.5, Blue.6}.`

2.9 Prefixing

Der Präfixoperator `->` beschreibt eine endliche Abfolge von Ereignissen, auf welche die Ausführung eines Prozesses folgt. Die allgemeine Definition lautet `e -> P`, wobei `e` ein Ereignis und `P` ein Prozess ist. Der Ausdruck `e->P` ist wieder ein Prozess, der beliebig lange auf das Ereignis a wartet und sich anschließend wie P verhält. Der Prozess `STOP` ist eine eingebaute Konstante. Er repräsentiert die Verklemmung eines Prozesses, also eine Endlosschleife, welche die weitere Kommunikation von Ereignissen verhindert: `P =` e_1`->`e_2`->STOP.`
`P` wird definiert als der Prozess, der sich so verhält wie die rechte Seite. Nach der Kommunikation von `e1` folgt das Ereignis `e2`. Anschließend wird eine Endlosschleife betreten und es finden keinerlei Ereigniskommunikationen mehr statt. Eine andere Möglichkeit der Verhinderung von endlichen Prozessen ist die Rekursion. Dabei definiert der Prozess `P =` e_1`->`e_2`->P` eine unendliche Ereignisabfolge e_1`->`e_2`->`e_1`->`e_2`->...` .

2.10 Auswahloperatoren

Die bisher vorgestellten Methoden erlauben es nur, sequentielle Folgen von Ereignissen zu definieren, ohne dabei Einfluss auf den Verlauf ihrer Abarbeitung in Abhängigkeit von Zeit und Umgebungszustand zu haben. Dies ist vor allem hinsichtlich der mangelnden Flexibilität der Ereignissteuerung und der verfügbaren Ressourcen nicht praktikabel. Eine Möglichkeit die Priorität bei der Kommunikation von Ereignissen festzulegen, bieten die Auswahloperatoren.

Eingabe/Ausgabe

Besteht die Absicht, eine Sammlung von Prozessen zu definieren, deren Ereignisfolge sich nur atomar unterscheidet, so ist eine kürzere Schreibweise möglich: `c?x : A → P(x)`.
Diese Instruktion beschreibt mehrere Verhaltensweisen von `P(x)`, in dem als Ereignis alle Elemente `x` aus der Teilmenge `A` (oder vom Typ A) des Kanals `c` extrahiert werden und dem Prozess als Eingabe dienen. Dazu wird x im restlichen Verlauf an die Ereignismenge aus `c` gebunden. Wird `:A` weggelassen, so sind alle Ereigniskommunikationen aus `c` möglich. Gleichzeitig kann die Ausgabe einer Kanaleingabe nicht nur über Prozessargumente erfolgen, sondern auch über den jeweiligen Kanal mit dem output-Befehl. Ein geeignetes Beispiel hierzu ist das Folgende: `P = pressButton?x → openDoor!x → P` [Fo11]. Es beschreibt einen Prozess, der immer dann eine Tür öffnet, wenn ein Knopf gedrückt wird. Dazu empfängt P die Eingaben aus dem Kanal `pressButton` und gibt diese anschließend wieder über den Kanal `openDoor` aus.

External Choice

Der Nachteil, dass die Auswahl eines Prozesses immer von einem bestimmten Anfangsereignis abhängt, wird durch den External Choice Operator verhindert: `P [] Q`. Unterscheiden sich die Anfangsereignisse aus `P` und `Q`, so kann die äußere Umgebung, z.B. der aufrufende Prozess über die Auswahl eines Ereignisses festlegen, welcher der Prozesse ausgeführt wird. Sind die Anfangsereignisse gleich, so ist die Auswahl nichtdeterministisch und kann nur intern erfolgen. Hierzu verwendet man den Internal Choice-Operator.

Internal Choice

Sind die Initialereignisse beider Prozesse gleich, so tritt ein Spezialfall ein. Auf diese Weise kann der aufrufende Prozess keine eindeutige Wahl zwischen den Prozessen treffen, da die Ereigniskette unbekannt ist. Soll in diesem Fall eine Entscheidung getroffen werden, so ist dies nur während der Ereigniskommunikation der einzelnen Prozesse möglich. Man führt daher einen neuen Operator ein, die *Internal Choice*: `P |~| Q`.

Guard

Ein Prozess `Q` verhält sich genau dann wie `P`, wenn der boolesche Ausdruck `b` wahr ist. Ist `b` unwahr, so verklemmt `Q` und verhält sich somit wie `STOP`: `Q = b & P`.

2.11 Nebenläufigkeit

Um Prozesse möglichst gleichzeitig und unabhängig voneinander ausführen zu können bietet CSPM eine Reihe von Parallelisierungsoperatoren.

Alphabetised Parallel

Während der parallelen Ausführung beider Prozesse `P` und `Q` kann die Kommunikation eines Ereignisses der Menge `{e1}` kann nur durch `P` , eines der Menge `{e2}` nur durch `Q` erfolgen: `P [{e1}||{e2}] Q`.

Replicated Alphabetised Parallel

Werte das Alphabet `A(x)` und den Prozess `P(x)` ∀x∈<statements> aus und führe für jeden resultierenden Prozess *alphabetised Parallel* aus. Die Ereignismenge für den jeweiligen Prozess `P(x)` ist dem Alphabet `A(x)` zu entnehmen:
`|| <set statements> @ [A(x)] P(x)` (Für Statements, siehe auch *Kapitel 6.7*).

Interleave

Die Ausführung beider Prozesse `P` und `Q` ist unabhängig voneinander. Teilen `P` und `Q` Ereignisse, so kann nur genau einer der beiden Prozesse ein Ereignis kommunizieren: `P ||| Q`.

Synchronising External Choice

Dieser Operator ist eine Neuheit (seit FDR 2.94) und wird bisher nicht von ProB unterstützt. Er kann als Hybrid zwischen External Choice und Generalized Parallel

interpretiert werden: `P[+{e}+]Q`. Hier werden `P` und `Q` durch Ereignismenge `{e}` synchronisiert. Sobald ein Ereignis `e` in einer der beiden Prozesse nicht kommuniziert wird, verhält sich der Operator wie External Choice. Ansonsten entsteht eine Verklemmung.

2.12 Behauptungen

In der Praxis ist vor allem die Sicherheit komplexerer Systeme von großem Interesse. Um Maschinen zu testen und ihren sicheren Abbruch gewährleisten zu können, setzt man an bestimmten Stellen eines CSP-Programms Behauptungen ein, die zuerst überprüft werden müssen. Im Gegensatz zu Ausnahmen setzen Behauptungen keinen bestimmten Wert voraus, sondern ermitteln auf mathematischem Weg, ob das System noch lauffähig ist oder nicht.

Verklemmungsbehauptung

Um herauszufinden, ob ein Prozess verklemmungsfrei ist, kann einfach
der Befehl `assert P :[deadlock free] v`erwendet werden.

LTL/CTL-Behauptung

Ein besonderes Merkmal von *cspmj* ist die Fähigkeit zur Überprüfung von temporallogischen Formeln. Die Temporallogik ist eine Erweiterung Aussagenlogik, bei der es nicht um die Beschreibung von zeitlichen Abläufen geht, sondern um die Eigenschaften von Zuständen und ihr Verhalten in Abhängigkeit von bestimmten Ereignissen. So wird die Instruktion
`assert P |- LTL: "LTL-Formel"` nur dann geparst, wenn die angegebene LTL-Formel korrekt ist. Anschließend wertet *ProB* aus, ob die Formel auf den Prozess `P` erfüllt ist oder nicht.

2.13 Beispiel

Ein klassisches Beispiel für ein sicherheitskritisches System ist das Philosophenproblem.

Das Problem

Fünf Philosophen (N=5) teilen sich einen Esstisch und haben Sitzplätze, die ihnen fest zugewiesen sind. Auf dem Tisch liegen fünf Gabeln, die jeweils nur von einem Philosophen gehalten werden können. Möchte ein Philosoph essen, kann er dies nur, indem er sich zuerst hinsetzt und zwei Gabeln gleichzeitig benutzt. Kann ein Philosoph keine zweite Gabel nehmen, so muss dieser warten, bis ein anderer Philosoph mit dem Essen fertig ist.

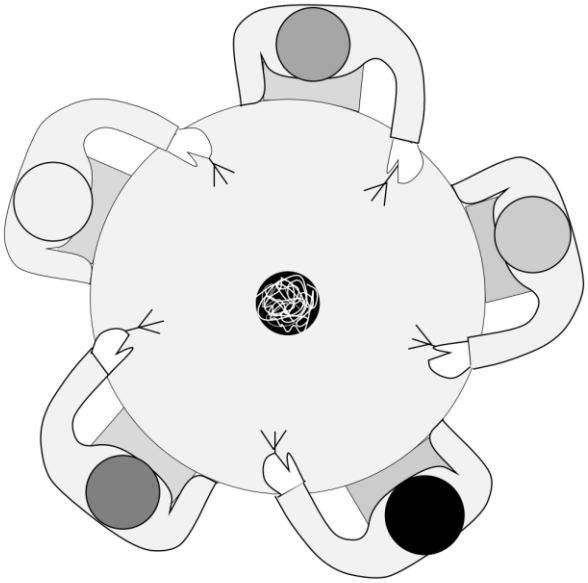


Abbildung 2 Philosophen beim Essen

Die Modellierung

Zu Beginn muss gewährleistet werden, dass keine Gabel von zwei Philosophen gehalten werden kann. Dazu werden fünf Prozesse `FORK(i)`, i∈[0;N[erstellt, die jeweils eine Gabel repräsentieren und fünf Prozesse `PHIL(i)` ,i∈[0;N[, die jeweils einen Philosophen darstellen. Die Anzahl der Gabeln und Philosophen wird jeweils in einer dafür vorgesehenen Menge gespeichert (PHILNAMES und FORKNAMES). Die Ereignisse, die den Zustand einer Gabel beschreiben sind picksup.i.i und putsdown.i.i, wobei i.i die Information enthält, welcher Philosoph i welche Gabel i hält. Für das Aufnehmen der rechten Gabel, also der nachfolgenden Gabel i+1 muss überprüft werden, ob i+1>N-1. In dem Fall handelt es sich nicht um Gabel N sondern 0. Dies beschreiben die Ereignisse `picks!i!((i+1)%N)` und `putsdown!i!((i+1)%N)`. Neben den schon beschriebenen Ereignissen für Gabeln haben die Philosophen noch zusätzlich die Möglichkeit zu sitzen, essen und aufzustehen (sits.i.i, eats.i.i, getsup.i.i) . Da es sich bei den Philosophen um Menschen handelt, ist davon auszugehen, dass sie individuelle Entscheidungen treffen und ihre Handlungen in willkürlicher Reihenfolge erfolgen. Unter der Voraussetzung, dass jeder Philosoph die linke Gabel zuerst aufnimmt und zuletzt ablegt, ermöglicht der Aufruf von `PHIL(i)` einen Essvorgang. Die Alphabete `AlphaP(i)` und `AlphaF(i)` beinhalten alle Ereignisse, die für die jeweiligen Gabeln und Philosophen eintreten können. Die parallele Komposition von `PHIL(i)` und `FORK(i)` ermöglicht die parallele Ausführung beider Prozesse unter Voraussetzung i und dass Philosophen und Gabeln nur Ereignisse kommunizieren können, die in Ihrem Alphabet definiert sind. So kann ausgeschlossen werden, dass eine von dem Philosophen weiter entfernte Gabel verwendet wird [Ro05].

```
N = 5
PHILNAMES= {0..N-1}
FORKNAMES = {0..N-1}
channel sits, eats, getsup:PHILNAMES
channel picks, putsdown:PHILNAMES.FORKNAMES
PHIL(i) = sits!i -> picks!i!i -> picks!i!((i+1)%N)
-> eats!i -> putsdown!i!((i+1)%N) ->
                                    putsdown!i!i ->
getsup!i -> PHIL(i)
AlphaP(i) = {sits.i,picks.i.i,picks.i.(i+1)%N,
       eats.i,putsdown.i.i,putsdown.i.(i+1)%N,getsup.i}
FORK(i) = picks!i!i -> putsdown!i!i -> FORK(i)
 []  picks!((i-1)%N)!i -> putsdown!((i-1)%N)!i -> FORK(i)
AlphaF(i) = {picks.i.i, picks.(i-1)%N.i,
putsdown.i.i, putsdown.(i-1)%N.i}
SYSTEM = || i:PHILNAMES@[union(AlphaP(i),AlphaF(i))]
               (PHIL(i)[AlphaP(i)|| AlphaF(i)] FORK(i))
```

Abbildung 3 Philosophenproblem in CSP$_M$-Code

3. SableCC

3.1 Definition

SableCC ist ein *LALR(1)-Parsergenerator*, der 1998 im Rahmen seiner Master-Arbeit von Étienne Gagnon vorgestellt wurde [Ga98].

3.2 Funktion

Unter Angabe einer Grammatik und lexikalischer Einheiten generiert *SableCC* (HHU-Version 3.2.9) spezifische Java-Klassen, welche den zur Grammatik gehörenden abstrakten Syntaxbaum (AST) äquivalent in höherer Programmiersprache darstellen. Auf diese Weise kann die Entwicklungszeit stark reduziert werden. Unterschieden werden drei Kategorien von Funktionen:

- ***Lexer***
  Zerlegung des Input-Streams in einzelne Wörter und Einordnung in logisch zusammenhängende Einheiten (Token). Für jedes Token wird eine Knoten-Klasse beginnend mit *T* angelegt.
- ***Parser***
  Generierung eines AST. Dabei wird für jede Regel und Alternative der Grammatik eine Knoten-Klasse beginnend mit *A* generiert
- ***Analyse***
  Klassen, die das *Visitor-Pattern* implementieren (AST-Visitor – *DepthFirstAdapter.java*) Auf diese Weise kann der AST sowohl vorwärts als auch rückwärts durchlaufen und zur Laufzeit analysiert werden.

3.3 SableCC-Dateien

Um die oben genannten Funktionen in höherer Programmiersprache zu erstellen, wird eine Datei benötigt, die alle notwendigen Informationen enthält. Diese wird in 7 Abschnitte unterteileilt:

- ***Package (optional)***
  Name des Projekts, bzw. Ort des Hauptverzeichnisses für oben genannte Klassen.
- ***Helpers***
  Bestimmung von Zeichenmengen, die zur Definition von Tokens hilfreich sind. Zum Beispiel beschreibt `['A'..'Z']` die Menge aller großen Buchstaben des Alphabets.
- ***States (optional)***
  Soll ein Token nur an einer bestimmten Stelle (z.B. in einer Teilgrammatik) als solches erkannt werden, dann kann es einem bestimmten State zugeordnet werden. In Kombination mit der Filtermethode des Lexers kann der erzeugte Tokenstream zur Laufzeit effizient manipuliert werden.
- ***Tokens***
  Angabe aller atomaren Einheiten, die bei der lexikalischen Analyse als solche erkannt werden und in eine Tokenliste eingereiht werden. Anschließend wird diese Liste an den Parser überreicht.
- ***Ignored Tokens (optional)***
  Angabe aller Terminalsymbole, die im Tokenstream übersprungen werden sollen. In den meisten Fällen sind dies Kommentare und Whitespace-Tokens wie Leerzeichen und Tabulatoren. In diesem Projekt kommt der Abschnitt nicht zum Einsatz, da die Funktionalität nicht differenziert genug für die Spezifikation von CSP_M ist.

- ***Productions***
  Definition des Concrete Syntax Tree durch Angabe einer Grammatik in EBNF-Ähnlicher Notation.
- ***Abstract Syntax Tree***
  Angabe aller AST-Transformationen zur Kompaktifizierung des CST. Alternativregeln, die nur die Aufgabe haben auf eine neue Präzedenzstufe zu zeigen, können so weggelassen werden. Näheres hierzu liefert *Kapitel 5.2*

3.4 Ein einfaches SableCC Beispiel

Ein einfaches Beispiel für die Angabe einer *SableCC*-Datei ist eine Sprache als modifizierte Untermenge von CSP_M:

- ***Terminale***
  Integerzahlen, Bezeichner beginnend mit einem Buchstaben und endend mit beliebig vielen alphanumerischen Zeichen, Präfixoperator, Parenthese, Operatoren für die Arithmetik (Addition, Subtraktion, Multiplikation, Division, Modulo).
- ***Regelwerk***
  Alle Bezeichner sind immer Ereignisse und Prozesse zugleich. Eine Zahl ist ein Ereignis, aber kein Prozess und kann somit immer nur links vom Präfix stehen.
- ***Präzedenz***
  Bindungsstärke von links (stark) nach rechts (schwach):
  Atome -> Parenthese -> Punktoperatoren -> Strichoperatoren -> Präfix

Die folgende *SableCC*-Datei soll die oben spezifizierte Sprache beschreiben:

```
Package Beispiel;
Helpers
    digits = ['0' .. '9'];
    first_digit = [digits - '0'];
    letter = (['A' .. 'Z'] | ['a' .. 'z']);
    alphanum = (digits|letter);
    whitespace = (' ' | 13 | 10)+;
Tokens
    identifier = letter alphanum*;
    number     = first_digit digits*;
    prefix     = '->';
    addsub     = '+'|'-';
    muldivmod  = '*'|'/'|'%';
    par_l      = '(';
    par_r      = ')';
    white      = whitespace;
Ignored Tokens
    white;
Productions
exp {->exp}  =  {prefix} exp prefix exp2        {->New exp.prefix(exp.exp,exp2.exp)}
              |{e2} exp2                        {->exp2.exp};
exp2 {->exp} =  {addsub} exp2 addsub exp3       {->New exp.addsub(exp2.exp,exp3.exp)}
              |{e3} exp3                        {->exp3.exp};
exp3 {->exp} =  {muldivmod} exp3 muldivmod atom {->New exp.muldivmod(exp3.exp,atom.exp)}
              |{atom} atom                      {->atom.exp};
atom {->exp} = {id} identifier                  {->New exp.identifier(identifier)}
               |{num} number                    {->New exp.number(number)}
               |{par} par_l exp par_r           {->New exp.parenthesis(exp.exp)};
Abstract Syntax Tree
exp   = {prefix} [exp]:exp [exp2]:exp
       |{addsub} [exp2]:exp [exp3]:exp
       |{muldivmod} [exp3]:exp [atom]:exp
       |{identifier} identifier
       |{number} number
       |{parenthesis} [exp]:exp;
```

Abbildung 4 Ein einfaches SableCC Beispiel

Offensichtlich erfüllt die Grammatik aus *Abbildung 4* die angegebene Spezifikation nicht vollständig, denn diese Lösung ermöglicht die Gültigkeit von Zahlen auf der rechten Seite des Präfixoperators.

Da eine Verkomplizierung der Grammatik nicht zielführend ist und mehr Regeln und Knoten erzeugt, kommt hier das *Visitor-Pattern* zum Einsatz. Um Atome mit gewissen Typen zu abzufangen (Typechecking), wird eine Klasse angelegt, die den Referenz-AST-Visitor *DepthFirstAdapter.java* überschreibt. Dabei können alle Methoden gelöscht werden, die für eine Typenanalyse irrelevant sind, z.B. alle In- und -Out-Methoden. Anschließend wird eine `Hashmap` vom Typ `<Node, String>` erstellt. Dabei referenziert der Schlüssel vom Typ `Node` den zuletzt betretenen Knoten. Der Wert vom Typ `String` speichert den nach der Auswertung dieses Knotens zurückgegebenen Typ. Wird nun der Knoten eines Atoms auf der rechten Seite des Präfixoperators aufgerufen, so kann über die Inhaltsanalyse der Hashmap an der kritischen Stelle herausgefunden werden, ob die Eingabe Typenkorrekt ist. Gegebenenfalls muss ein Fehler ausgegeben werden. Eine weitere Anwendungsmöglichkeit des Visitor-Patterns wird in *Kapitel 5.7* näher erläutert.

4. Kontinuierliche Integration

Um die Qualität von Software zu gewährleisten, kommt das Prinzip der kontinuierlichen Integration zum Einsatz. Zwei unverzichtbare Werkzeuge, um das auch permanente Integration genannte Prinzip von Java-Projekten zu ermöglichen, sind Gradle und jUnit. In diesem Projekt dient Gradle der automatisierten Kompilierung von Java-Dateien und jUnit dem Testen des Ausgabequellcodes von *cspmj* bzw. Eingabecode für den Prolog-Interpreter von ProB gegenüber der Ausgabe von dem bestehenden CSP_M-Parser *cspmf*. Beide Tools sind in Github versioniert und Herunterladbar.[2] [3]

4.1 Gradle

Definition

Gradle ist ein auf Java basierendes Build-Management-Automatisierungs-Tool, das der automatisierten Erzeugung von ausführbaren Java-Programmen und deren Verwaltung dient.

Motivation und Aufbau eines Gradle-Builds

Die Motivation Gradle zu nutzen, besteht darin, den Build-Prozess größerer Projekte zu beschleunigen und unabhängig von der ausführenden Plattform zu machen. Eine Build-Definition ist eine Abfolge von Tasks und Abhängigkeiten, die als direkt ausführbarer Code in der Datei `build.gradle` festgehalten werden. Während der Buildverarbeitung werden immer jeweils die Konfigurationsphase und Ausführungsphase durchlaufen. Im Konfigurations-Zyklus wird die gesamte Build-Definition durchlaufen und ein Abhängigkeitsgraph erstellt, der die Reihenfolge der zu bearbeitenden Schritte festhält. Anschließend werden die Tasks der vorkonfigurierten Reihenfolge nach abgearbeitet [HA15]. Der Gradle-Build-Prozess des *cspmj*-Projekts wird mit dem Befehl `gradle build` über die Kommandozeile angestoßen und hat folgenden Aufbau:

- Herunterladen von *SableCC* (Version 3.2.10) und *jUnit* (Version 4.+)
- Herunterladen von plattformabhängiger *cspmf*-Version nach `build/classes/main`
- Generierung der Java-Klassen des CSP_M-Parsers mittels SableCC
- Generierung der Java-Klassen des LTL- und CTL-Parsers mittels SableCC
- Kompilieren der Java-Dateien im Ordner `src/main/java` bzw. erzeugen aller `.class`- Dateien (`javac`-Befehl)
- Kopieren von Produktionsressourcen (.scc-Dateien) in den Ordner `build/resources`
- Generierung der ausführbaren *cspmj.jar*
- Kompilierung der Java-Dateien im Ordner `src/test/java` (`javac`-Befehl)
- Ausführen der jUnit-Tests
- Ausführen sämtlicher Verifikationsaufgaben

Schlägt ein jUnit-Test fehl, erhält man ein Feedback über die Anzahl der gescheiterten Tests und ihren Namen. Der Build-Prozess schlägt automatisch fehl und verursacht eine Fehlerausgabe `FAILURE: Build failed with an exception.`

[2] https://github.com/junit-team/junit4
[3] https://github.com/gradle/gradle

4.2 jUnit

Definition

jUnit ist ein Open-Source-Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units (Klassen oder Methoden) geeignet ist [Mi06].

Motivation

Im Hinblick auf das Ziel der Arbeit, der Anbindung eines neuen Parsers an ProB, musste die Übersetzung von CSP_M nach Prolog möglichst nahe an das Verhalten des bereits bestehenden Haskell-Parsers angeglichen werden. Aus diesem Grund wurde mit einem Reverse-Engineering-Verfahren der AST-Visitor zum Generieren von Prolog-Termen mit *cspmf* synchronisiert.

Um nachweisen zu können, dass die Ausgabe beider Parser bei gleichen Eingabedateien äquivalent ist, werden in diesem Projekt jUnit-Tests verwendet. Diese ermöglichen den Vergleich von *cspmj*- und *cspmf*-Ausgabedateien. Vor dem Parsen des Testcodes muss der AST normalisiert werden, um die jeweiligen Ausgaben vergleichbar zu machen. Dabei werden Positionsangaben von Operatoren und Bezeichnern mit `no_loc_info_available` ersetzt und der Header mit Versionsinfos und dynamischen Prädikaten, sowie die Symbolliste und die Liste mit allen Kommentaren und Pragmas gelöscht. Danach erfolgt die Deaktivierung des Symbolrenamings. Auch in *cspmf* findet dieser Vorgang statt. Eigens dazu wurde ein neues Kommandozeilenargument implementiert. Statt des Endarguments `--prologOut=dateiname.csp.pl` muss hierbei `--prologOutNormalised=dateiname.csp.pl` verwendet werden.

Einen weiteren Zweck erfüllen jUnit-Tests hinsichtlich der Testung von bestimmten Abschnitten der CSP-Grammatik. Schlägt ein Test fehl und muss der Quellcode geändert werden, so gibt ein jUnit-Test Auskunft darüber, ob durch die vorherige Änderung eine ältere Funktion verlorengegangen ist. Auf diese Weise ermöglicht automatisiertes Testen eine erhebliche Zeitersparnis. Alle Tests werden zeitgleich mit jedem Aufruf von `gradle build` ausgeführt.

```
@Test
public void ExpressionOperators() throws Exception //
{
    check(
            "v = -(1^2)^2"
            +newline+"w = 0-1+2*3/4%-5"
            +newline+"x = #1"
            +newline+"z = true or false and not true"
          ,
            "'bindval'('v','negate'('^'('^'('int'(1),'int'(2)),'int'(2)))"
            +",'no_loc_info_available')."
            +newline+"'bindval'('w','+'('-'('int'(0),'int'(1)),'%'('/'('*'"
            +"('int'(2),'int'(3)),'int'(4)),'int'(-5))),'no_loc_info_available')."
            +newline+"'bindval'('x','#'('int'(1)),'no_loc_info_available')."
            +newline+"'bindval'('z','bool_or'('true','bool_and'('false',"
            +"'bool_not'('true'))),'no_loc_info_available')."
            +newline+"'symbol'('v','v','no_loc_info_available','Ident (Groundrep.)')."
            +newline+"'symbol'('w','w','no_loc_info_available','Ident (Groundrep.)')."
            +newline+"'symbol'('x','x','no_loc_info_available','Ident (Groundrep.)')."
            +newline+"'symbol'('z','z','no_loc_info_available','Ident (Groundrep.)')."
          );
}
```

Abbildung 5 Quellcode eines jUnit Tests

Zu sehen ist ein Test, der die Korrektheit der Prologterm-Generierung einer CSP-Datei mit sämtlichen arithmetischen Operationen prüft. Dazu wird die Methode `check(String, String)` aufgerufen. Der erste String repräsentiert dabei die Eingabedatei, der zweite String ist die erwartete Ausgabe. Entspricht die Ausgabe des Parsers nach Verarbeitung des ersten Strings dem zweiten String, so ist der jUnit-Test erfolgreich verlaufen.

Eine Übersicht über den Korrektheitsgrad aller Tests wird in einer html-Datei `index.html` angelegt. Diese befindet sich in:

```
CSPMJ\build\reports\tests
```

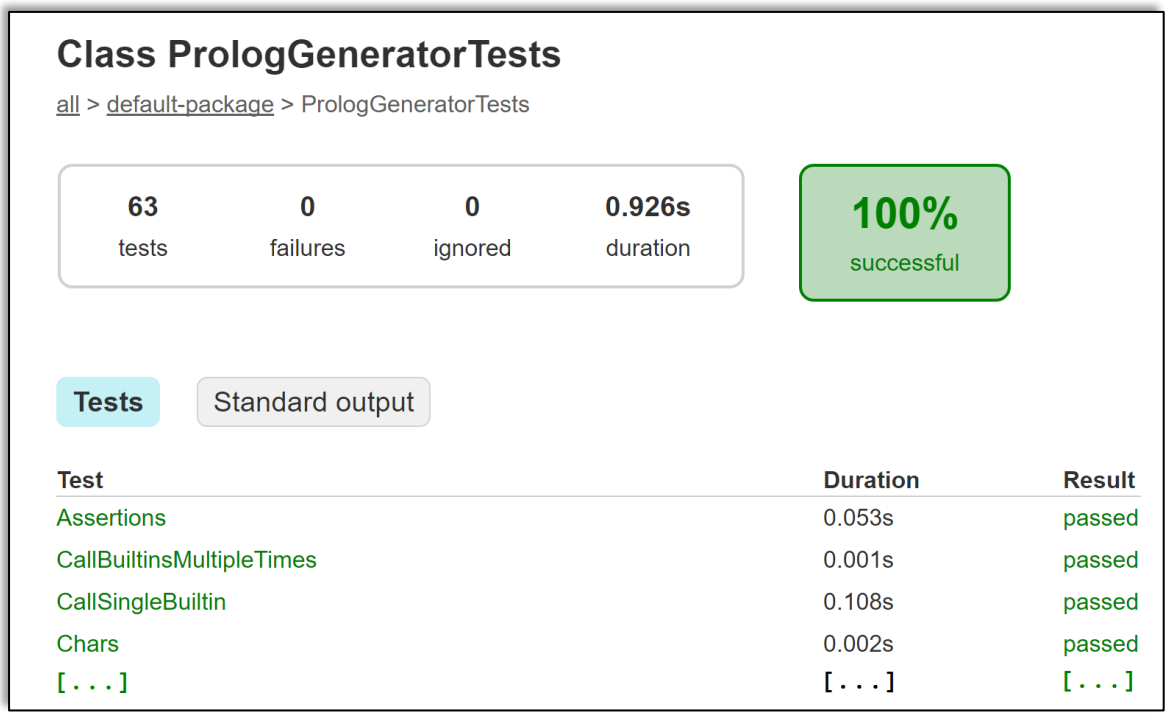


Abbildung 6 jUnit – Übersicht zum Verlauf der Tests

5. Der Parser

5.1 Vergleich mit CSPMF

Ein Ziel der Arbeit war die Anbindung eines neuen CSPM-Parser an ProB und somit die Angleichung des Verhaltens an den bestehenden ProB-Parser *cspmf*. Gleichzeitig sollten aber auch weitere Funktionen implementiert werden, die seit 2011 in der Spezifikation von FDR hinzugekommen und nun in der aktuellen Version FDR 3.4 ausgeschrieben sind. Aus diesem Grund stellt *cspmj* eine Kombination aus den Funktionen beider Parser da. Unterstützt werden alle Befehle, die *cspmf* bereits unterstützt waren und zusätzlich dazu einige neue Operatoren, eingebaute Funktionen und Konstanten bis FDR Version 3.0. Die folgende Übersicht erklärt alle Befehle, die neu implementiert wurden:

Name	Regel	Bedeutung
Map Leere Map	`(| k1 => v1, ..., kN => vN |)` `(| |), mit Leerzeichen in der Mitte!`	Weise Schlüsseln k_i Werte v_i zu
Char-Literal/Modell	`'c'`	Ein Unicode-Zeichen mit einfachen Anführungsstrichen
String-Literal/Modell	`"String"`	Eine Sequenz von Zeichen mit doppelten Anführungsstrichen
Has-Trace-Assertion (optional mit `[M]`, wobei `M = T|F|FD|R`)	`assert P :[has trace [M]]: s`	Überprüfe die Behauptung, dass eine Sequenz `s` in `P` vorkommt
Tau-Prio-Assertion	`assert P[M=Q:[taupriority{x}]`	Äquivalent zu cspmf-Ausdruck `assert P[M=Q:[tau priority]:{x}`
Non-deterministic input Non-det. restricted input	`$p` `$p:a`	Verhalten wie bei Input `?p`, Auswahl erfolgt durch Internal Choice statt External Choice
Exception	`P [|{e}|> Q`	Starte `P`. Wird ein Ereignis `e` durch `P` ausgeführt, so starte `Q`
Synchronising External Choice	`P [+{e}+] Q`	Synchronisiere `P` und `Q` durch Ereignismenge `{e}`. Gilt e∈P^e∉Qve∉P^e∈Q, so verhält sich der Operator wie `[]`
Synchronising Interrupt	`P /+{e}+\ Q`	Verhalten wie `/\` wenn e∈P^e∉Qve∉P^e∈Q.
Replicated Synchronising Parallel	`[+{e}+] <stmts> @ P(x)`	Werte P ∀x∈<stmts> aus und setze resultierende Prozesse mit `[+{e}+]` zusammen. Gilt P(x)∉<stmt>,∀x, so verhält sich der Prozess wie `STOP`

Abbildung 7 Neue Funktionen in cspmj

Ebenfalls geändert wurde das Verhalten beim Aufruf eingebauter Funktionen und Konstanten. Die Behandlung erfolgt stets auf die gleiche Weise. Den Aufruf eines eingebauten Wortes, das noch nicht zuvor aufgerufen wurde, interpretiert der Prolog-Erzeuger als Funktions- oder Variablendefinition und legt einen Eintrag in der Prolog-Symbolliste mit der Information `BuiltIn primitive` an. Alle Symbole dieser Art können überschrieben werden. Bei einer Neudefinition wird der Listeneintrag für das entsprechende Symbol ersetzt (siehe auch *Kapitel 7.8*).

Die folgende Liste enthält alle Wörter, deren Interpretation auf die oben genannte Weise erfolgt:

Konstanten	Set-Funktionen	Sequenz-Funktionen	Map-Funktionen	Fehler-behandlung	Prozesse	Dot-Funktionen
Bool	card	concat	emptyMap	error	CHAOS	extensions
Char	diff	elem	mapDelete	show	DIV	productions
Int	empty	head	mapFromList		RUN	
Proc	inter	length	mapLookup		SKIP	
Events	Inter	null	mapMember		STOP	
True	member	set	mapToList		WAIT	
False	seq	tail	mapUpdate			
	Seq		mapUpdateMultiple			
	set		Map			
	union					
	Union					

Abbildung 8 Neue Builtin-Funktionen

Eine genaue Beschreibung der Funktionalität ist der CSP_M-Dokumentation von FDR zu entnehmen.

5.2 Entwicklungsaufwand

Vergleichen mit anderen Parser-Generatoren wie Bison, ist der Entwicklungsaufwand eines Parsers mithilfe von SableCC besonders groß. Dies wird bereits anhand eines kleines Ausschnittes der Grammatik von *libcspm* sichtbar:

```
_proc : PAR gens AT LSQUARE set RSQUARE proc          %prec AT
            | NDET gens AT proc                       %prec AT
            | BOX gens AT proc                        %prec AT
            | INTL gens AT proc                       %prec AT
            | SEMI gens AT proc                       %prec AT
            | LCOMM set RCOMM gens AT proc            %prec AT
            | proc BACKSLASH set
            | proc INTL proc
            | proc LCOMM set RCOMM proc
            | proc LSQUARE set PAR set RSQUARE proc
            | proc NDET proc | proc BOX proc
            | proc TIMEOUT proc | proc INTR proc
            | proc SEMI proc | bool GUARD proc
            | dotted fields ARROW proc | dotted ARROW proc
            | STOP | SKIP| CHAOS OPEN set CLOSE; [Sc98]
```

Abbildung 9 Ein Ausschnitt aus der libcspm-Grammatik

Alle binären Operatoren tauchen in einer einzigen Regel als Alternative auf. Die Präzedenz muss dabei nicht explizit durch die Grammatik ausgedrückt werden, sondern wird durch ein Prozentzeichen angegeben. Grammatikanteile, die aufgrund der Definition an mehreren Stellen vorkommen müssen, können trotz Mehrdeutigkeit geparst werden. Der hier dargestellte Ausschnitt

des CST ist in SableCC viel Größer. Der Grundsätzliche Aufbau des Grammatik-Kerns (siehe *Abbildung 13*, ***dunkelblau***) sieht vor, für jede Präzedenzstufe eine neue Regel zu erzeugen anstatt einer Alternative. Pro Präzedenzstufe wird wiederum eine Alternative benötigt um zur nächsten Stufe überzugehen:

```
A = A BinOp B | B
B = B BinOp C | C
C = C UnOp | D
D = Atom
```

Dabei ist die Einhaltung der folgenden vier Regeln von größter Bedeutung:

1. Unäre und Binäre Operatoren dürfen nicht die gleiche Präzedenzstufe teilen.
2. Links- und Rechts-Unäre Operatoren dürfen nicht die gleiche Präzedenzstufe teilen.
3. Alle unären Operatoren müssen eine höhere Präzedenz haben als alle anderen binären Operatoren.
4. Links- und Rechts-Assoziative Binäroperatoren dürfen nicht die gleiche Präzedenzstufe teilen [Ga04].

Die Verletzung einer Regel führt entweder zur Mehrdeutigkeit der Grammatik oder zu scheinbaren Fehlern beim Parsen, die nur durch veränderte Klammersetzung zu lösen sind. Da die CSP_M-Sprachenspezifikation aber genau diese Regeln verletzt, ist es nur mit erheblichem Aufwand und States im AST-Visitor möglich, bestimmte Konstrukte eindeutig zuzuordnen. Beispielsweise werden Variablen-Modelle und Ausdrucks-Variablen an der gleichen Stelle in der Grammatik geparst. Ihre Bedeutung, ihr Typ und Interpretation als Prologvariable hängt aber davon ab in welchem Kontext sie und ihre Umgebung stehen. Eine Lösung für die Mehrdeutigkeit von Generatoren- und Prädikat-Statements in SableCC wird in *Kapitel 5.7* erläutert.

Eine weitere Herausforderung stellt die Interpretation von Zeilenumbrüchen dar. Diese können verschiedene Bedeutungen haben. Jede CSP-Instruktion wird von ihrer Nachbarinstruktion durch einen Zeilenumbruch getrennt. Jedoch ist es auch möglich an ganz bestimmten Stellen Umbrüche zu setzen, die der Übersichtlichkeit des Codes dienen und nicht als Instruktionsende erkannt werden dürfen. Dazu gehören unter Anderem beide Seiten binärer Operatoren und die Stellen hinter öffnenden und vor schließenden Klammern. Unter Voraussetzung sehr guter SableCC-Kenntnisse ist es möglich, in der Filtermethode des Lexers alle newline-Token zu dereferenzieren. Auf diese Weise werden sie nicht geparst. Tritt ein Token auf, das am Anfang, bzw. am Ende einer Instruktion stehen muss, wird ein erneuter Aufruf der filterWrap()-Methode erzwungen, um eine zuvor ignoriertes newline-Token wieder in den Tokenstream einzureihen. Ein Vorteil, den diese Methode mit sich bringt, ist die Komprimierung des Tokenstreams zur Laufzeit. Neben newlines werden alle informationslose Tokens wie Tabulatoren und Leerzeichen aus der Tokenliste entfernt und ermöglichen ein speichereffizientes Lexen. Dennoch ist anzumerken, dass sich gewisse Java-Datenstrukturen, die auch SableCC verwendet, nicht zum Parsen von textintensivem CSP_M-Code eignen. Grund für diese Annahme ist eine Testdatei mit 3 Megabyte Inhalt, die in der Parser-Klasse einen Speicherfehler beim Überlaufen einer Hashmap erzeugte.

CSP_M ist eine stark typisierte Sprache und die Implementierung eines Typecheckers ist ohne erheblichen Aufwand nicht möglich. Der Einsatz eines Typisierungsverfahrens wie der Typinferenz nach Hindley-Milner wird in Zukunft den Aufwand reduzieren können.

Ein weiterer Unterschied zu den meisten Programmiersprachen ist, dass die Lokalität einer Definition in einem bestimmten Sichtbarkeitsbereich unerheblich ist. Beispielsweise ist ein Java-Ausschnitt mit `int y = x; int x = 1;` ungültig, denn das `x` wird verwendet bevor es definiert wird. Aus diesem Grund ist zur Identifikation von Variablen die Implementierung eines speziellen Algorithmus notwendig. Eine Erkennung aller Symbole zur Laufzeit ist in CSP nicht möglich. Details hierzu liefert auch *Kapitel 5.11*.

5.3 Präzedenzunterschiede

Leider gibt es zwischen *cspmf* und *libcspm* Unterschiede hinsichtlich der Präzedenz von Operatoren. Eine Änderung der Präzedenzen würde einen großen Arbeitsaufwand im Backend von ProB bedeuten und gleichzeitig keine Vorteile bieten. Aus diesem Grund gleichen sich auch hier *cspmf* und *cspmj*. Jedoch ist anzumerken, dass auch die Einordnung von Operatoren erfolgen muss, die seit FDR 3.0 neu hinzugekommen sind. Aus diesem Grund wurde eine Präzedenztabelle erstellt, die erstmals auch Operatoren aus dem Bereich `synchronising` einordnet.

Die folgende Tabelle beschreibt die Bindungsstärke aller von *cspmj* unterstützten Operatoren von schwach (oben) nach stark (unten):

Stufe	Operator	Eigenschaft
1	Hide	links-assoziativ, binär
2	Interleave	links-assoziativ, binär
3	Exception, Alphabetised Parallel, Generalised Parallel, Linked Parallel	nicht-assoziativ, binär
4	Internal Choice	links-assoziativ, binär
5	External Choice, Synchronising External Choice	links-assoziativ, binär
6	Synchronising Interrupt,Interrupt	links-assoziativ, binär
7	Sliding Choice/Timeout	links-assoziativ, binär
8	Sequential Composition	links-assoziativ, binär
9	Guard,Prefix, Lambda, Let-Within, If-Then-Else, Replicated	rechts-assoziativ, linksunär nicht-assoziativ, linksunär nicht-assoziativ, linksunär
10	Nondeterministic Input, Input, Output (?,$,!)	links-assoziativ, linksunär
11	Restriction (:)	nicht-assoziativ
12	Dot (.)	rechts-assoziativ, binär
13	or	links-assoziativ, binär
14	and	links-assoziativ, binär
15	not	links-assoziativ, binär
16	Vergleich (>,<, >=,<=,!=, ==)	nicht-assoziativ, binär
17	Addition (+), Subtraktion (-)	links-assoziativ, binär
18	Multiplikation (*), Division (/), Modulo (%)	links-assoziativ, binär
19	Unäres Minus (-)/Negation	links-assoziativ, linksunär
20	Sequenzlänge (#)	links-assoziativ, rechtsunär
21	Concat/Append-Modell (^)	links-assoziativ, binär
22	Rename	nichts-assoziativ, rechtsunär
23	Parenthese,Tupel, Atome	nicht-assoziativ

Abbildung 10 Operatorenpräzedenztabelle

5.4 Architektur

Die folgende Übersicht zeigt den Weg einer CSP_M-Instruktion durch den Parser bis hin zur Erzeugung des zugehörigen Prolog-Terms. Der Ablauf ist in einer Methode `parsingRoutine` in der Hauptklasse *CSPMparser.java* zusammengefasst. Die folgenden Unterkapitel beschreiben den hier illustrierten Ablauf.

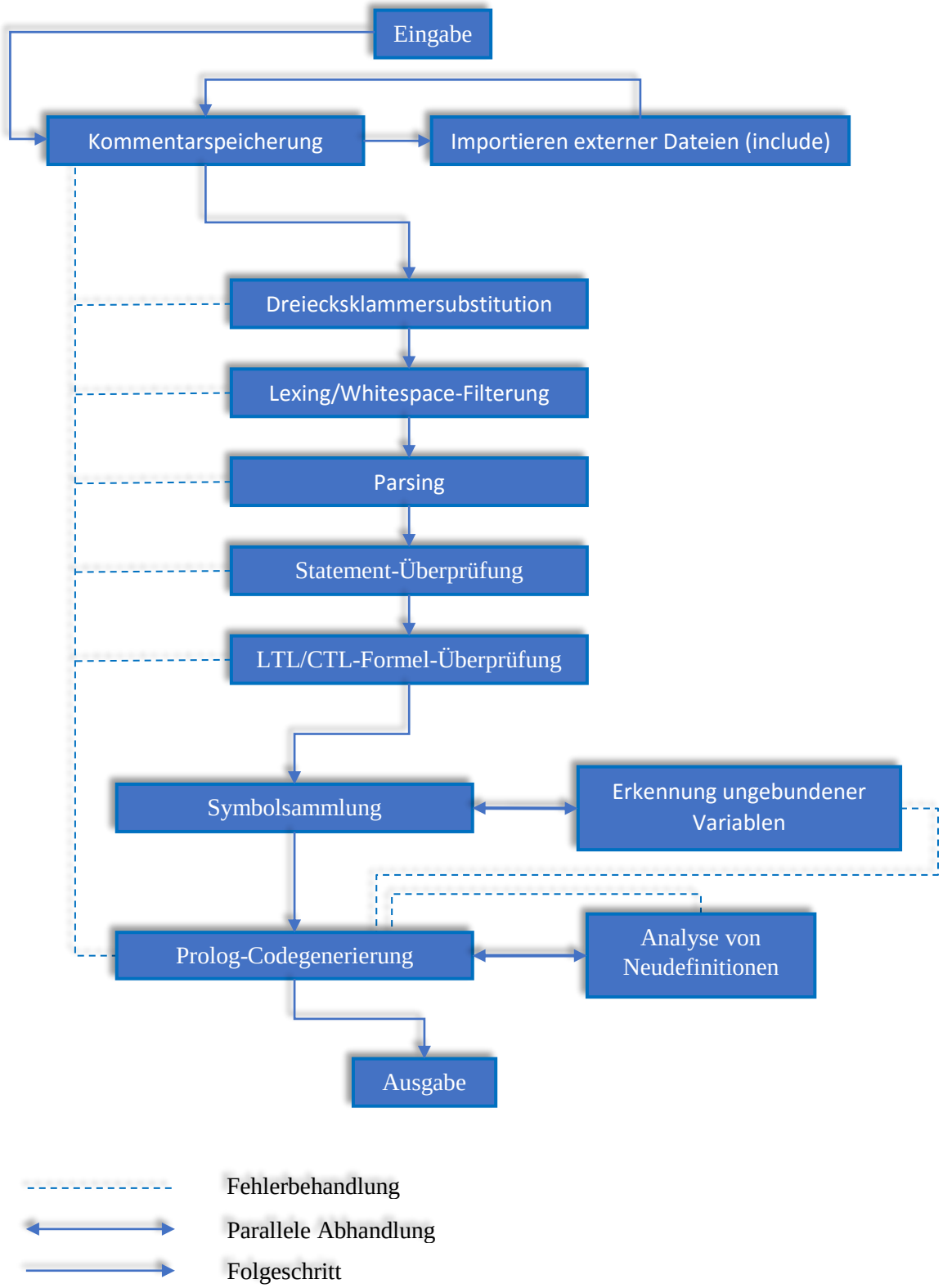


Abbildung 11 Architektur des Parsers

5.5 Prelexing

Die Erkennung von Tokens im SableCC-Lexer ist nur dann sinnvoll, wenn gewisse Regeln zugrunde liegen, die ihr Auftreten im Tokenstream regeln. Kommentare hingegen werden häufig in der Kategorie `Ignored Tokens` im Lexer aufgeführt, denn ihrer inneren Struktur liegen keine Regeln zugrunde. Ist es erforderlich, Änderungen an dem gepufferten Inhalt der CSP-Datei zur Laufzeit durchzuführen, so müssen Kommentare von der Änderungsroutine ausgeschlossen sein. Um den Inhalt von Kommentaren zu schützen, ist ihre manuelle Interpretation und Auslagerung sinnvoll. Dies geschieht in dem Parser unmittelbar nach Auslesen der Eingabedatei, um Kommentare, die von der Dreiecksklammersubstitution betroffen wären, zu schützen.

Kommentarpufferung

Die beiden Kommentararten Line-Comment und Multiline-Comment werden durch einen Algorithmus in der Hauptklasse *CSPMparser.java* durch die Methode `saveComments(String)` in einer `ArrayList<CommentInfo>` gespeichert. Eine Kommentarinformation ist dabei eine Datenstruktur, die als Information die Zeile- und Spalte des Anfangszeichens, die Nummer des Anfangszeichens, die Länge des Kommentars, den Kommentarinhalt inklusive der Zeichen für die Kommentarsetzung (`--`, `{-`, `-}`) und ob es sich um einen Multiline-Comment handelt oder nicht speichert. Die Methode `analyse(String)`, die im Konstruktor ausgeführt wird, identifiziert dabei Pragmas `{-# "Formel" "Kommentar" #-}` mithilfe von regulären Ausdrücken (*Java Regex*) und verleiht der Kommentarinformation weitere Attribute zur Speicherung des Inhalts der temporallogischen LTL/CTL-Formel, ob diese eine LTL-oder CTL-Formel ist und ob der Kommentar ein Pragma ist. Anschließend werden alle Zeichen (außer `\n` und `\r`), die mit Kommentaren assoziiert sind, durch Leerzeichen ersetzt. Auf diese Weise kann gewährleistet werden, dass im Falle eines Fehlers die Positionsangabe des fehlerverursachenden Tokens erhalten bleibt. Nach der Löschung aller Kommentare können externe Dateien eingebunden werden. Es ist erforderlich, alle Kommentare vorher zu löschen, damit include-Befehle innerhalb von Kommentaren nicht berücksichtigt werden. Wird eine CSP-Datei importiert, so muss der Algorithmus erneut ausgeführt werden, um auch Kommentare dieser Datei zu puffern.

Dreiecksklammersubstitution

Im Gegensatz zu libcspm unterstützt cspmf die Verwendung von Vergleichsoperatoren innerhalb von Sequenzausdrücken. Eine Sequenzklammer unterscheidet sich jedoch nicht von dem Vergleichszeichen. Das parsen von `<3>4>` ist zum Beispiel nicht möglich, da an der Stelle hinter der `3` noch nicht klar ist, ob die Sequenz geschlossen wird, oder mit `4` verglichen werden soll. Aus diesem Grund wurde eine Klasse zur Substitution von Zeichen implementiert. Diese ermöglicht das Umwandeln des obigen Ausdrucks in: `«3£4»`.

Dabei gibt es vier Regeln zum Ersetzen von Zeichen:

```
1. < ≙ öffnende Sequenz      ⇒    \u00AB ≙ «
2. > ≙ schließende Sequenz   ⇒    \u00BB ≙ »
3. > ≙ größer als            ⇒    \u00A3 ≙ £
4. < ≙ kleiner als           ⇒    \u20AC ≙ €
```

Alle anderen Tokens, die Dreiecksklammern enthalten, bleiben erhalten. Zu Beginn werden die Dateiränder untersucht, d.h. eine Dreiecksklammer am Anfang und Ende einer CSP-Datei kann nur zu einer Sequenz gehören. Um zu verhindern, dass während der restlichen Ersetzung eines der vier o.g. Zeichen unbeabsichtigterweise ein anderes Token verändert wird, werden zunächst folgende Zeichenketten durch Vor- und Rückwärtssubstitution geschützt:

<=>	⇔	\u00A2\u00A4\u00A2
<=	⇔	\u00A2\u00A4
=>	⇔	\u00A4\u00A2
<->	⇔	\u00A6\u00A5\u00A6
->	⇔	\u00A5\u00A6
<-	⇔	\u00A6\u00A5
[>	⇔	\u00A7\u00A8
<|	⇔	\u00B1\u00B2
|>	⇔	\u00B2\u00B1

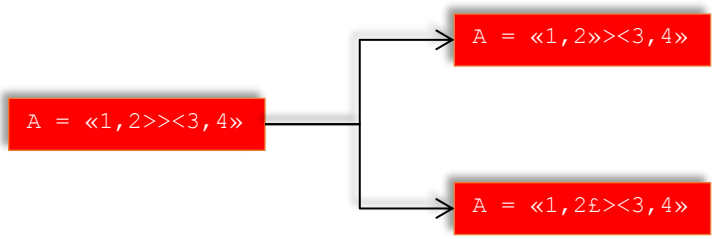
Nach der Vorwärtssubstitution erfolgt die Ersetzung der restlichen Klammern. Diese wird solange durchgeführt, bis keine Änderungen mehr erfolgt. Ein Beispiel für eine Regel, die eine Dreiecksklammer in eine schließende Sequenzklammer umwandelt, ist die Kommaregel. Befindet sich eine schließende Klammer vor einem Komma, so kann es sich nicht um einen Vergleichsoperator handeln. In diesem Fall ist die Zuordnung eindeutig und es kann substituiert werden. Die folgende Regel ist nur eine von 63 Regeln, die durch Erkennung eines bestimmten Musters die richtige Klammer einfügt:
`stream = stream.replaceAll(">"+white+"[,]","\u00BB$1,");` .Auf diese Weise können viele Klammern eindeutig zugewiesen werden. In seltenen Fällen kann jedoch keine Substitution erfolgen. Ein Beispiel hierfür ist eine CSP-Datei dem Inhalt

```
A = <1,2>><3,4>.
```

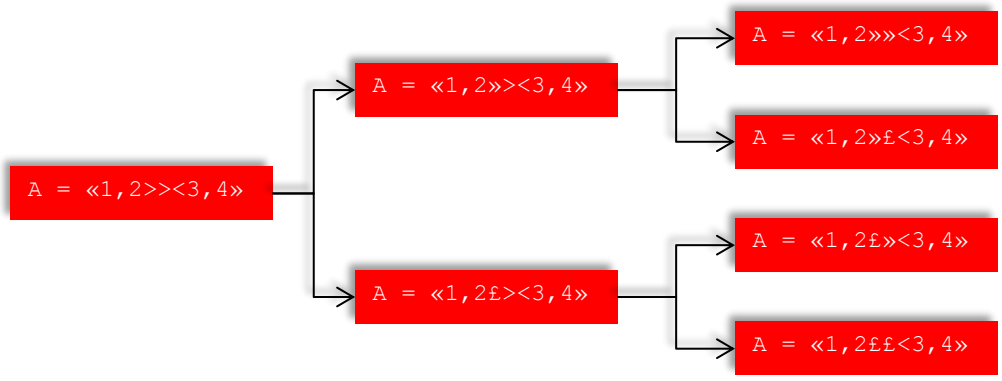
Nach Anwendung der oben genannten Regeln fehlen noch 3 Klammern:

```
A = «1,2>><3,4»
```

Aus diesem Grund wurde ein weiterer Algorithmus implementiert, der mithilfe eines Bruteforce-Schemas versucht, eine Kombination von Klammern zu finden, sodass der oben genannte Ausdruck parsbar wird. Dazu wird die erste gefundene Dreiecksklammer sowohl durch » als auch £ ersetzt. Anschließend wird versucht beide entstandenen Ausdrücke zu parsen.



Da dies in beiden Fällen nicht möglich ist, werden die nächsten beiden Klammern ersetzt. Dabei entstehen jeweils 2 weitere Alternativen.



Auch diesmal kann keine Instruktion erfolgreich geparst werden. Also wird die letzte unbekannte Klammer durch jeweils 2 Alternativen ersetzt. Es existieren nun 2^3 Alternativen:

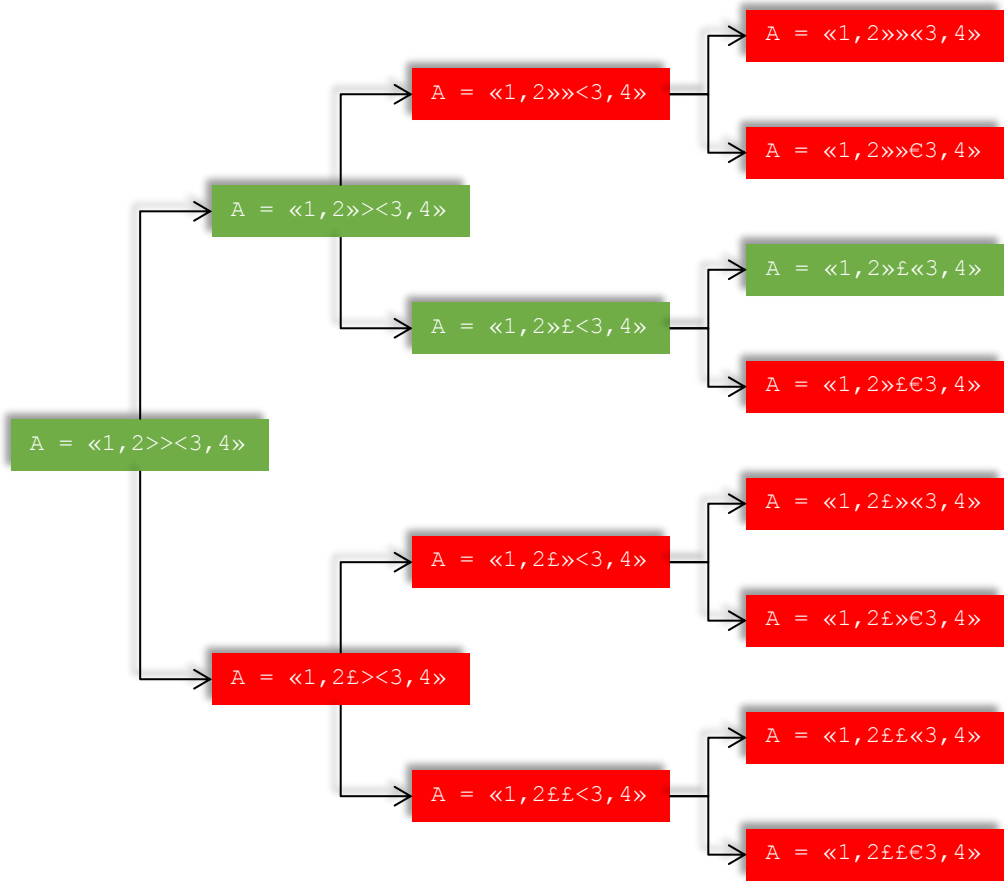


Abbildung 12 Erfolgreiche Ersetzung von Dreiecksklammern mittels Brute-Force

In diesem Durchgang kann eine Instruktion geparst werden. Nur der Vergleich zweier Sequenzen ist korrekt. Beispiele, bei denen mehrere Pfade zu einem richtigen Ergebnis führen, können nicht Typenkorrekt sein. Der Vergleich von zwei Ord-Zwangsbedingungen [Gi16b] erfüllenden Atomen führt grundsätzlich zur Rückgabe eines booleschen Wertes, der wiederum nicht erneut vergleichbar ist. Die Ausführung des Brute-Force-Algorithmus bezieht sich immer auf einzelne Zeilen. Liegt ein Umbruch vor, so wird eine weitere Zeile dem Betrachtungsbereich hinzugezogen. Die Implementierung ist offensichtlich nicht optimal, da im Falle eines Fehlers immer mehr Zeilen hinzu kommen müssten. Dies hätte wiederum zu Folge, dass die Anzahl der zu identifizierenden Klammern so groß würde, dass eine praxisrelevante Laufzeit nicht mehr gewährleistet werden könnte.

5.6 Diagramm zum Aufbau des CST

Das folgende Bild soll eine grobe Übersicht von dem Aufbau der cspmj-Grammatik vermitteln und dient als Erklärungshilfe der folgen Kapitel.

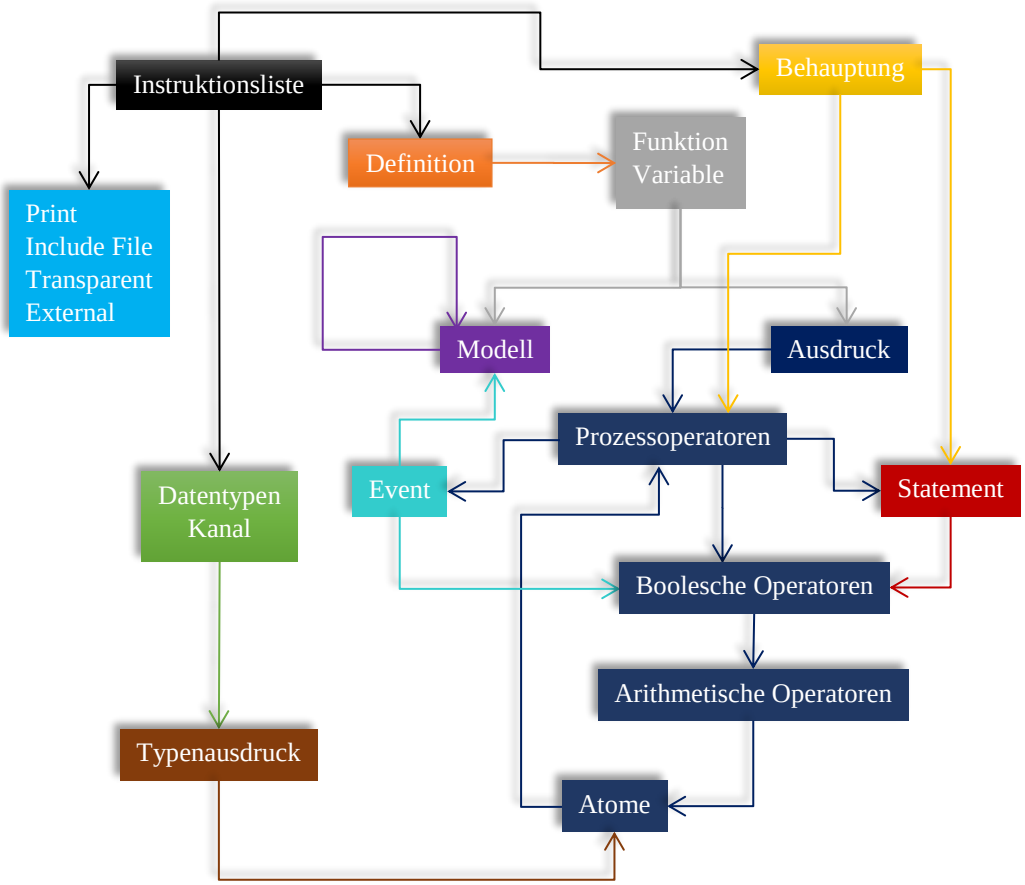


Abbildung 13 Aufbau der Grammatik von cspmj

Dabei repräsentiert jedes Rechteck einen Bereich des CST, der sich zusammenfassen lässt. Die Pfeile zeigen die jeweiligen Zusammenhänge zwischen den einzelnen Teilgrammatiken an. Beispielsweise enthält eine Behauptung z.B. sowohl ein Element aus der Teilgrammatik für Statements als auch einen Prozessausdruck. Dies wird durch die gelben Pfeile kenntlich gemacht.

5.7 Statement-Überprüfung

Die in Comprehensions vorkommende Statement-Liste, z.B. {x | x<-{1}, true}, besteht in CSP_M aus einer durch Komma getrennten Liste zweier Statement-Arten. Dazu gehören

- *Generator-Statements* `p<-a` oder `p:a`
- *Predicate-Statements* `b`

Dabei können beide Arten an jeder Stelle einer Liste stehen. Ein Modell jedoch kann eine Zeichenfolge enthalten, die sich nicht von einem Ausdruck unterscheiden lässt. So könnte z.B. `{1}^{2}` die Konkatenation zweier Mengen sein, die jeweils Zahlenausdrücke beinhalten. Jedoch wäre als Interpretation eine Anhang-Modell-Verknüpfung zweier Mengen-Modelle mit Zahlen-Modellen ebenso möglich. Die Unterscheidung kann somit erst nach Parsen des entsprechenden Ausdrucks erfolgen und hängt davon ab, ob ein `<-` folgt oder nicht. Weil aber Modelle und Ausdrücke durch verschiedene Teilgrammatiken erzeugt werden, ist dieser Teil der Sprache mehrdeutig. Um Konflikte zu verhindern, wurde eine große Kerngrammatik implementiert (Siehe Abbildung 13, dunkelblau). Diese beinhaltet als Untermenge alle Regeln, die für die

Erzeugung von Modellen erforderlich sind. Wird der Knoten für Generator-Statements betreten, so muss überprüft werden, ob `p` ein Modell ist. Dazu durchläuft ein AST-Visitor alle Ausdruckknoten und überprüft dabei, ob das Betreten erlaubt ist. Beim Aufruf eines reinen Ausdruckknotens (z.B. Addition) wird überprüft ob ein Modell erwartet wird. Ist dies der Fall, so erfolgt der Wurf einer `noPatternException`. Wird ein Hybrid-Knoten betreten (z.B. Concat/Append-Modell), so muss gegebenenfalls überprüft werden, ob ein Modell weitere Ausschlusskriterien erfüllen muss. Ein Mengen-Modell darf im Gegensatz zu Mengenausdrücken nur ein Argument erhalten. Wird ein reiner Modell-Knoten betreten (z.B. Platzhalter-Modell), aber kein Modell erwartet, dann erfolgt ebenfalls eine Fehlerausgabe.

5.8 LTL/-CTL-Formel-Überprüfung

Auch die Übersetzung einer LTL/-CTL-Assertion ist möglich. Im Gegensatz zu *cspmf* beinhaltet *cspmj* allerdings einen AST-Visitor, der mit Hilfe zweier SableCC-Parser die Korrektheit der angegebenen Formel überprüft. Aus diesem Grund ist die folgende Instruktion fehlerhaft und bewirkt die Ausgabe einer *TreeLogicException*: `assert 1 |= LTL: "ungueltig".`

5.9 Symbolsammler

Bevor mit der Erzeugung der Prolog-Datei begonnen werden kann, müssen alle relevanten Informationen zu verwendeten Variablen gesammelt werden. Dazu wird ein weiterer AST-Visitor, ein Objekt der Klasse *SymbolCollector.java* erstellt. In einem Objekt der Baum-Datenstruktur *ScopeTree.java* wird unter Anderem festgehalten, in welchem Sichtbarkeitsbereich sich der Algorithmus zur Laufzeit befindet. Jeder Bereich ist durch eine Nummer eindeutig identifizierbar. Diese wird als Schlüssel in einer Hashmap zusammen mit einem Wert gespeichert, welcher der Nummer des Vorgängerbereichs entspricht.

Die Folgende Instruktion verfügt über 6 Sichtbarkeitsbereiche:

```
A(x) = let B = x within {1|y<-{2}}
```

- 0: `A`
- 1: `(x)`
- 2: `let B within`
- 3: `x`
- 4: `x<-{2}`
- 5: `1`

Dabei wird eine Hashmap mit 5 Paaren angelegt: `{(1:0),(2:1),(3:2),(4:2),(5:4)}`. Bei jeder Variablendefinition wird ein Eintrag in einer Liste vom Typ `ArrayList<SymInfo>` erstellt. Eine Symbolinformation stellt ein Objekt der Klasse *Syminfo.java* dar, indem es folgende Informationen speichert:

- Bezeichner-Knoten (Liefert vor Allem Positionsangaben)
- Symboltyp – Beschreibung der Verwendungsart, z.B. `Function or Process`
- Symbolname – Originalbezeichnung der Variable
- Symbolreferenzname – Name der Variable inklusive Nummer und Unterstrichvorsatz (z.B. `_x2` für das zweite `x`, das gleichzeitig ein Modell ist)
- Sichtbarkeitsbereich – Nummer des Sichtbarkeitsbereichs, in dem die Variable aufgerufen oder definiert wird

Nach dem AST-Durchlauf sind die Informationen aller Variablen der geparsten CSP_M-Datei gespeichert.

5.10 Analyse von Neudefinitionen

Während der in *Kapitel 7.8* beschriebenen Symbolsammlung wird zeitgleich eine Renaming-Analyse durchgeführt. Unterschieden wird dabei zwischen einer horizontalen und einer vertikalen Renaming-Analyse.

Die horizontale Analyse untersucht, ob ein Variablen-Modell eines Sichtbarkeitsbereichs mehrfach auftaucht. So führen Beispielsweise folgende Instruktionen zu einer Fehlerausgabe `Redefinition of Identifier x`:

```
A(x,x) = 1
```

```
A = {1|x@@x<-{1}}
```

```
A(x^x) = 1
```

```
A = c?x?x -> STOP
```

Die vertikale Renaming-Analyse untersucht hingegen, ob eine Variablenzuweisung oder Funktionsdefinition in demselben Sichtbarkeitsbereich mehrfach auftaucht.

Folgende Instruktionen verursachen die Fehlerausgabe `Redefinition of Identifier A`:

```
A = 1
A = 2
```

```
A    = 1
A(x) = 2
```

```
A(x) = 1
A    = 2
```

```
B = let
          A = 1
          A = 2
    within STOP
```

Eine Ausnahme stellt dabei die Neudefinition von Funktionen dar. Erlaubt ist das Überschreiben von Funktionen für den Fall, dass die Erstdefinition die Vorgängerinstruktion ist. Eine Datei mit folgendem CSP_M-Code ist gültig:

```
A(x) = 1
A(x) = 2
```

Jedoch ungültig ist dieser Code:

```
A(x) = 1
B    = 2
A(x) = 3
```

Bei jedem Aufruf einer eingebauten Funktion oder Konstante wird in der Symbolliste ein Eintrag mit der Information `BuiltIn primitive` angelegt. Eine anschließende Definition des gleichen Symbols führt zu keinem Renaming-Fehler. In diesem Fall wird der Inhalt des alten Symboleintrages ersetzt und es beginnt die Gleichbehandlung der Variable in der Renaming-Analyse. Sofern kein Fehler vorhanden ist, wird das Symbollistenobjekt an den Prolog-Codegenerator übergeben.

5.11 Prolog-Codegenerierung

Der Prolog-Codegenerator ist ein AST-Visitor und Objekt der Klasse *PrologGenerator.java*. Wie auch der Symbolsammler verfügt der Codegenerator über eine Baumdatenstruktur zur Orientierung. Darüber hinaus wird eine Hilfsklasse *PrologTermOutput.java* verwendet, um die Erzeugung von Prolog-Termen zu formatieren. Nach dem Aufruf einer Methode (z.B. `printAtom(String)`) auf ein Objekt *p* der Hilfsklasse wird das Argument vollautomatisch, korrekt geklammert und interpunktiert in einen Puffer geschrieben (`StringWriter`). Auf diese Weise ist eine fehlerarme Übersetzung jedes einzelnen Knotens in grammatisch korrekte Prolog-Terme gewährleistet. Als Strategie zur Identifikation des Verhaltens von *cspmf* dient das Prinzip *Reverse Engineering*. Dazu wird für jeden AST-Knoten ein CSP_M-Code-Beispiel erstellt und anschließend mit *cspmf* übersetzt.

Anschließend erfolgt die Rekonstruktion des Terms durch Anpassung der Grammatik (AST-Listentransformationen) und Ergänzung des jeweiligen AST-Visitor-Knotens um die oben genannten Hilfsmethodenaufrufe. Nach einem manuellen Vergleich zwischen den Ausgabedateien von *cspmf* und *cspmj* wird ein neuer *jUnit*-Test erzeugt oder ein bestehender ergänzt. Anhand der unten aufgeführten Methoden ist die Übersetzung der CSP-Instruktion `A = 1+2` nachvollziehbar:

(1)

```
@Override
public void caseADefsStart(
              ADefsStart node)
{
    inADefsStart(node);
    {
        List<PDef> copy;
        copy = new ArrayList<PDef>(
                    node.getDef());
        for(PDef e : copy)
        {
           e.apply(this);
           if(!currentInChannel)
           {
               p.fullstop();
           }
           currentInChannel = false;
        }
    }
              [...]
```

(2)

```
@Override
public void caseAExpressionDef(
              AExpressionDef node)
{
    inAExpressionDef(node);
    if(node.getExp() != null)
    {
         node.getExp().apply(this);
    }
    printSrcLoc(node);
    p.closeTerm();
    outAExpressionDef(node);
}
```

(3)

```
@Override
public void caseAPatternExp(
              APatternExp node)
{
    inAPatternExp(node);
    p.openTerm("bindval");
    if(node.getPattern1() != null)
    {
        groundrep += 1;
        node.getPattern1().apply(
                              this);
        groundrep -= 1;
    }
    tree.newLeaf();
    if(node.getProc1() != null)
    {
        node.getProc1().apply(
                             this);
    }
    tree.returnToParent();
    outAPatternExp(node);
}
```

(4)

```
@Override
public void caseAWildcardPattern(
              AWildcardPattern node)
{
    inAWildcardPattern(node);
    p.printVariable("_");
    if(node.getWildcard() != null)
    {
        node.getWildcard().apply(
                                this);
    }
    outAWildcardPattern(node);
}
```

(5)

```
@Override
public void caseAAdditionExp(
              AAdditionExp node)
{
    inAAdditionExp(node);
    p.openTerm("+");
    if(node.getValExp() != null)
    {
        node.getValExp().apply(
                            this);
    }
    if(node.getValExp1() != null)
    {
        node.getValExp1().apply(
                             this);
    }
    p.closeTerm();
    outAAdditionExp(node);
}
```

(6)

```
@Override
public void caseANumberExp(
              ANumberExp node)
{
    inANumberExp(node);
    if(node.getNumber() != null)
    {
        node.getNumber().apply(
                            this);
        p.openTerm("int");
        int i;
        i = Integer.valueOf(
         node.getNumber().getText());
        p.printNumber(i);
        p.closeTerm();
    }
    outANumberExp(node);
}
```

Abbildung 14 Sechs Methoden aus dem Prolog-Codegenerator

Um einen korrekten Prolog-Ausdruck

```
'bindval'(_,+(int(1),int(2)),'src_span'(1,1,1,8,0,7))
```

zu erzeugen, ist der Aufruf der in *PrologTermOutput.java* vordefinierten Hilfsmethoden an der richtigen Stelle entscheidend. Nur ein kleiner Fehler verursacht eine Prädikatverdrehung, die zwar syntaktisch korrekt ist, aber beim Vergleich mit *cspmf* einen *jUnit*-Fehler verursacht. Die folgenden Schritte beschreiben die Entstehung des oben genannten Ausdrucks. Dabei stehen `IN` und `OUT` für das Betreten bzw. Verlassen von Knoten:

- `IN: ADefsStart`
- `IN: AExpressionDef`
- `IN: APatternExp`
  Hinzufügen von Teilausdruck `'bindval'(` durch Aufruf von `p.openTerm("bindval").`
- `IN: AWildcardPattern`
  Hinzufügen von Teilausdruck _ durch Aufruf von `p.printVariable("_").`
  `OUT: AWildcardPattern`
- `IN: AAdditionExp`
  Hinzufügen von Teilausdruck `,'+'(` durch Aufruf von `p.openTerm("+").`
- `IN: ANumberExp`
  Hinzufügen von Teilausdruck `'int'(1)` durch Aufruf von `p.printNumber(1)` und anschließend von `)` durch Aufruf von `p.closeTerm().`
  `OUT: ANumberExp`
- `IN: ANumberExp`
  Hinzufügen von Teilausdruck `,'int'(2)` durch Aufruf von `p.printNumber(2)` und von `)` durch Aufruf von `p.closeTerm().`
  `OUT: ANumberExp`
- `OUT: APatternExp`
- Hinzufügen von `)` durch Aufruf von `p.closeTerm().`
  `Out: AAdditionExp`
- Hinzufügen von `'src_span'(1,1,1,8,0,7)` durch Aufruf von `printSrcLoc(Node).`
  Hinzufügen von `(` durch Aufruf von `p.closeTerm().`
  `OUT: AExpressionDef`
- Hinzufügen von `.\r\n` bzw `.\n` durch Aufruf von `p.fullstop().`
  `OUT: ADefsStart`

5.12 Positionsangaben

Für die Interpretation von CSP_M-Code ist es erforderlich Positionsangaben zu bestimmten Knoten von Operatoren, Bezeichnern und ganzen Instruktionen an ProB zu liefern. Da SableCC den Ort von Knoten im Quellcode normalerweise nicht kennt, wurde eine spezielle SableCC-Version der Heinrich-Heine-Universität verwendet. Diese erweitert die Klasse *Node.java* um *PositionedNode.java*[4]. Auf diese Weise können durch Aufrufen verschiedener Methoden an dem Knoten des jeweiligen Token Index, Zeile und Spalte ermittelt werden. So enthält die Positionsangabe `'src_span'(1,1,1,8,0,7)` die Ortsinformation der gesamten oben beschriebenen CSP-Instruktion.

[4] `de.hhu.stups.sablecc.patch.PositionedNode`

Dabei haben die sechs einzelnen Argumente folgende Bedeutung für einen betrachteten Knoten `node`:

1. Zeile des ersten Zeichens
2. Spalte des ersten Zeichens
3. Zeile des letzten Zeichens
4. Spalte des letzten Zeichens
5. Index des ersten Zeichens
6. Index des letzten Zeichens - Index des ersten Zeichens

5.13 Variablenzuweisung/Erkennung ungebundener Variablen

Um Variablen, Funktionen oder Prozesse, die auf der rechten Seite von Definitionen auftauchen, zuordnen zu können, wurde ein Suchalgorithmus implementiert. Als Datenstrukturen kommen sowohl die Symbolliste aus der Symbolsammlung, als auch die Baumstruktur *ScopeTree.java* zum Einsatz. Letztere jedoch speichert neben den Sichtbarkeitsbereichen zusätzlich diejenigen Symbole, die zur Laufzeit in dem jeweiligen Bereich definiert wurden. Die hierzu gespeicherte Information ist jeweils ein Paar, das aus Symbolname und Symbolreferenzname (Beginn mit Unterstrich bei Variablen und Nummerierung, falls Nummer >1) besteht, z.B. `(x,x7)`. Die Zuweisung einer Variablen erfolgt durch Aufruf der Methode `printSymbol(String,Node)`, in welcher folgende drei Schritte solange wiederholt werden, bis das Symbol `x` gefunden oder der Sichtbarkeitsbereich 0 erreicht wurde.

1. Suche Symbol `x` in der Liste der definierten Variablen des aktuellen Sichtbarkeitsbereichs (*ScopeTree* zur Laufzeit)
2. Falls Symbol `x` nicht gefunden wurde, prüfe, ob die Definition erst später erfolgt. Durchsuche dazu die Symbolliste des Symbolsammlers nach Symbolen `x`, denen der aktuelle Sichtbarkeitsbereich zugeordnet ist
3. Falls `x` immer noch nicht gefunden wurde, kehre zum vorherigen Sichtbarkeitsbereich zurück

Ist der Bezeichner nach Abbruch der o.g. Schleife bekannt, so kann der Prolog-Term ergänzt werden. Ist er jedoch unbekannt, muss untersucht werden, ob es sich bei ihm um eine eingebaute Funktion handelt. Ist dies nicht der Fall, erfolgt eine Fehlerausgabe `Unbound Identifier x.`

Im Gegensatz zum *cspmf-Parser*, der Modelle und Definitionen in zwei Schritten nummeriert, werden in *cspmj* alle Bezeichner in nur einem AST-Durchlauf mit Nummern versehen. Dieses Vorgehen ist hinsichtlich der Laufzeit positiv zu bewerten und verursacht keine Nachteile.

5.14 Fehlerbehandlung

Alle möglichen Fehlerarten eines Übersetzungsvorgangs werden in der Methode `parsingRoutine(...)` der Hauptklasse `CSPMparser.java` abgefangen.
Die Ausdifferenzierung einzelner Fehlerarten ist hinsichtlich weiterer Entwicklungsbemühungen und der Benutzerfreundlichkeit von größter Bedeutung. Aus diesem Grund wurde für alle Möglichen Fehler jeweils eine Klasse angelegt, die `java.lang.Exception` erweitert. Mögliche Fehlerausgaben bei der Interpretation einer CSP-Datei sind:

- `LexerException`
  Fehler bei der Identifikation eines Tokens
- `ParserException, IOException`
  Verletzung syntaktischer Regeln, andere Fehler beim durchlaufen des AST

- `RenamingException`
  Ungültige Neudefinition in betrachtetem Sichtbarkeitsbereich
- `UnboundIdentifierException`
  Aufruf eines ungebundenen Bezeichners
- `NoPatternException`
  Ungültiger Ausdruck für ein *Modell* in einem *Generatoren-Statement*
- `TriangleSubstitutionException`
  Fehler bei der Umwandlung von Dreiecksklammern in Sequenzklammern oder Vergleichsoperatoren
- `IncludeFileException`
  Fehler beim Importieren einer CSP-Datei.
- `FileNotFoundException`
  Die zu parsende Datei wurde nicht gefunden
- `TreeLogicException`
  Eine LTL- oder CTL-Assertion hat eine Formel, die syntaktische Fehler aufweist

5.15 Befehle

Die von Gradle generierte *cspmj.jar* kann über eine Kommandozeile ausgeführt werden. Folgende Befehle werden akzeptiert:

- `java -jar cspmj.jar -parse dateiname.csp`
  Parse eine CSP-Datei dateiname.csp und generiere eine Prolog-Datei mit dem Namen dateiname.csp.pl
- `java -jar cspmj.jar -parse eingabe.csp --prologOut=ausgabe.csp`
  Parse eine CSP-Datei *eingabe.csp* und generiere eine Prolog-Datei mit dem Namen *ausgabe.csp.pl*
- `java -jar cspmj.jar -parseAll`
  Durchsuche das aktuelle Verzeichnis und alle Unterverzeichnisse nach Dateien, welche die Endung *.csp* haben. Parse alle gefundenen Dateien und lege entsprechende Prolog- Dateien mit dem gleichen Anfangsnamen und der Endung *.pl* an.
- `java -cp build/classes/main PerformanceTest suchpfad egebnis-pfad`
  Rufe *cspmj.jar* und *cspmf.exe* für alle Dateien mit Endung *.csp* in `suchpfad` und Unterordnern auf. Halte die Zeit fest, die jeweils zum Parsen benötigt wird und lege eine Vergleichsübersicht in `ergebnis-pfad` an.

5.16 Typechecking

Im frühen Entwicklungsstadium dieses Projekts wurde die Implementierung eines Typecheckers in Erwägung gezogen. Im weiteren Verlauf stellte sich jedoch heraus, dass die drohende Zeitknappheit und die Komplexität des Typecheckings für eine Spezifikationssprache dies verhindern würde. Dennoch wurde ein Typechecker auf Basis des automatisch generierten AST-Visitors von SableCC erstellt, bevor der CST durch AST-Transformationen kompaktifiziert wurde. Im Rahmen der Entwicklung dieses Typecheckers wurde eine Methode zur Aufschlüsselung von Datentypen entworfen. Wird zum Beispiel ein Ereignis `channel c:{1}.{true}` definiert, dann ist `c` vom Typ `Int=>Bool=>Event`. Der implementierte Algorithmus `reduce(String)` kann überprüfen, ob eine Eingabe vom Typ `Dotable` ist oder die Zwangsbedingung `Complete` erfüllt. Beispielsweise ist die Eingabe `c.1.true` vom Typ `Event`. Man schreibt auch `c.1.true :: Event`. Der Typ `Event` ist atomar und enthält keine Pfeile (`=>`) mehr. Das bedeutet automatisch, `c.1.true` erfüllt die Zwangsbedingung `Complete`. Die Eingabe `c.1` hingegen

erfüllt die Zwangsbedingung nicht, da sie sich noch über Anwendung des Dot-Operators zu einem Ereignis erweitern lässt, oder kurz gesagt noch Pfeile im Typ auftauchen (`c.1 :: Bool => Event`). Aus der Eingabe `c.1` (Typen: `c :: Int=>Bool=>Event` und `1:: Int`) setzt der Algorithmus einen neuen Typ zusammen (`Int=>Bool=>Event.Int`), dreht die Typenkette vor dem Punkt um, löscht das Paar Int.Int und dreht die Typenkette wieder zurück. Zum Schluss bleibt der Typ von `c.1` übrig. Möglich ist dies mit beliebig langen Ketten von Ausdrücken, die mit Dot zusammengesetzt wurden. Das obige Beispiel zeigt, dass der entworfene Algorithmus vor allem für die Überprüfung von Typen beim Prefixing elementar wichtig ist. Zur Erweiterung des Typecheckers ist ein Neuaufbau des AST notwendig, da die meisten Knoten nicht mehr der aktuellen Version des Parsers entsprechen.

5.17 Performance

Um einen Nachweis über die Praxistauglichkeit von CSPMJ zu liefern, eignet sich nicht nur der Vergleich des Prolog-Codes beider Parser, sondern auch ein Vergleich der Laufzeit. Aus diesem Grund wurde eine Klasse *PerformanceTest.java* erstellt. Der Kommandozeilenbefehl

```
java -cp build/classes/main PerformanceTest suchordner testergebnis
```

durchsucht den Ordner `suchordner` nach CSP-Dateien und führt für jede gefundene Datei $file_i$, $i \in [0;129[$ sowohl *cspmj.jar* als auch *cspmf.exe* aus. Dabei wird jeweils in Sekunden auf drei Nachkommastellen genau festgehalten, wie lange die Übersetzung dauert - $time_{cspmf}(i)$ *und* $time_{cspmj}(i)$. Anschließend ensteht ein Eintrag in einer Textdatei mit dem Namen der übersetzten Datei, den beiden Übersetzungszeiten und dem Vergleichsquotienten

$$\frac{time_{cspmj}(i)}{time_{cspmf}(i)}$$

Der vorletzte Eintrag `TOTAL` hat die Felder

(1) $\sum_{i=0}^{128} time_{cspmf}(i)$ (2) $\sum_{i=0}^{128} time_{cspmj}(i)$

und den Vergleichsquotienten $\frac{(2)}{(1)}$. Der Eintrag `ParseAll` enthält den Wert aus Summe (2), die Übersetzungszeit aller 130 Dateien mit nur einem Aufruf von *cspmj* ($time_{parseAll}$) und dem Vergleichsquotienten

$$\frac{time_{parseAll}}{\sum_{i=0}^{128} time_{cspmj}(i)}$$

Auffällig ist hier, dass $time_{cspmf}(i) << time_{cspmj}(i)\ \forall i$. Dies ist nicht darauf zurück zu führen, dass der Parser unabhängig von einer Laufzeitvorstellung entwickelt wurde. Untersuchungen mit dem `-parseAll`-Argument von *cspmj* haben ergeben, dass nicht das Parsen und Übersetzen viel Zeit kostet, sondern ein Aufruf der Java Virtual Machine. Wird diese für die Abwicklung der Parsing-Routine nur einmal aufgerufen, dann benötigt der Vorgang nur noch 30% mehr Zeit als *cspmf*. Weitere Entwicklungsschritte könnten zu deutlichen Verbesserungen führen. So nimmt alleine die Substitution von Dreiecksklammern mit dem Ansatz der Exhaustionsmethode in einer Datei 1,3 Sekunden in Anspruch. Die Implementierung eines AST-Visitors und das Überschreiben der SableCC-Filtermethode wäre hier ein Ansatz, der eine erhebliche Zeiteinsparung bewirken würde.

Die oben beschriebene Übersicht eines Tests mit 130 CSP-Dateien befindet sich im Anhang.

6. Fazit und Ausblick

6.1 Bewertung der aktuellen Funktionalität

Vorteile

Die Optimierung der lexikalischen Analyse durch überschreiben der SableCC-Filtermethode reduziert die Laufzeit des neuen CSP_M-Parsers. Nahezu jede CSP-Testdatei in deutlich weniger als einer Sekunde übersetzt. Aus diesem Grund ist in der Anwendungspraxis mit keinen Einschränkungen zu rechnen. Aufgrund des Entwicklungszeitraumes, der fünf Jahre nach Fertigstellung von *cspmf* ansetzt, ist die Funktionalität dem alten Parser von 2011 überlegen. Ein Nachweis über die Unterstützung der Plattformen Linux (32 und 64 Bit), OS X und Windows sowie die ProB-Anbindungsfähigkeit ist anhand der Testung und Abgleichung sämtlicher Funktionen mittels *jUnit* erbracht. Ein weiterer Vorteil ist die Behebung diverser Bugs gegenüber *cspmf*, wie z.B. ein Problem, das in *cspmf* zu einer fehlerhaften Darstellung des Backslash-Zeichens führte. Da Java die am weitesten verbreitete Programmiersprache der Welt ist, können Änderungen von jedem Programmierer durchgeführt werden. Die Bemühung, einen verständlichen und gut dokumentierten Java-Code zu implementieren, begünstigt diesen Vorteil zusätzlich.

Nachteile

Leider ist die Lösung für das Parsen von Vergleichen innerhalb von Sequenzen unbefriedigend langsam. So erhöhen typeninkorrekte Kettenvergleiche die Laufzeit im schlimmsten Fall so stark, dass eine Fertigstellung der Übersetzung nicht mehr zeitnah erfolgen kann. Des Weiteren verfügt der Parser über einige unvollständige Konstrukte, die aus der Dokumentation von FDR3.4 übernommen wurden. Diese werden im Prolog-Erzeuger noch nicht berücksichtigt, können jedoch ohne größeren Aufwand vervollständigt werden. Das Ziel der Arbeit sollte die Entwicklung eines Parsers sein, der mit FDR 3.4 kompatibel ist. Dies ist nur zum Teil gelungen. Vieles Funktionen aus *libcspm* konnten aus zeitlichen Gründen noch nicht implementiert werden. Eine vollständige Abgleichung mit *cspmf* war nicht möglich, da sich die Darstellung bestimmter Konstrukte in FDR in den vergangenen Jahren leicht verändert hat. So können z.B. im Gegensatz zu *cspmf* alle Konstanten überschrieben werden und müssen deshalb anders übersetzt werden als bisher. Die Stabilität und Kontinuität ist ohne Testung durch mehrere Anwender nicht zu gewährleisten.

6.2 Zukünftige Entwicklung

Folgende Punkte könnten in Zukunft zu einer Verbesserung der Leistung und Funktionalität von *cspmj* führen oder stehen noch aus, um die Integration in ProB zu gewährleisten:

- Implementierung aller Funktionen von *libcspm*, die in der FDR 3.4-Dokumentation ausgeschrieben sind. Dazu zählen die *transparent*- und *external*-Funktionen (nur im Backend von ProB zu berücksichtigen), *Typenannotationen*, *Assertion-Optionen* wie die *Partial Order Reduction*, *Module*, *parametrisierte Module* und *Zeitstrecken*.
- Entwicklung eines Laufzeitfreundlichen Algorithmus oder AST-Visitors zur Substitution von Dreiecksklammern.
- Änderungen am Back-End von *ProB* vornehmen und Einbau der Neuheiten
- Ersetzen von laufzeithungrigen Methoden wie `contains(String)` und regulären Java-Ausdrücken
- Komprimieren des *Concrete Syntax Tree* durch Zusammenfassen von Regeln und häufigere Anwendung von Quantifizierern
- Vervollständigung des Typecheckers und Anwendung des *Hindley-Milner* -Verfahrens

7. Literaturverzeichnis

[Ba95] Barrett, G.: Model checking in practice: The T9000 Virtual Channel Processor. IEEE Transactions on Software Engineering, 1995.

[BPS99] Buth, B.; Peleska, J.; Shi, H.: Combining methods for the livelock analysis of a fault-tolerant system. Technology, Proceedings of the 7th International Conference on Algebraic Methodology and Software, 1999.

[Bu97] Buth, B. et al.: Deadlock analysis for a fault-tolerant system. Technology, Proceedings of the 6th International Conference on Algebraic Methodology and Software, 1997.

[Fo11] Fontaine, M.: A Model Checker for CSPM. Dissertation, Düsseldorf, 2011.

[Fr] Freiberg, B.: Einführung in Communicating Sequential Processes. Seminararbeit, Aachen.

[Ga04] Gagnon, É. M.: Specifying Binary and Unary Operator Precedence. http://www.sable.mcgill.ca/listarchives/sablecc-list/msg01208.html, 21.07.2016.

[Ga98] Gagnon, É.: SABLECC, AN OBJECT-ORIENTED COMPILER FRAMEWORK. Masterarbeit, Montreal, 1998.

[Gi16a] Gibson-Robinson, T.: Built-In Definitions - FDR 3.4.0 documentation. Compression Functions. https://www.cs.ox.ac.uk/projects/fdr/manual/cspm/prelude.html#compressions, 08.08.2016.

[Gi16b] Gibson-Robinson, T.: Type System - FDR 3.4.0 documentation. http://www.cs.ox.ac.uk/projects/fdr/manual/cspm/types.html, 03.08.2016.

[Gi16c] Gibson-Robinson, T.: Definitions - FDR 3.4.0 documentation. https://www.cs.ox.ac.uk/projects/fdr/manual/cspm, 20.07.2016.

[HA15] Hans Dockter; Adam Murdoch: Gradle User Guide. https://docs.gradle.org/current/userguide/userguide.pdf.

[Ho78] Hoare, C.: Communicating Sequential Processes, Belfast, 1978.

[LB07] Leuschel, M.; Butler, M.: ProB: An Automated Analysis Toolset for the B Method, Düsseldorf, 2007.

[LF08] Leuschel, M.; Fontaine, M.: Probing the Depths of CSP-M: A new fdr-compliant Validation Tool, Düsseldorf, 2008.

[Lo96] Lowe, G.: Breaking and fixing the Needham-Schroeder public-key protocol using FDR. Springer-Verlag, 1996.

[Mi06] Mike Clark: JUnit FAQ. http://junit.sourceforge.net/doc/faq/faq.htm, 03.08.2016.

[Mi16] Michael Leuschel: The ProB Animator and Model Checker - Institut für Software und Programmiersprachen. https://www3.hhu.de/stups/prob, 19.07.2016.

[Ro05] Roscoe, A. W.: The Theory and Practice of Concurrency. Pearson, 2005.

[Sc98] Scattergood, B.: The Semantics and Implementation of Machine-Readable CSP. Dissertation, 1998.

8. Abbildungsverzeichnis

Anhang

Datei	$time_{cspmf}$	$time_{cspmj}$	Vergleichsquotient
altbitprotocol.csp	0.041	0.275	6.732
bankv1.csp	0.033	0.279	8.523
bankv2.csp	0.032	0.314	9.792
BigUnionInterChannelTest.csp	0.023	0.2	8.826
BigUnionInterTests.csp	0.02	0.204	10.052
BLinkTest.csp	0.02	0.212	10.821
Buffer.csp	0.021	0.218	10.187
Buffer_hide.csp	0.02	0.192	9.46
buses.csp	0.02	0.21	10.474
ClosureCompTests.csp	0.023	0.198	8.558
comments.csp	0.018	0.207	11.507
comment_eof.csp	0.017	0.181	10.583
ComplicatedChannelGuards.csp	0.019	0.202	10.641
ComplicatedLinkedParallel.csp	0.023	0.2	8.869
ComplicatedLinkedParallel2.csp	0.02	0.206	10.295
ComplicatedSync.csp	0.025	0.233	9.424
dotpattern.csp	0.019	0.202	10.625
dtype.csp	0.017	0.16	9.389
emptySet.csp	0.017	0.181	10.598
EnumerationTests.csp	0.024	0.244	10.129
exp.csp	0.019	0.192	10.358
ExpressionsNewlinesBetween.csp	0.018	0.184	10.424
Fibonacci.csp	0.019	0.213	11.067
FM08review.csp	0.019	0.202	10.695
frogs2.csp	0.029	0.245	8.609
functional_override.csp	0.017	0.167	9.721
GenericBuffer1.csp	0.025	0.24	9.579
hanoi.fix.csp	0.023	0.271	11.585
inctest.csp	0.017	0.191	11.437
inctest2.csp	0.017	0.202	12.082
independent.csp	0.019	0.201	10.747
Lambda.csp	0.017	0.185	10.803
LambdaComplex.csp	0.023	0.224	9.665
LambdaSimple.csp	0.019	0.203	10.702
LetFunctionPassedOut.csp	0.019	0.207	10.738
LetMultipleEquations.csp	0.021	0.217	10.344
LetMultipleFuns.csp	0.027	0.247	9.104
LetTests.csp	0.025	0.239	9.54
LetTestsChannel.csp	0.022	0.225	10.131
letwithin.csp	0.018	0.19	10.571
lokal_definitions.csp	0.018	0.178	9.964
mbuff.csp	0.026	0.306	11.644
McCarthy.csp	0.02	0.235	12.04
microwave.csp	0.019	0.184	9.583
nametype_test.csp	0.02	0.196	9.896
nametype_test2.csp	0.019	0.186	9.661
NameWithApostrophe.csp	0.019	0.199	10.567
NastyNonDet.csp	0.022	0.202	9.187
nestedOps.csp	0.017	0.184	10.567
newdebug.csp	0.018	0.214	11.983
newlinesBetween.csp	0.017	0.166	9.498
newmbuff.fix.csp	0.029	0.459	16.013
occursCheck.csp	0.017	0.22	12.862
oopseq.csp	0.032	0.294	9.06
oopsla.csp	0.031	0.328	10.468
PairMedium.csp	0.02	0.215	10.636
PairSimple.csp	0.019	0.204	10.571
ParserIssues.csp	0.021	0.193	9.385
PatMatchPair.csp	0.019	0.205	10.854

Datei	$time_{cspmf}$	$time_{cspmj}$	Vergleichsquotient
PatMatchTuple.csp	0.019	0.202	10.696
PatMatchTupleComplex.csp	0.025	0.265	10.552
peterson.csp	0.029	0.232	8.131
phils.fix.csp	0.021	0.201	9.382
PrimedVar.csp	0.017	0.181	10.531
prioProb.csp	0.017	0.164	9.487
prize.csp	0.021	0.215	10.452
prologTest.csp	0.017	0.203	11.692
prologTest2.csp	0.017	0.181	10.436
protocol.fix.csp	0.054	0.591	11.02
put12.csp	0.026	0.208	8.051
ReadMe.csp	0.017	0.186	10.825
RecursiveDatatype.csp	0.019	0.2	10.702
ReplicatedAlphParallel.csp	0.024	0.224	9.157
ReplicatedInterleave.csp	0.019	0.201	10.831
ReplicatedSequential.csp	0.02	0.19	9.486
ReplicatedSharing.csp	0.019	0.202	10.621
RepWithGuard.csp	0.018	0.185	10.213
same_identifier_error.csp	0.017	0.183	10.492
SeqCompTests.csp	0.024	0.233	9.902
SeqRangeTests.csp	0.021	0.216	10.281
SeqTests.csp	0.024	0.225	9.236
SeqType.csp	0.021	0.206	10.023
SequenceComprTests2.csp	0.021	0.216	10.135
sequences.csp	0.018	0.207	11.349
sequences2.csp	0.018	0.184	10.24
SequentialRouter.csp	0.033	0.284	8.592
SetCompAdvanced.csp	0.024	0.206	8.499
SetCompComplicated.csp	0.023	0.224	9.901
SetCompComplicated2.csp	0.021	0.211	10.128
SetCompTests.csp	0.03	0.249	8.238
SetCompWithLambda.csp	0.02	0.184	9.348
SetTests.csp	0.036	0.268	7.454
simple.csp	0.02	0.221	11.04
SimpleAlphaPar.csp	0.023	0.219	9.587
SimpleCHAOS.csp	0.023	0.224	9.803
SimpleCompLinkedPar.csp	0.022	0.193	8.894
SimpleCompRenaming.csp	0.019	0.183	9.75
SimpleGenGen.csp	0.021	0.199	9.412
SimpleGenGenForFDR.csp	0.022	0.201	9.168
SimpleGetSet.csp	0.018	0.198	10.742
SimpleIfThenElse.csp	0.021	0.211	10.06
SimpleInterleaveSkipTest.csp	0.02	0.207	10.331
SimpleInterleaveSkipTest2.csp	0.019	0.204	10.49
SimpleInterruptTimeout.csp	0.021	0.2	9.445
SimpleInterruptTimeout2.csp	0.02	0.189	9.619
SimpleIntTim_statespace.csp	0.017	1.358	80.807
SimpleLinkedParallel.csp	0.022	0.199	8.911
SimpleLinkedParallel2.csp	0.026	0.258	9.838
SimplePatMatch.csp	0.027	0.231	8.471
SimpleRenaming.csp	0.024	0.223	9.491
SimpleRenaming2.csp	0.022	0.22	10.155
SimpleRepAlphParallel.csp	0.022	0.22	9.805
SimpleReplicated.csp	0.02	0.208	10.561
SimpleRepLinkedParallel.csp	0.02	0.223	10.95
SimpleSeqComp.csp	0.019	0.204	10.486
SimpleSubsets.csp	0.02	0.206	10.344
SimpleTransparent.csp	0.019	0.202	10.893
speareate.csp	0.018	0.188	10.356
speareate_error.csp	0.018	0.186	10.284
StatementPattern.csp	0.017	0.204	12.206
StrangeAgents.csp	0.028	0.234	8.358
student1.csp	0.02	0.208	10.572
subtype_ex.csp	0.023	0.202	8.656

Datei	$time_{cspmf}$	$time_{cspmj}$	Vergleichsquotient
subtype_nametype_ex.csp	0.023	0.222	9.618
TestDotPat.csp	0.02	0.19	9.7
tickets.csp	0.02	0.201	10.223
unary_minus.csp	0.017	0.181	10.501
verysimple.csp	0.017	0.184	10.611
VerySimpleTimeout.csp	0.018	0.197	10.684
vm2.csp	0.018	0.2	10.936
TOTAL	2.803	29.256	10.437
ParseAll	2.803	3.726	1.329

Testumgebung

Betriebssystem:	Windows 10 Pro 64-bit (10.0, Build 10586)
Prozessor:	Intel(R) Core(TM) i5-4570 CPU @ 3.20GHz (4 CPUs), ~3.2GHz
Arbeitsspeicher:	8192MB RAM
Speicher:	ADATA SSD S511 120GB